Rumford, Max

Working Paper

Robotik im Anlagevermögen: Algorithmenbasiertes Handeln in der Versicherungsbranche

Arbeitspapiere der FOM, No. 77

Provided in Cooperation with:
FOM Hochschule für Oekonomie & Management gGmbH

Suggested Citation: Rumford, Max (2021) : Robotik im Anlagevermögen: Algorithmenbasiertes Handeln in der Versicherungsbranche, Arbeitspapiere der FOM, No. 77, ISBN 978-3-89275-165-6, MA Akademie Verlags- und Druck-Gesellschaft mbH, Essen

This Version is available at:
https://hdl.handle.net/10419/230666

FOM Hochschule

Arbeitspapiere der FOM

Nr. 77

Robotik im Anlagevermögen: Algorithmenbasiertes Handeln in der Versicherungsbranche

~

Max Rumford

Max Rumford

Robotik im Anlagevermögen:
Algorithmenbasiertes Handeln in der Versicherungsbranche

Arbeitspapiere der FOM, Nr. 77

Essen 2021

ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)
ISBN 978-3-89275-164-9 (Print) – ISBN 978-3-89275-165-6 (eBook)

Dieses Werk wird herausgegeben von der FOM Hochschule für Oekonomie & Management gGmbH

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

MA Akademie Verlags-
und Druck-Gesellschaft mbH
Leimkugelstraße 6, 45141 Essen
info@mav-verlag.de

Max Rumford

Robotik im Anlagevermögen – Algorithmenbasiertes Handeln in der Versicherungsbranche

Arbeitspapiere der FOM Hochschule für Oekonomie & Management

Nr. 77, Essen 2021

ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

ISBN 978-3-89275-164-9 (Print) – ISBN 978-3-89275-165-6 (eBook)

Vorwort

Die Versicherungswirtschaft und speziell die Sparte Lebensversicherung leidet seit Beginn der Finanzkrise im Jahr 2007 unter einem ungünstigen Zinsspread, der sich aus dem Garantiezins für die Versicherungskundinnen und -kunden und den erzielbaren Renditen für Anleihen am Markt errechnet. Auslöser des Niedergangs des ehemals dominierenden Produkts war der Wegfall des steuerlichen Privilegs für die Kapitallebensversicherung, der durch die Absenkung des Garantiezinses auf aktuell null Prozent verschärft wird. Insbesondere die Kapitalversicherung hat im Jahr 2019 teilweise über 5 Prozent an Hauptverträgen verloren. Auch die Rentenversicherung hat Hauptverträge im signifikanten Ausmaß abgegeben. Die aktuelle Corona-Krise und die hieraus enorm steigenden Verschuldungsgrade der europäischen Staaten und der Bundesrepublik Deutschland könnten die Versicherungsbranche wieder anleiten, verstärkt in die Assetklasse Aktien zu investieren.

Die Gestaltung von innovativen Versicherungskonzepten und automatisierten Prozessen ist für die nachhaltige Organisation eines langfristig tragfähigen gesellschaftlichen Wohlstands unerlässlich, der auf der stetigen Erneuerung von bestehenden Geschäftsmodellen und u. a. auf der Neugründung von Unternehmen (Insurtecs), aber auch auf potentiell neuen Anlagestrategien in Deutschland beruht und insbesondere die private Vorsorge der Bundesbürgerinnen und -bürger zum Inhalt hat.

Auf der Grundlage der genannten, zum Teil ungünstigen Auswirkungen, bedingt durch das aktuelle Garantiezinsniveau von null Prozent und eine überschaubare Überschussbeteiligung im Mittel von 2,5 Prozent in der Zukunft, skizziert Herr Max Rumford nicht nur die dramatische Problemstellung in der Kapitalanlage von Versicherungskonzernen neu, sondern diskutiert auch die erheblichen Risiken der aus den Anforderungen von Solvency II resultierenden strategischen Kalküle. Herr Rumford widmet sich folgerichtig dieser Thematik und schlägt als versicherungsspezifischen Lösungsbestandteil die Robotik im Anlagevermögen in Form des algorithmenbasierten Handelns in der Versicherungsbranche für die Sparte Komposit und die Lebensversicherungssparte vor.

Die Masterthesis von Herrn Rumford beruht auf mehreren Forschungskomponenten. Die erste Komponente führt die Leserinnen und Leser zur wissenschaft-

lich exakten Aufbereitung des aktuellen Forschungsstands. Die zweite Komponente des Forschungsdesigns diskutiert u. a. das Ziel der Branche, Insurance-Kosten zu senken und z. B. im Cashflow Underwriting gewinnbringende Kapitalanlage zu betreiben. Demzufolge wird eine Methode für die vollautomatisierte kurz- bis mittelfristige Kapitalanlage entwickelt, die die Belange der Versicherungswirtschaft als Kapitalanleger berücksichtigt und die Restriktionen wie das VAG und Solvency II beachtet. Diese an der Versicherungsbranche und ihren speziellen Kapitalanlageverordnungen gespiegelten Anforderungen werden als lauffähiges Programm in der Programmiersprache JAVA implementiert. Die dritte und tragende Komponente ist die Entwicklung eines teilspezifischen Trading Systems (TS), welches dem „Backtesting" zugeführt wird, um das Trading System zu validieren. Im Ergebnis erlaubt die Nutzung dieses vollumfänglich programmierten Tradingsystems einen einfachen Einstieg in die Diversifikation der Kapitalanlage eines jeden Versicherers.

Herr Rumford stellt in der Summe eindrucksvoll unter Beweis, dass es ihm gelungen ist, sich als Wirtschaftsinformatiker in eine komplexe und aktuelle Thematik umfassend einzuarbeiten und einen wertvollen Beitrag zur wissenschaftlichen und gleichermaßen praktischen Lösung in diesem Themenfeld zu leisten.

Kaiserslautern, im Januar 2021

Prof. Dr. Markus Dirk Ebner
Professor für Wirtschaftsinformatik und Finance
FOM Hochschule in Mannheim

Inhalt

Abweichung der Druck- und Online-Versionen:

Die Anhänge *A3 (Programmcode)* und *A4 (Testcode)* auf den Seiten 123–350 werden aufgrund ihres Umfangs in der Druckversion nicht angezeigt. Sie sind jedoch in der Online-Version zugänglich: https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122

Über den Autor

Max Rumford M.Sc. hat ein duales Bachelorstudium der Wirtschaftsinformatik, Schwerpunkt Application Management, an der Dualen Hochschule Baden-Württemberg sowie ein Masterstudium in IT-Management an der FOM Hochschule in Mannheim absolviert. Während seines berufsbegleitenden Erststudiums arbeitete er als Anwendungsentwickler in Ausbildung und ist seitdem bei der Continentale Krankenversicherung (vormals Mannheimer Versicherungen) im Bereich der zentralen Anwendungsentwicklung festangestellt.

Autorenkontakt:
E-Mail: max.rumford@gmx.net

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Formeln

Symbole und Indizes

Symbole

Symbol	Bedeutung
A	Decay alter EWMA-Werte
$c_{ante,t}$	Kapital vor einem Handel zum Zeitpunkt t
$c_{post,t}$	Kapital nach einem Handel zum Zeitpunkt t
E_t	EWMA zum Zeitpunkt t
$E_{v,x,t}$	Volatilitäts-EWMA zum Basiswert x zum Zeitpunkt t
F	Forecast-Skalar
$f_{ber,t}$	Bereinigter Forecast zu einem Zeitpunkt t
$f_{res,t}$	Resultierender Forecast zu einem Zeitpunkt t
$f_{roh,t}$	Roher Forecast zu einem Zeitpunkt t
$f_{scal,t}$	Skalierter Forecast zum Zeitpunkt t
$f_{sum,t}$	Summierter Forecast zum Zeitpunkt t
$i_{x,t}$	Rendite einer Datenreihe x zum Zeitpunkt t
$k_{x,t}$	Kurswert einer Datenreihe x zum Zeitpunkt t
M_d	Diversifikationsfaktor
$M_{p,x}$	Produktpreisfaktor einer Position p zur Datenreihe x
$p_{p,t}$	Produktpreis einer Position p zum Zeitpunkt t
$r_{i,x}$	Inverse Durchschnittskorrelation einer Datenreihe x
$r_{x,y}$	Korrelation zweier Datenreihen x und y
$\overline{r_x}$	Durchschnittliche Korrelation einer Datenreihe x
$s_{p,t}$	Stückzahl einer Position p zum Zeitpunkt t

$s_{voll,p,t}$	Volle Positionsgröße einer Position p zum Zeitpunkt t
v_n	Volatilität einer Stichprobe mit n Beobachtungen
w_x	Gewichtung einer Datenreihe x
σ_n	Standardabweichung einer Stichprobe mit n Beobachtungen
$\sigma_{x,E,t}$	EWMA Standardabweichung einer Datenreihe x zum Zeitpunkt t

Indizes

Index	Bedeutung
$ante$	Vor einem Handelszeitpunkt
ber	Bereinigt
d	Diversifikation
E	EWMA-basiert
$invers$	Inversion
n	Stichprobengröße
p	Position
$post$	Nach einem Handelszeitpunkt
res	Resultierend
roh	Roh
$scal$	skaliert
sum	summiert
t	Zeitpunkt
v	Volatilität
$voll$	Volle Position
x, y	Datenreihen

Abkürzungsverzeichnis

Abkürzung	Bedeutung
CSV	Comma separated values – kommaseparierte Werte
DAX	Deutscher Aktienindex
ER	Emerging Risk
EWMA	Exponentially weighted moving average – Exponentiell gewichteter gleitender Mittelwert
EWMAC	Exponentially weighted moving average crossover – Kreuzungspunkt exponentiell gewichteter gleitender Mittelwerte
JDK	Java Development Kit
KI	Künstliche Intelligenz
OTC	Over The Counter – Direkthandel zwischen zwei Marktteilnehmern
POM	Project Object Model
S&P 500	Standard & Poor‘s 500
S&P	S&P 500 Aktienindex
STOXX	EURO STOXX 50 Aktienindex

1 Einführung

Kapitalanlage ist, neben dem Kerngeschäft des Verkaufs von Versicherungspolicen, ein wichtiger Faktor zum Erhalt und Ausbau finanzieller Freiheit einer Vielzahl von Unternehmen geworden. Insbesondere die Versicherungsbranche widmet sich diesem Thema intensiv: 2018 gehörte sie mit einem investierten Geldvolumen von knapp „1,7 Billionen Euro … zu den größten institutionellen Investoren in Deutschland“ und legte ihr Kapital breit gestreut und langfristig orientiert an.[1] Solch große Investitionssummen werden staatlich reglementiert und von entsprechenden Stellen geprüft.

1.1 Rahmenbedingungen

Das Netz gesetzlicher Vorgaben reglementiert die Möglichkeiten der Versicherungsbranche dabei recht eng: Hohes Risiko ist stets zu vermeiden. Ein Missachten der Vorgaben ist mit empfindlichen Strafen belegt – bis hin zum Verlust der Versicherungslizenz, wenn das durch Regelungen wie die europäische Richtlinie 2009/128/EG[2], besser bekannt unter dem Namen *Solvency II,* vorgeschriebene Mindestkapital unterschritten wird.[3] So stark solche Regelungen den Versicherer einschränken, so sehr stärken sie auch die Versicherungskundinnen und -kunden, die darauf vertrauen, die Vorteile durch ihren Versicherungsschutz nutzen zu können.[4]

Dass dies nicht immer der Fall war, zeigen Fälle wie die *Mannheimer Leben*, die sich durch ihr Handeln an den Börsen in negative Schlagzeilen brachte. Dabei

---

1 Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Fakten zur Versicherungswirtschaft“, S. 30 ff.

2 Der genaue Wortlaut der offiziellen deutschen Übersetzung ist online unter https://eur-lex.europa.eu/legal-content/DE/TXT/?uri=celex:32009L0138 einsehbar.

3 Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Säule I“.

4 „Das Hauptziel der [*auf dem Versicherungsaufsichtsgesetz fußenden, Anm. d. Autors*] Versicherungsaufsicht ist nach § 294 VAG der Schutz der Versicherungsnehmer und der Begünstigten von Versicherungsleistungen. … Besondere Bedeutung kommt dabei der Solvenzaufsicht zu.“

verbuchte der Versicherer im Jahr 2002 einen „Verlust von 50 Millionen Euro“, im ersten Quartal 2003 „betrug das Minus knapp … 57 Millionen [Euro].“[5] Regelungen wie Solvency II verhindern solche Totalausfälle der Versicherer, indem der Versicherungsbranche ein Drei-Säulen-Modell aus Kapitalvorgaben[6], Governance und Risikomanagament[7], sowie Berichtspflichten[8] unter Androhung und Vollzug hoher Strafen gesetzlich vorgeschrieben wird. Die deutsche Umsetzung dieser Richtlinie, das *Versicherungsaufsichtsgesetz*, verbindet seit dem Inkrafttreten des *Gesetzes zur Modernisierung der Finanzaufsicht über Versicherungen* am 01.01.2016 die EU-Regelungen mit den geltenden nationalen Bestimmungen.[9]

Solvency II steht nicht in Gegnerschaft zur Kapitalanlage. Es werden, neben den Regelungen über zu bildende Rückstellungen, auch Auflagen über die Anlage von „Vermögenswerten nach dem Grundsatz der unternehmerischen Vorsicht“ definiert. Die durch die verkauften Versicherungspolicen zugesicherten Leistungen müssen „mit entsprechenden geeigneten Vermögensanlagen bedeckt sein“, die das Unternehmen darüber hinaus mit „genügend freie[n] Finanzmittel[n]“ zusätzlich gegen „unerwartete Verluste“ absichert.[10]

Weitere, den Verbraucherschutz betreffende Regelungen schreibt u.a. das *Versicherungsvertragsgesetz* fest. Diese und weitere gesetzliche Vorgaben verursachen bei Versicherungsunternehmen hohe Kosten, denen regelmäßig kein erhöhter Ertrag gegenübersteht. Insbesondere in den Bereichen der IT-Sicherheit und des Datenschutzes werden Projekte aufgesetzt, die die Kassen der Versicherer belasten. Darüber hinaus legt sich die Versicherungsbranche regelmäßig auch freiwillige Verpflichtungen auf, die das Vertrauen in die Branche stärken sollen.[11]

---

5 Spiegel Online, „Erster Fall für den Protektor“.
6 Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Säule I“.
7 Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Säule II“.
8 Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Säule III“.
9 Vgl. Bundesanstalt für Finanzdienstleistungsaufsicht, „Solvency II“.
10 Vgl. Bundesanstalt für Finanzdienstleistungsaufsicht, „Versicherungsaufsicht“.
11 Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Fakten zur Versicherungswirtschaft“, S. 40 f.

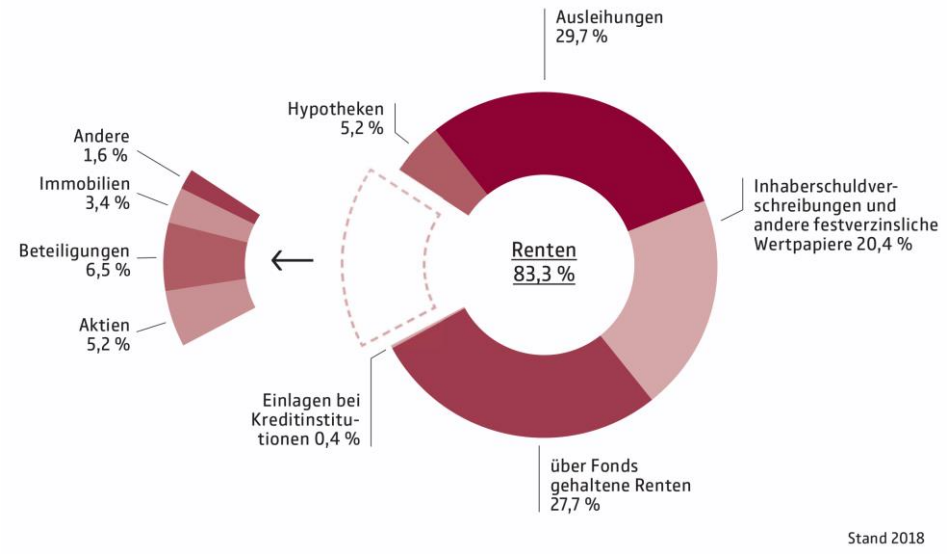
Die obigen Punkte zeigen, dass die Versicherungsbranche durch nationale und europarechtliche Gesetzgebung in ihrer Kapitalanlage stark reguliert ist. Doch ist der Versicherer zur ausreichenden Deckung der durch ihn versicherten Gefahren verpflichtet, wofür er auf die Gewinne aus der Kapitalanlage angewiesen ist. Insbesondere Regelungen wie Solvency II, die dem Versicherungsunternehmen die genannte Deckungspflicht verordnen, ziehen dabei enge Grenzen aus Vorschriften, innerhalb derer es keine Standardlösung für die Kapitalanlage gibt. Insbesondere Regelungen wie Solvency II verpflichten den Versicherer zur Kapitalanlage innerhalb der engen Grenzen aus Vorschriften, für die es am Markt noch keine Standardlösung gibt.

1.2 Ziel, Methodik und Aufbau dieser Arbeit

Die Versicherungsbranche ist als Kapitalanleger und in ihrer Rolle als „Anbieter von Risikoschutz … an einer nachhaltigen Entwicklung“ ihrer investierten Gelder interessiert.[12] Ein Anlageziel, dem die Versicherungsbranche dabei scheinbar kritisch gegenübersteht, wohl auch aufgrund der Geschichte und der ausgeprägten Regularien in diesem Gebiet, ist die Investition in kurzfristig abrufbare Finanzprodukte wie Aktien oder Beteiligungen: 2018 betrug der Anteil der angelegten Summe in diese Produkte lediglich 11,7% und lag damit weit hinter Anleihen und Rentenfonds (29,7% bzw. 27,7%). Abbildung 1 zeigt die Aufteilung der durch die Versicherungsbranche investierten Gelder.

[12] Vgl. Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Fakten zur Versicherungswirtschaft“, S. 36.

Abbildung 1: Kapitalanlagen der Versicherer



Quelle: Gesamtverband der Deutschen Versicherungswirtschaft e.V., „Fakten zur Versicherungswirtschaft“, S. 33.

Die rechtlichen Regelungen verbieten der Versicherungswirtschaft nicht, höhere Summen in die genannten Bereiche zu investieren. Dennoch scheut die Versicherungswirtschaft, wohl auch die Verfehlungen durch Investitionen u.a. in die *New Economy Blase* im Hinterkopf, weiterhin die großflächige Anlage in risikoreichere Produkte.

Ziel dieser Arbeit ist es, ein System für die vollautomatisierte kurz- bis mittelfristige Kapitalanlage zu entwickeln, das auf die Belange der Versicherungswirtschaft als Anleger zugeschnitten ist, und einen Implementierungsvorschlag für dieses zu liefern. Fokus und Grenzen dieser Methode werden an den jeweils relevanten Stellen aufgezeigt und es wird auf entsprechende weiterführende Literatur verwiesen. Zur Erreichung dieses Ziels werden folgende Fragen bearbeitet:

1. Was ist algorithmenbasiertes Trading und welche Chancen ergeben sich im Bereich neuer Technologien?
2. Kann die Versicherungsbranche algorithmenbasiertes Trading einsetzen?

3. Kann, unter Beachtung der strengen Auflagen, denen die Versicherungswirtschaft unterliegt, ein System zum vollautomatisierten Finanzhandel entwickelt werden, der von Versicherern eingesetzt werden kann?
4. Wie könnte solch ein System aussehen?
5. Wie könnte solch ein System umgesetzt werden?

Zur Beantwortung dieser Fragen wird folgende Methodik verfolgt: Zu Beginn wird über entsprechende Primärliteratur grundlegend benötigtes Wissen für das Verständnis dieser Arbeit offengelegt. Anschließend wird, über weitere Untersuchung der Literatur, der Stand der Technik, sowie die sich ergebenden Chancen und Risiken durch den technologischen Wandel beschrieben und in den Kontext dieser Arbeit in Bezug auf die Kapitalanlage eingeordnet. Letztlich werden, basierend auf diesen Grundlagen, unter Anwendung industrieprofessioneller Methoden und Werkzeuge, die Voraussetzungen für eine Software erarbeitet, welches von der Versicherungsbranche für die vollautomatisierte Kapitalanlage eingesetzt werden kann. Dieses wird schließlich entworfen und in einer branchennahen Technologie umgesetzt.

Zur Beantwortung der zuvor gestellten Fragen ergibt sich folgende Struktur dieser Arbeit. Kapitel 1 gibt einen Überblick über die Ausgangslage, in der sich die Versicherungsbranche befindet, sowie eine Begründung dafür, wieso die Versicherungsbranche die kurz- bis mittelfristige Anlage in Aktien zuletzt eher scheute. Kapitel 2 stellt die für das Verständnis der weiteren Ausführungen relevanten wissenschaftlichen Grundlagen dar. Kapitel 3 befasst sich mit dem Themenkomplex neuer Technologien innerhalb der Versicherungsbranche und mit der Frage, welche Chancen und Risiken sich daraus ergeben, insbesondere im Hinblick auf die Kapitalanlage.

Anschließend wird in Kapitel 4 ein auf die Versicherungswirtschaft angepasstes System zur Entwicklung eines eigenen Tradingsystems vorgestellt. Weiterhin wird ein Vorschlag für ein solches System in Java ausgearbeitet und implementiert, da es sich hierbei um eine in der Versicherungsbranche verbreitete Anwendungssprache handelt. Zudem werden die Ergebnisse dargestellt, die mit diesem System erzielt werden. In den Kapiteln 5 und 6 folgen eine kritische Betrachtung

sowie ein Fazit der Ergebnisse, in welchem auch auf die eingangs gestellten Fragen eingegangen wird. Kapitel 6 gibt außerdem einen Ausblick auf weitere Forschungsfragen, die nicht im Fokus dieser Arbeit liegen.

1.3 Zeitliche Abgrenzung

Diese Untersuchung entstand im Frühjahr 2020, zu Zeiten einer globalen Pandemie, verursacht durch die Verbreitung des Coronavirus SARS-COV-2. Die durch diesen Virus ausgelöste Viruserkrankung COVID-19, sowie die einhergehende Unsicherheit mit einer noch so neuen Krankheit mit nicht gänzlich erforschten Krankheitsverläufen, belasteten die ganze Welt. Die Folgen dieser Krise, wie Versammlungsverbote und Ausgangssperren, forderten ihren Tribut.

Diese Entwicklungen gingen auch an den Finanzmärkten nicht vorüber. Zahlreiche Märkte meldeten ab März 2020 branchenübergreifend Kursstürze, sodass wiederholt der Handel an den Börsen ausgesetzt[13] und bereits nach wenigen Tagen die erste Insolvenz infolge extremer Umsatzverluste angemeldet wurde.[14] Drastische Entwicklungen wie die im Frühjahr 2020, die die geltenden Gesetzmäßigkeiten außer Kraft setzen, können nur schwerlich parametrisiert werden. Daher liegen dieser Arbeit ausschließlich Finanzdaten bis einschließlich 31.12.2019 zugrunde.

[13] Vgl. Süddeutsche Zeitung, „Kurssturz an der Wall Street“.
[14] Vgl. Finanzen.net, „Vapiano zahlungsunfähig“.

2 Wissenschaftliche Grundlagen

Der Finanzmarkt unterliegt einer Fülle von Faktoren, die seinen Verlauf beeinflussen. Häufig ist es nicht möglich, Verläufe oder Änderungen rational zu erklären. Nichtsdestotrotz finden sich in den Lehrbüchern Kennzahlen und Konzepte, die versuchen, diesen undurchschaubaren Markt zu beschreiben. Die folgenden Kapitel geben einen kurzen Abriss über die grundlegenden Konzepte und Begrifflichkeiten, die der Leser für das Verständnis dieser Arbeit benötigt.

2.1 Weighted Moving Average – gleitender Mittelwert

Im Bereich der Finanzanlage wird mit großen Mengen aktueller Daten gerechnet. Dabei wird aktuellen Werten regelmäßig eine höhere Relevanz zugewiesen als älteren Werten. Zur Bildung von Mittelwerten, die auch weiterhin ein wichtiger Indikator zur Beurteilung von Skalenwerten sind, wird daher eine Methode benötigt, die jüngere Daten stärker gewichtet als historische.

Hierzu werden *exponentiell gewichtete gleitende Mittelwerte (EWMA)*[15] herangezogen. Dabei verlieren vergangene Werte stärker an Bedeutung, je weiter sie (auf einer Zeitachse) vom aktuellen Wert entfernt liegen. Basierend auf der Anzahl der Beobachtungen n, über die der EWMA berichten soll, wird der sog. *Decay* A gebildet, der die Gewichtung des jeweils aktuellen Wertes beschreibt. Durch die wiederholte Potenzierung des Decays (aufgrund der rekursiven Berechnung des EWMA) fließen vergangene Werte zu immer kleiner werdenden Teilen in den Mittelwert ein, bis sie an Relevanz verlieren.[16]

A ergibt sich gem. Formel 1 aus der Anzahl zu berücksichtigender Zeitpunkte n.

[15] Aus dem Englischen: Exponentially Weighted Moving Average.
[16] Vgl. CFA Institute, „EWMA (Exponentially Weighted Moving Average)“.

Formel 1: Decay alter EWMA-Werte A

$$A = \frac{2}{(n+1)}. \tag{1}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 283.

Für eine sinnvolle Beobachtungsspanne ($n > 1$) ergibt sich so ein Decay $0 < A < 1$. Die Zeitspanne $n = 1$ ist zu vermeiden, da mit ihr der Mittelwert über nur einen Zeitpunkt gebildet würde, welcher dem aktuellen Wert entspräche und so keinen Mehrwert besäße.[17]

Unter Benutzung des Decays A wird der EWMA E_t für den Zeitpunkt t gem. Formel 2 rekursiv aus dem Vorperioden-EWMA E_{t-1} und dem Kurswert des betrachteten Finanzprodukts k_t für den Zeitpunkt t bestimmt.

Formel 2: EWMA E_t zum Zeitpunkt t

$$E_t = (A * k_t) + [E_{t-1} * (1 - A)]. \tag{2}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 283.

Zur Berechnung des ersten Wertes E_0 wird $E_{t-1} = 0$ angenommen. Dieser, sowie alle Werte bis einschließlich dem $n - 1$ten besitzen nur eine verminderte Aussagekraft, da diese innerhalb ihres Betrachtungszeitraums den Vorperioden-EWMA von 0 zu einem nicht insignifikanten Anteil beinhalten.[18]

2.2 Positionen in einem Handel

Handelsgeschäfte auf dem Finanzmarkt können mitunter unübersichtlich werden. Durch die Vielzahl an Derivaten, die es im täglichen Geschäft zu erwerben gibt,

[17] Vgl. Carver, *Systematic Trading*, S. 283.
[18] Vgl. Carver, *Systematic Trading*, S. 282.

werden Begriffe zur Beschreibung der in einer Finanztransaktion gehaltenen Position benötigt. Für diese Zwecke haben sich die Begriffe der *Long*- und *Short*-Position etabliert.

Bei der Positionsbestimmung gilt folgende Faustformel: Der Inhaber einer Long-Position gewinnt, wenn der Basiswert steigt. Die Short-Position hingegen profitiert von fallenden Werten. Das bedeutet, dass ein Anleger, je nach Finanzprodukt, zugleich Short- und Long-Positionen innehaben kann. Kauft er bspw. eine Option, die auf den Wertverfall eines Indexes setzt, so nimmt er in Relation zu diesem Produkt eine Long-Position ein[19], während er zum zugrundeliegenden Basiswert eine Short-Position bezieht. Der Anleger setzt also auf ein Finanzprodukt, das nur an Wert gewinnt, wenn der Basiswert fällt.

Zur Wahrung der Übersichtlichkeit der eingenommenen Position wird hier im Verlauf die Position in Relation zum Basiswert angegeben. Wird bspw. über den *Deutschen Aktienindex (DAX 30,* im Folgenden: *DAX),* oder auf dem DAX basierende Produkte oder Indizes diskutiert, so wird immer die Position relativ zum DAX genannt.

Für den Fall, dass weder eine Long- noch eine Short-Position eingenommen wird, z.B. da weder mit dem Anstieg noch mit dem Fallen eines Kurses gerechnet wird, wird diese Position als *Hold* bezeichnet.

2.3 Psychologische Faktoren

Kognitive Verzerrung ist ein Sammelbegriff für psychologische Phänomene, bei denen Personen ihre eigenen Fähigkeiten falsch einschätzen. Die Verzerrung rührt dabei aus der meist niedrigeren objektiven als der höheren, subjektiv wahrgenommenen Fähigkeit, einen Sachverhalt korrekt einzuschätzen. Häufig werden zufällige Entwicklungen aus der Vergangenheit auf Ereignisse zurückgeführt, die in keinem Zusammenhang zum eingetretenen Ergebnis stehen. Solche Überschätzungen der Vorhersagbarkeit der Zukunft können z.B. ökonomischen oder

[19] Vgl. Kaufman, *A Short Course in Technical Trading*, S. 82.

statistischen Ursprungs sein.[20] Häufig spielt hier auch der sog. *Dunning-Kruger-Effekt* eine Rolle, welcher beschreibt, dass es inkompetenten Menschen häufig an der Fähigkeit fehlt, sich selbst als inkompetent einzuschätzen, wohingegen eine Steigerung der Fähigkeit auch mit der wachsenden Kompetenz zur Selbsteinschätzung einhergeht.[21]

Eine weitere für die Belange dieser Arbeit relevante Wahrnehmungsverzerrung ist das sog. *Prospect-Theorem*. Es diskutiert die unterschiedliche Wahrnehmung von Risiko im Bereich von Gewinnen und Verlusten. Dabei wird das Verlieren von einmal erreichten Gewinnen subjektiv höher bewertet als das weitere Verlieren bei bereits eingetretenen Verlusten.[22] Daraus folgt im Bereich der Geldanlage, dass stetig verlierende Vermögenswerte eher behalten werden als solche, die zuletzt stiegen. Bei Ersteren wird erwartet, dass ein künftiges Wachstum Gewinne einbringt, während bei Letzteren ein Preissturz befürchtet wird, vor dem sich geschützt werden muss. Zumeist tritt der gegenteilige Fall ein: Die Verlierer sinken weiter im Wert, während die Gewinner bessere Performance realisieren.[23]

2.4 Backtest

Regeln sind objektiv messbare Algorithmen, auf deren Basis Anlageentscheidungen getroffen werden.[24] Bevor mit solch einer Regel Handel betrieben werden kann, muss diese validiert werden. Hierfür bietet sich das etablierte Format des *Backtests* an. Beim Backtest wird eine objektiviert formulierte Regel oder Behauptung getestet, indem ihr Verhalten beim Einsatz in vergangenen Märkten geprüft und bewertet wird.[25]

---

20 Vgl. Taleb, *Fooled by Randomness*, S. 78 f.
21 Vgl. Kruger und Dunning, „Unskilled and Unaware of It“, S. 1123 ff.
22 Vgl. Pulham und Deeken, *Zur Rationalität von Anlageentscheidungen*, S. 21 ff.
23 Vgl. Shefrin und Statman, „The Disposition to Sell Winners Too Early and Ride Losers Too Long“, S. 779; Vgl. Odean, „Are Investors Reluctant to Realize Their Losses?“, S. 1775 ff.
24 Mehr zu Tradingregeln in Kapitel 4.2.5.
25 Vgl. Carver, *Systematic Trading*, S. 14.

Backtests bestehen aus zwei Phasen, dem *Fitting* und dem *Testing*. Beim Fitting wird eine Regel so modifiziert, dass sie in einem bestimmten Zeitraum die beste Performance erzielt (z.B., jedoch nicht zwingend, durch Messung des risikobereinigten Ertrags).[26] Welche Kennzahl ein Indikator für „gute" Performance ist, ist dabei durch die Anwenderin oder den Anwender individuell festzulegen. Die so erstellte Variation wird in der Testing-Phase auf einen definierten Zeitraum angewandt und ihre Performance wird beurteilt.[27]

Nach einem durchgeführten Test kann eine Regel nochmals angepasst und erneut überprüft werden. Bei der Evaluation des Testings sollte darauf geachtet werden, dass nicht ausschließlich auf den vergangenen Ertrag geachtet wird[28], sondern ebenfalls Faktoren wie z.B. die durch eine Regel entstehenden Transaktionskosten (durch eine hohe Anzahl an Positionswechseln) bzw. die damit in Verbindung stehenden Haltezeiten einer Position betrachtet werden, sodass *Over-Fitting* vermieden wird.[29]

Als Over-Fitting wird der Vorgang bezeichnet, wenn ein Algorithmus zu stark an vorhandene, vergangene (Markt-)Daten angepasst wird, sodass er bei der Anwendung auf diese Daten überproportional gute Ergebnisse liefert. Dies mag im Backtest funktionieren; in der späteren Implementierung der Regel auf den (nicht vorhersagbaren) Markt wird ein solcher Algorithmus jedoch regelmäßig unterdurchschnittliche Renditen erzielen.[30]

Ein weiterer Fallstrick beim automatisierten Test ist die Suggestion, dass im Test realisierte Handelsgeschwindigkeiten auch auf den realen Markt anwendbar sind. Regelmäßig können beim „echten" Handeln nicht diejenigen Preise erzielt wer-

---

26 Vgl. Carver, *Systematic Trading*, S. 51 ff.
27 Vgl. Kaufman, *A Short Course in Technical Trading*, S. 7.
28 Vgl. Carver, *Systematic Trading*, S. 58 f.
29 Vgl. Carver, *Systematic Trading*, S. 41 ff.
30 Vgl. Carver, *Systematic Trading*, S. 54.

den, die ein Handelstableau verspricht. Daher sind Tests von Regeln, die regelmäßig schnell Positionswechsel auslösen, mit einer entsprechenden Ungewissheit zu bewerten.[31]

2.4.1 In-Sample-Backtest

Beim Backtest wird zwischen zwei Ansätzen unterschieden. Beim sog. *In-Sample-Backtest* überschneiden sich die genutzten Zeiträume für Fitting und Testing (oder decken sich gar völlig).[32] Tabelle 1 zeigt einen In-Sample Backtest mit sich überschneidenden Zeitfenstern. Die eingefärbten Bereiche stellen dar, welches Zeitfenster für welche Phase verwendet wird. Im hiesigen Falle werden die Zeiträume t_0 bis t_8 für das Fitting und t_8 und t_9 für das Testing genutzt.

Tabelle 1: In-Sample Backtest mit sich überschneidenden Zeitfenstern

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
Fitting										
Testing										

Tabelle 2 zeigt einen In-Sample-Backtest, bei dem sich die verwendeten Zeiträume für Fitting und Testing vollständig decken.

Tabelle 2: In-Sample Backtest mit sich deckenden Zeitfenstern

	t_0	t_1	t_2
Fitting			
Testing			

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 55 (Figure 7).

Zum In-Sample-Backtest könnte bspw. gegriffen werden, wenn die vorhandenen Daten nur von geringem Umfang sind. Soll über eine Anlage entschieden werden, die aufgrund der nicht lange zurückliegenden erstmaligen Emission bislang

---

31 Vgl. Kaufman, *A Short Course in Technical Trading*, S. 49.
32 Vgl. Carver, *Systematic Trading*, S. 54 f.

wenige Daten aufweist, werden die vorhandenen historischen Daten effizienter genutzt. In Fällen solcher Datenknappheit ist jedoch zu beachten, dass der vorhandene Datenfundus nur wenige Facetten des Marktes abdeckt. Wird ein Wertpapier bspw. in einem Bullenmarkt erstmals emittiert und entwickeln sich die Kurse am Markt in den darauffolgenden Perioden positiv, ist die Kombination aus Finanzprodukt und Regel in keinem anderen Szenario getestet. Es kann demnach keine Aussage darüber getroffen werden, wie sie sich in einem Bären- oder Seitwärtsmarkt verhalten würde. Generell sind solche Produkte zu bevorzugen, zu denen ausreichende Marktdaten vorliegen.[33]

Die Überschneidung der beiden Testräume bringt ein entscheidendes Manko hervor: Im Fitting wird die Regel insoweit optimiert, dass sie optimale Ergebnisse erzielt. Wird das Fitting bspw. über einen Zeitraum von t_0 bis t_3 durchgeführt, würde ein Testing zum Zeitpunkt t_2 auf einer Regel aufbauen, von der durch das Fitting über diesen Zeitraum bekannt ist, dass diese dort die besten Ergebnisse erzielt. Der Backtest verzerrt somit die von einer Regel erwartete Performance, indem er sie zu stark bewertet. Hier besteht die Gefahr des Over-Fittings. Schließlich kann solch eine Optimierung im realen Einsatz über einen zukünftigen Zeitraum nicht vorgenommen werden, sodass die Regel hier schlechtere Ergebnisse erzielen wird.[34]

2.4.2 Out-Of-Sample Backtest

Bei Out-Of-Sample Backtests überschneiden sich die Zeiträume für Fitting und Testing nicht.[35] Bei einem vorhandenen Datenfundus über bspw. zehn Jahre wird dieser geteilt, z.B. die ersten fünf Jahre für das Fitting, die verbleibenden fünf für das Testing. Tabelle 3 zeigt beispielhaft, wie eine Aufteilung der vorhandenen Daten aussehen könnte.

---

33 Vgl. Kaufman, *A Short Course in Technical Trading*, S. 74 f.
34 Vgl. Carver, *Systematic Trading*, S. 54 f.
35 Vgl. Carver, *Systematic Trading*, S. 54.

Tabelle 3: Out-Of-Sample Backtest

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
Fitting										
Testing										

Quelle: In Anlehnung an Carver, S. 56 (Figure 8).

Zwar ist diese Variante des Backtests „ehrlicher" als In-Sample-Backtests, in dem Sinne, dass nicht für den Testing-Zeitraum optimiert wird. Der Out-Of-Sample Backtest birgt aber auch Nachteile. Zum einen wird bei einer 50/50-Aufteilung wie in Tabelle 3 die Hälfte der Daten nicht zur Verifizierung genutzt. Das ist insbesondere ein Nachteil bei kleinen Datenfundus. Zudem werden Marktänderungen, die in der zweiten Hälfte der Datenreihe auftreten, nicht für den Algorithmus berücksichtigt.[36]

Eine Abwandlung des Out-Of-Sample Backtests, die diese Problematik zu beheben versucht, ist die des sich *erweiternden Zeitfensters*. Bei beispielhaft angenommenen Daten, die über zehn Jahre kumuliert wurden, und bei einem Bedarf von Daten über mindestens ein Jahr zur Erreichung einer festgelegten Signifikanz würde der gesamte Backtesting-Prozess neunmal durchlaufen werden. Beim ersten Durchlauf findet das Fitting über die Daten des ersten Jahres, das Testing über diejenigen der verbleibenden neun statt. Bei jedem neuen Durchlauf wird der Testing-Zeitraum um das älteste Jahr gekürzt, wobei dieses in den Datenfundus für das Fitting aufgenommen wird.[37] Tabelle 4 zeigt beispielhaft drei Durchläufe beim Out-Of-Sample Backtest mit sich erweiterndem Zeitfenster.

[36] Vgl. Carver, *Systematic Trading*, S. 56 f.
[37] Vgl. Carver, *Systematic Trading*, S. 54 f.

Tabelle 4: Out-Of-Sample Backtest – sich erweiterndes Zeitfenster

Durchlauf		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
1	Fitting										
	Testing										
2	Fitting										
	Testing										
3	Fitting										
	Testing										

Quelle: In Anlehnung an Carver, Systematic Trading, S. 57 (Figure 9).

Das sich erweiternde Zeitfenster nutzt dabei die Vorteile des Out-Of-Sample Backtestings in Bezug auf die Überschneidungsfreiheit von Fitting und Testing-Zeiträumen, während die Datenverschwendung minimiert wird. Eine Problematik hierbei ist der sich immer weiter verlängernde Beobachtungszeitraum für das Fitting, wobei die kürzlich stattgefundenen Marktveränderungen immer weiter an Bedeutung verlieren, da ihr Anteil an der Gesamtmenge der Daten abnimmt.[38]

[38] Vgl. Carver, *Systematic Trading*, S. 56.

Tabelle 5: Problem des sich erweiternden Zeitfensters

Korrelationen	DAX/STOXX	DAX/S&P	STOXX/S&P
03.02.1997 – 31.12.2018	$r = 0{,}292217444$	$r = 0{,}959420445$	$r = 0{,}269623395$
2018	$r = 0{,}989206697$	$r = 0{,}405135941$	$r = 0{,}385056132$
2019	$r = 0{,}979092078$	$r = 0{,}953202826$	$r = 0{,}976981128$

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[39]

Letzteres kann nicht grundsätzlich angenommen werden. Die historischen Korrelationen der Kurswerte in Tabelle 5 des DAX und *EURO STOXX 50 Aktienindex (STOXX)* (zweite Spalte) weichen zwar deutlich von denjenigen im Jahr 2019 ab, wobei die Korrelation des Jahres 2018 sehr nah an derjenigen von 2019 rangiert. Die historischen Korrelationen von DAX und S&P 500[40] (dritte Spalte) hingegen sagen die Korrelation für 2019 deutlich besser vorher, als es diejenige aus 2018 kann. Bei der Korrelation von STOXX und S&P (vierte Spalte) zeigt sich ein drittes Bild: Hier konnten weder die historischen, noch die jüngeren Daten die Korrelation der beiden Basiswerte vorhersagen.

Ein weiteres Out-Of-Sample-Verfahren bedient sich einem *wandernden* (oder *rollenden*) *Zeitfenster*. Hierbei wird zu jedem Testing-Zeitpunkt t ein zeitlich direkt zuvor liegender Zeitraum bestimmter Länge für das Fitting genutzt. Bei einer Fitting-Zeitspanne von fünf Jahren sind für den Test-Zeitraum 2019 die Jahre 2014 bis 2018 relevant. Tabelle 6 zeigt drei Durchläufe des rollenden Zeitfensters.[41]

---

39 Datenquelle: ARIVA, „DAX 30 Historische Kurse“; ARIVA, „Euro Stoxx 50 Historische Kurse“; ARIVA, „S&P 500 Historische Kurse“; die aus diesen Quellen bezogenen Basisdaten wurden verrechnet, um die hier dargestellten Werte zu erhalten. Diese Vorgehensweise wird für den Rest dieser Arbeit beibehalten.

40 Standard & Poor’s 500 Aktienindex. Im Folgenden: S&P.

41 Vgl. Carver, *Systematic Trading*, S. 56 f.

Tabelle 6: Out-Of-Sample Backtest – rollendes Zeitfenster

Durch-lauf		t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
1	Fitting										
	Testing										
2	Fitting										
	Testing										
3	Fitting										
	Testing										

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 57 (Figure 10).

Die Herausforderung hierbei ist, einen Fitting-Zeitraum zu wählen, der so kurz wie möglich (zur Vermeidung „alter" Fitting-Daten), aber so lang wie nötig (zur Erreichung statistisch signifikanter Ergebnisse) ist, wobei hier auch die Erkenntnisse aus Tabelle 5 zu beachten sind. Weiterhin ist zu beachten, dass in einem Datenfundus vorhandene Aufzeichnungen über eine extreme Marktsituation, wie z.B. einen Börsenkrach, nur in einem Teil der Durchläufe zum Fitting und unter Umständen überhaupt nicht zum Testing verwendet werden. Beinhaltet bspw. der Zeitpunkt t_1 in Tabelle 6 einen Börsenkrach, würden Durchläufe 1 und 2 diesen zum Fitting verwenden, während Durchlauf 3 ohne diesen gefittet wird. Weiterhin würde bei einem Backtest nach Tabelle 6, also bei einem Testing-Beginn in t_5 die Periode des Krachs nie getestet werden. Eine Datengrundlage sollte möglichst viele unterschiedliche Marktsituationen berücksichtigen.[42]

[42] Vgl. Kaufman, *A Short Course in Technical Trading*, S. 75.

3 Neue Technologien

Die deutsche Versicherungswirtschaft gilt im Allgemeinen als „eher zurückhaltend, wenn es um Technologien und Innovationen geht".[43] Die hierdurch entstehende bzw. wachsende technische Schuld dämmt die Innovation weiter ein. Junge Unternehmen im Bereich der Versicherungswirtschaft, sog. *InsurTechs*, können hingegen meist unbelastet von solch technischer Schuld „auf der grünen Wiese" starten und so deutlich schneller Produkte entwickeln und auf den Markt bringen. Die von diesen Akteuren eingesetzten technischen Neuerungen können dabei auch von der tradierten Versicherungsbranche genutzt werden. Dieser Abschnitt soll zeigen, inwieweit sich diese Technologien auf die Versicherungsbranche auswirken, als Chance oder als Risiko.

3.1 Chancen durch technologischen Wandel

Emerging Risks (ER)[44] stellen für alle Branchen eine große Herausforderung dar. Sie weisen einen „hohen Grad an Unsicherheit bezüglich der zukünftigen Entwicklung" auf, da „keine historischen Beobachtungsdaten" zur Beurteilung herangezogen werden können.[45] Zu ER „liegen häufig keine Erkenntnisse oder Daten vor, um konkrete Auswirkungen [oder] … Eintrittswahrscheinlichkeit[en]" bestimmen zu können.[46]

Insbesondere Themenkomplexe wie der der *künstlichen Intelligenz (KI),* sowie der mit der KI einhergehende Wandel hin zu autonomen Maschinen, stehen im Fokus der Betrachtungen. Die Auswirkungen, die solche Systeme insbesondere auf Haftungsfragen haben, sind unerforscht und daher unklar. „Traditionelle Deckungen müssen angepasst werden, um sowohl Privatpersonen als auch Unter-

---

43 Kröger, „Digitalisierung in der Versicherungsbranche".

44 Unter dem Begriff ER wird regelmäßig eine große Menge „neuartige[r] zukunftsbezogene[r] Risiken" verstanden, deren Auswirkungen zumeist nicht quantifizierbar sind. Vgl. Schweizerischer Versicherungsverband SVV, „Emerging Risks", S. 4.

45 Ausschuss Enterprise Risk Management, „Emerging Risks 2020", S. 5.

46 Ausschuss Enterprise Risk Management, „Emerging Risks 2020", S. 6.

nehmen entsprechend zu schützen … [Es] werden sich … neue Haftpflichtversicherungsmodelle durchsetzen, die Hersteller und Softwareanbieter stärker in die Pflicht nehmen“ urteilt etwa der Ausschuss Enterprise Risk Management des Deutschen Aktuarvereinigung e.V.[47]

Nicht alle Charakteristika, die diese Technologien mitbringen, sind mitunter als schlecht für die Versicherungsbranche einzustufen. Häufig entstehen aus ihnen Ideen, die die bestehenden Verhältnisse positiv beeinflussen und die Situation in anwendenden Unternehmen verbessern können.

Der Themenkomplex KI hat vielerlei Implikationen, insbesondere auch für die Kapitalanlage. So könnten selbständig lernende KI-Modelle durch fortwährende Analyse der zur Verfügung stehenden Daten das Anlageprofil des Versicherers kontinuierlich anpassen. Nimmt eine solche Intelligenz bspw. Unruhen in bestimmten Wirtschaftssektoren wahr, könnte sie die Anlage aus diesen Bereichen zurückziehen und in Unternehmen investieren, die aus dem entstehenden Stimmungswandel profitieren. All das geschähe mit einer Geschwindigkeit, die für menschliche Anlageverwalterinnen und -verwalter nicht nachvollziehbar oder reproduzierbar wäre, sodass solch ein Positionswechsel bereits vollzogen sein könnte, bevor der Verwalter von den Unruhen erfährt.

Diesen Wandel hat auch der Anbieter der größten deutschen Aktienmärkte, die Gruppe Deutsche Börse, erkannt. Das Unternehmen, das u.a. die Börse Frankfurt und den Handelsplatz Xetra betreibt, beschreibt seinen eigenen Wandel im Zuge der „Digitalisierungstrends“, deren Entwicklung sich voraussichtlich „beschleunigen“ werde, sowie die „Herausforderung[en] für etablierte Anbieter“, die sich aus diesem Wandel ergeben.[48] So ist die Gruppe bisweilen auch „erster Finanzdienstleister, der die regulatorische Freigabe für die Nutzung der *Cloud*-Technologie [SAP S/4Hana] erhalten hat“ und setzt so „Maßstäbe für die Finanzbranche“.[49] Durch die Nutzung dieser technologischen Neuerungen profitiert die Gruppe von verschlankten Prozessen und kann ihren Fokus weiter auf ihr Kerngeschäft richten.[50]

---

47 Ausschuss Enterprise Risk Management, „Emerging Risks 2020“, S. 37.
48 Vgl. Gruppe Deutsche Börse, „Geschäftsbericht 2018“, S. 142.
49 Gruppe Deutsche Börse, „Deutsche Börse stellt IT auf Cloud-Nutzung um“.
50 Vgl. Gruppe Deutsche Börse, „Deutsche Börse stellt IT auf Cloud-Nutzung um“.

Ein weiterer Themenkomplex, der die Kapitalanlage in den kommenden Jahren und Jahrzehnten beschäftigen wird, ist der Bereich *Big Data*. Big Data ist dabei als Sammelbegriff für Operationen an und mit großen Datenmengen zu verstehen – je nach Anwendungsbereich bis zu mehreren Petabytes an Daten. Bereits 2016 wuchs der Speicherverbrauch der Videoplattform YouTube um ca. 1 Petabyte pro Tag, wobei Brewer u.a., bei anhaltendem Wachstum, von einer Verzehnfachung der Datenmenge alle fünf Jahre ausgehen.[51] Davon ausgehend kann angenommen werden, dass die Datenmenge auch in anderen Bereichen weiter ansteigt (wenn auch nicht in den Maßstäben von YouTube, Konzernmutter Alphabet, oder -schwester Google).

Diese immer weiter wachsende Menge an öffentlich zur Verfügung stehenden Daten macht sich mitunter auch die Versicherungsbranche zu Nutze: Die Münchner Rück nutzt in einem Projekt mit ihrem IT-Service-Provider EMPOLIS öffentlich zugängliche Quellen wie internationale Nachrichten und Lokalmedien zur Beurteilung von Schadenereignissen und zur schnelleren Abwicklung der daraus resultierenden Versicherungsansprüche. Durch „automatisierte[s] Monitoring … [t]ausende[r] globale[r] Nachrichtenquellen“, wobei der „Inhalt jeder einzelnen Nachricht“ ausgewertet wird, kann eine KI über Klassifizierung der Ereignisse, sowie der damit in Verbindung stehenden Eckdaten wie „Schadensort, Schadenszeitpunkt etc.“, alle für ggf. auftretende Schadensansprüche relevanten Informationen extrahieren und „allen Anwendungen zur Verfügung“ stellen.[52]

Diese schiere Masse an zur Verfügung stehenden Daten, sowie die immer leistungsstärkeren Computer, die zur Analyse dieser Daten genutzt werden, könnten sich auch für den Einsatz in der Kapitalanlage bestens eignen. Durch die Möglichkeit, Finanzdaten in quasi Echtzeit[53] direkt von den Börsen zu erhalten, kann auch die Anlagesituation ebenfalls in Echtzeit (oder zumindest in Rechtzeit, also

---

51 Vgl. Brewer u. a., „Disks for Data Centers“, S. 3.

52 Vgl. Bitkom e.V. und Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, „Künstliche Intelligenz“, S. 176 f.

53 Einige Börsen verzögern die Übermittlung dieser Daten absichtlich. Die New Yorker Börse *Investors Exchange* (*IEX*) verlängert die Übertragungsdauer jeglicher Marktdaten um 350ms, indem sie die ausgehenden Signale durch ein 38 Meilen langes Glasfaserkabel leitet. Vgl. Lewis, *Flash Boys*, S. 176 ff.

zu einem noch immer nutzbaren Zeitpunkt, der aber nicht der Echtzeit entspricht) neu bewertet und ein Portfolio entsprechend umgeschichtet werden.

3.2 Robotik im Anlagevermögen

In diesem Überfluss an Daten fällt es ohne technische Hilfsmittel schwer, den Überblick zu bewahren. Insbesondere der Bereich der Kapitalanlage, in dem sich die Marktgegebenheiten innerhalb von Sekundenbruchteilen ändern können (man denke nur an die Ad-hoc-Mitteilungspflicht börsennotierter Unternehmen gem. Art. 17 Abs. 1 MMVO), ist nur schwer ganzheitlich von einzelnen Personen zu überblicken. Hinzu kommt die Fehleranfälligkeit durch subjektive Entscheidungen Einzelner, wie die Erkenntnisse aus Kapitel 2.3 zeigen.

Um diese menschlichen Fehlbarkeiten zu umgehen, wird eine Methode benötigt, die es ermöglicht, basierend auf objektiven Beobachtungspunkten, Entscheidungen zu treffen und diese wiederholgenau und vor allen Dingen revisionssicher zu reproduzieren. Hier kommt *Algorithmic Trading* ins Spiel.

Algorithmic Trading (auch *Robotic Trading* oder *Robo-Trading*) beschreibt das computergesteuerte Auslösen von Trading-Ordern, wobei der Eingriff oder die Überwachung durch einen Menschen nicht benötigt wird. Gesteuert werden diese Systeme durch deterministische Benchmarks oder Regeln, sodass sich (mit Einblick in Benchmark und Datengrundlage) jederzeit nachvollziehen lässt, wieso eine Order ausgeführt wurde.[54] 2008 wurden bereits 43% der gehandelten Wertpapiere durch Algorithmen ge- und verkauft.[55]

Der Vorteil solcher Systeme liegt auf der Hand: Kognitive Verzerrungen, wie sie bei menschlichen Anlegern vorkommen, können durch die Nutzung algorithmengesteuerter Anlage reduziert werden. Sie verfolgen stets und ausschließlich ihre implementierten Benchmarks und berechnen das Risiko, sowohl auf der Gewinn-, als auch auf der Verlustseite, unvoreingenommen. Das ist auch der

---

54 Vgl. Gsell, „Technological Innovations in Securites Trading“, S. 52 f.

55 Vgl. Deutsche Börse, „Preliminary Results Q4 and FY 2008“, S. 10; zitiert nach Gsell, „Technological Innovations in Securites Trading“, S. 53.

Grund dafür, wieso algorithmenbasierte Systeme regelmäßig besser abschneiden als es menschliche Vergleichsprobandinnen und -probanden können.[56] Zu beachten ist dabei natürlich auch, dass diese algorithmischen Systeme von Menschen implementiert werden, deren Verhaltensbias sich in die entwickelten Systeme überträgt.[57]

Unter der Voraussetzung, menschlichen Bias aus einem solchem System zu entfernen, z.B. durch ein diversifiziertes Panel an Qualitätssicherungspersonal in der Entwicklung, bringt algorithmisches Traden quantifizierbare Zugewinne für das Handeln von Finanzprodukten. Durch das deterministische Handeln solcher Systeme ist eine zeit- und kostenintensive Überwachung von Finanzdaten durch einen Menschen nicht notwendig.[58] Weiterhin können, bei einem direkten Anschluss an eine Wertpapierbörse, durch die Nutzung algorithmischer Handelssysteme Kosten gespart werden. Die Deutsche Börse bspw. rabattiert die Nutzung ihrer Systeme bei höheren Transaktionsfrequenzen gegenüber niederfrequenten Ausführungen.[59]

---

56 Vgl. Aigner und D'Acunto, „Robo-Advisor-Studie", 4:18 - 4:50.

57 Vgl. Fußnote 12 in D'Acunto, Prabhala, und Rossi, „The Promises and Pitfalls of Robo-Advising", S. 18 f.

58 Vgl. D'Acunto, Prabhala, und Rossi, „The Promises and Pitfalls of Robo-Advising", S. 7.

59 Vgl. Deutsche Börse, „Preisverzeichnis für die Nutzung der Börsen-EDV der Frankfurter Wertpapierbörse und der EDV XONTRO", S. 7.

4 Tradingsystem für die Versicherungsbranche

Die Versicherungsbranche benötigt eine Lösung, mit der sie langanhaltendes Trading betreiben kann, um sowohl ihr Asset-Management in die eigene Hand zu nehmen als auch zum Zweck der Erhöhung der liquiden Mittel. Im folgenden Abschnitt wird beschrieben, wie bestehende Ideen über den Finanzmarkt objektivierbar erfasst und programmatisch umgesetzt werden können.

Die folgende Umsetzung erfolgt in Anlehnung an die von Carver vorgestellte Methode zum Aufbau eines Tradingsystems.[60] Die Ideen Carvers werden übernommen und auf die Versicherungswirtschaft angewandt. Dabei fließen insbesondere die besonderen Gegebenheiten und Restriktionen der Versicherungswirtschaft in die Anwendung ein. Im Anschluss wird ein entsprechendes System entworfen und implementiert.

4.1 Vorbemerkungen

Der Handel mit Finanzinstrumenten unterliegt einer Vielzahl an Größen, die die Anlegerinnen und Anleger in ihre Kaufs- und Verkaufsentscheidungen einbeziehen müssen. Um die Aussagekraft der vorliegenden Untersuchung zu gewährleisten, werden daher einige Aspekte aus dem Fokus genommen, die für die Entwicklung des Systems nicht betrachtet werden. Diese werden im Folgenden erläutert.

Die folgende Entwicklung basiert auf den Regeln des börslichen Aktienhandels. In Betracht gezogen werden dabei, sowohl bei der folgenden theoretischen Betrachtung als auch bei der Validierung der Konzepte, nur solche Basiswerte, die Aktien sind, oder ihren Wert direkt aus ihnen ableiten (wie z.B. Marktindizes). Darüber hinaus umfasst dies die Entscheidung, Finanzinstrumente nur börslich zu handeln und auf den freien OTC-Handel[61] zu verzichten. Der Vorteil hierbei liegt in der deutlich besseren Datenbasis, die aus dem börslichen Handel resultiert. Diese Daten werden zur Prüfung der fachlichen Hypothesen herangezogen, die im kommenden Kapitel definiert werden. Der außerbörsliche OTC-Handel

---

60 Vgl. Carver, *Systematic Trading*.

61 Direkthandel zwischen zwei Handelspartnern. Aus dem Englischen: *Over The Counter* (*OTC*).

hingegen basiert auf Übereinkünften direkt zwischen Käufer und Verkäufer, sodass diese Geschäfte zumeist nicht über einen Intermediär abgewickelt und verfolgbar gespeichert werden.

Das hier gezeigte System strebt unter anderem die Vergleichbarkeit der Performance verschiedener Basiswerte und der auf ihnen entwickelten Regeln an. Es handelt sich hierbei explizit um keine Empfehlung für bestimmte Produkte oder Derivate. Daher werden, falls Berechnungen auf Preis- oder Wertentwicklungen stattfinden, ausschließlich die zugrundeliegenden Basiswerte zur Kalkulation herangezogen. Sind für einen für eine Berechnung benötigten Indikator keine Daten verfügbar, wird dieser entsprechend seiner Definition im Rahmen dieser Untersuchung eigenständig kalkuliert. Dies ist an den entsprechenden Stellen der kommenden Kapitel kenntlich gemacht.

International relevante Währungen wie der US-Dollar unterliegen starken Schwankungen und Beeinflussungen durch real- und fiskalpolitische Entscheidungen. Die Macht, die Notenbanken und politische Lager auf eine Devise ausüben, macht es dem Anleger schwer, eine Kursvorhersage zu treffen. Entscheidungen, wie die Festsetzung eines Mindestkurses (wie es die Schweizerische Nationalbank mit dem Schweizer Franken tat) oder der Währungskauf durch eine Notenbank sind über Backtests nicht nachvollziehbar.[62] Die Umstände, denen eine Devise unterliegt, sind im Falle einer Investitionsentscheidung genauestens zu prüfen. Im Rahmen dieser Untersuchung wird kein Handel mit Devisen angestrebt.

Eine weitere Einschränkung betrifft die Zusammenstellung des zu handelnden Portfolios. Regelmäßig wird Diversifikation eines Portfolios zu Zwecken der Risikostreuung und der damit einhergehenden höheren Renditeerwartung angestrebt.[63] Zu Zwecken dieser Arbeit wird sich auf eine kleine Menge an Basiswerten beschränkt, um die entwickelten Prozesse zu veranschaulichen. Bei einer Anwendung des Systems an realen Märkten sollte vom Anleger eine Analyse der zu handelnden Basiswerte, sowie eine Evaluation der Gewichtung der einzelnen

62 Vgl. Carver, *Systematic Trading*, S. 103.
63 Vgl. Markowitz, „Portfolio Selection“, S. 89.

Werte anhand angemessener Indikatoren erfolgen. Diese Indikatoren werden im Laufe dieses Kapitels erläutert.

Es wird vorausgesetzt, dass nur nach Börsenschluss verlässlicher und vorhersehbarer Handel betrieben werden kann. Das hier zu entwickelnde System basiert daher darauf, dass mit Schlusskursen gehandelt werden kann. Untertägige Effekte, wie die verzögerte Übermittlung von Kurswerten und die Tätigkeiten von Hochfrequenzhändlern[64], werden so aus dem System eliminiert. Für den realen Anwendungsfall ist zu beachten, dass Marktschlusspreise zur nächsten Handelsperiode häufig nicht realisierbar sind.[65]

Darüber hinaus wird erwartet, dass die Liquidität aller Finanzprodukte zum gewünschten Handelszeitpunkt gegeben ist, sodass jede Order wie eingestellt auch ausgeführt werden kann. Freilich ist dies keine pauschal korrekt treffbare Aussage, sodass das Thema Liquidität zu gegebener Zeit erneut thematisiert und die Größenordnung, in der diese Liquidität angenommen wird, entsprechend relativiert wird.

Der Handel mit Finanzprodukten erzeugt Kosten. Die Höhe dieser Kosten hängt davon ab, wie die Anlegerin oder der Anleger den Handel tatsächlich realisiert. Wird sich mit dem hier entwickelten System an eine Brokerin oder einen Broker angeschlossen, sind zumeist Transaktionskosten durch die Nutzung der Plattform zu erwarten. Weiterhin sind die Kosten solcher Broker nicht vergleichbar, sodass sich hierüber keine pauschale Aussage treffen lässt. Anstatt sich in dieser Arbeit auf die Kostenstruktur eines einzelnen Brokers zu beschränken, werden Finanztransaktionskosten aus den hiesigen Betrachtungen ausgeschlossen. Nichtsdestotrotz sollten bei privatem oder gewerbsmäßigem Einsatz dieses Systems die mit dem Handel verbundenen Verbindlichkeiten in den Anlageentscheidungen berücksichtigt werden.

---

64 Hochfrequenzhändler nutzen durch technische Finesse und die physische Nähe zu den Börsen das vergleichsweise langsamere Handeln „regulärer“ Anleger aus. Sie erhöhen damit die Preise für den regulären Handel bei jedem Trade um einen geringen Betrag, wodurch sie täglich millionenfach kleinste Margen erzielen. Einen Insider-Überblick über diese moralisch bedenkliche Form des Handels bietet Lewis, *Flash Boys*. Tiefergehende Analyse ist dem Anleger unbedingt zu empfehlen, wenn regelmäßig und in großen Stückzahlen untertägig gehandelt werden soll.

65 Vgl. Kaufman, *A Short Course in Technical Trading*, S. 70.

4.2 Anforderungen an das System

Carver beschreibt ein System zur systematischen Kapitalanlage, in welchem der Anleger unter Anwendung objektivierbarer Behauptungen über den Finanzmarkt Algorithmen entwickelt, nach deren Maßgabe Finanzinstrumente gehandelt werden. Darüber hinaus stellt er vor, welche Parameter der potenzielle Anwender eines solchen Systems vor dessen Einsatz prüfen muss.

Carvers Ausführungen sind eher generell gehalten und entsprechen seinem Anspruch, Tradingsysteme für Jedermann bereitstellen zu können. Sie decken ein breites Feld an Anwendungsfällen und potenziellen Nutzern ab. In den folgenden Abschnitten wird dieses System analysiert und auf die Belange der Versicherungswirtschaft angepasst. Dabei werden bestimmte Komponenten oder Optionen aus dem System ausgeschlossen, da diese bspw. nicht mit für die Versicherungswirtschaft geltenden Regularien vereinbar sind, oder es werden Anpassungen vorgenommen, die Carver in seiner Beschreibung nicht aufführt, die für eine Anwendung in der Versicherungsbranche jedoch unumgänglich sind.

4.2.1 Herangehensweise

Bevor mit dem Aufbau eines Handelssystems begonnen werden kann, muss die grundsätzliche Herangehensweise der Anlegerinnen und Anleger bestimmt werden. Diese hängt z.B. davon ab, aus welcher Fachrichtung sie ursprünglich kommen, wie technikaffin sie sind, oder welche finanzmathematischen Kenntnisse sie besitzen. In dieser Untersuchung wird zwischen zwei grundlegenden Ansätzen, dem *Data First* und dem *Ideas First*, unterschieden.

Beim Data First-Ansatz werden vorhandene Finanzdaten explorativ untersucht, um in ihnen Muster zu erkennen, welche sich wiederum in Tradingregeln übersetzen lassen. Typischerweise werden dabei Kennzahlen berechnet und verglichen, oder Auswertungen über Kurshistorien angefertigt, aus denen der Anleger Verhaltensweisen des Finanzmarktes abzulesen versucht. Positiv für die Wahl dieses Ansatzes steht die wachsende Rechenleistung, die ein Analysieren großer Datenmengen wie der des gesamten Finanzmarktes technisch immer leichter machen. Darüber hinaus kann durchdachte Datenanalyse Strategien hervorbringen, die nur durch das exzessive Analysieren der Marktdaten zum Vorschein

kommen, während diese bei einem ideenbasierten Vorgehen ggf. nicht auffallen.[66]

Beim alternativen Ansatz, dem Ideas First, werden, losgelöst von konkreten Zahlen und Performanceanalysen, Hypothesen über das Verhalten von Märkten aufgestellt, unter deren Ausnutzung ein zukünftiger Verlauf vorhergesagt werden soll. Die getroffenen Aussagen haben dabei stets qualitativen Charakter, welcher erst im Laufe der Regelentwicklung quantifiziert wird. Für den Ideas First-Ansatz spricht die „grüne Wiese"-Ausgangslage: Es werden unvoreingenommen von tatsächlichen Performancewerten Vermutungen aufgestellt, die erst in einem späteren Schritt in quantifizierbare Regeln übersetzt werden, anhand derer die Validität der Behauptungen dann geprüft wird. Darüber hinaus sind über den Ideas First-Ansatz entwickelte Regeln prosaisch erklärbar – schließlich sind sie so auch entwickelt worden. So ergeben diese Ideen intuitiv Sinn und können nicht nur anhand der Betrachtung von Performance-Daten validiert werden.[67]

In dieser Arbeit wird der Ideas First-Ansatz verfolgt. Dafür werden aus zwei Ideen über den Finanzmarkt Regeln abgeleitet, aus denen Forecasts[68] über die künftige Entwicklung des Portfolios gebildet werden. Hierbei wird sich denjenigen Werkzeugen bedient, die Carver mit seinem Framework mitbringt, und mit ihrer Hilfe wird ein System entworfen, das vollautomatisiert betrieben werden kann.

Eine weitere grundlegende Entscheidung umfasst das Backtest-Verfahren. Entsprechend der Erwägungen aus Kapitel 2.4 wird in dieser Arbeit ein Out-Of-Sample Backtest vorgeschlagen. Im Rahmen dieser Arbeit wird der Fitting-Zeitraum lediglich für das Bestimmen der Gewichtungen der Basiswerte genutzt. Das Implementieren weiterer Fitting-Parameter sprengte den Rahmen dieser Arbeit und liegt daher nicht im Fokus dieser.

4.2.2 Ideen über den Finanzmarkt

Der Finanzmarkt wird von vielen Faktoren gesteuert, die keine einzelne Entität überblicken oder gar vorhersagen kann. Jede Regel, mit der versucht wird, das

---

66 Vgl. Carver, *Systematic Trading*, S. 26 f.
67 Vgl. Carver, *Systematic Trading*, S. 26 f.
68 Die Begriffe „Forecast" und „Vorhersage" werden in dieser Arbeit synonym verwendet.

Verhalten „des Marktes“ abzubilden, stellt daher nur einen kleinen Teil eines komplexen Systems dar. Eine Beobachtung, die ein bestimmtes Verhalten der Märkte zu erklären versucht, ist die *Niedrig-Volatilitäts-Anomalie*. Der Finanzmarktforscher Robert Haugen untersuchte im Jahr 1969 „die Wertentwicklung von US-Aktien in Abhängigkeit von ihrem Risiko“ und stellte dabei fest, dass risikoärmere Papiere im Schnitt höhere Renditen generieren – ein Ergebnis, das nicht zum Verständnis von Finanzmärkten passt, dass hohe Gewinne regelmäßig mit hohen Volatilitäten einhergehen und hohe Renditen auch nur dann erwirtschaftet werden können, wenn Anlegerin und Anleger entsprechende Risiken in Kauf nehmen.[69]

Weitere Forschung, aufbauend auf Haugens ursprünglicher Hypothese, bestätigt diese Behauptung. Anlegerinnen und Anleger, die unter Beachtung der Volatilität eines Anlageproduktes handeln, können eine um ca. 2,9% höhere jährliche Rendite erwirtschaften – oder knapp 77% in 20 Jahren – als solche, die auf eine „Kaufen und Halten“-Strategie setzen.[70] Auch Haugen selbst erforschte seine Thesen weiter und konnte sie in unterschiedlichen globalen Märkten beobachten.[71]

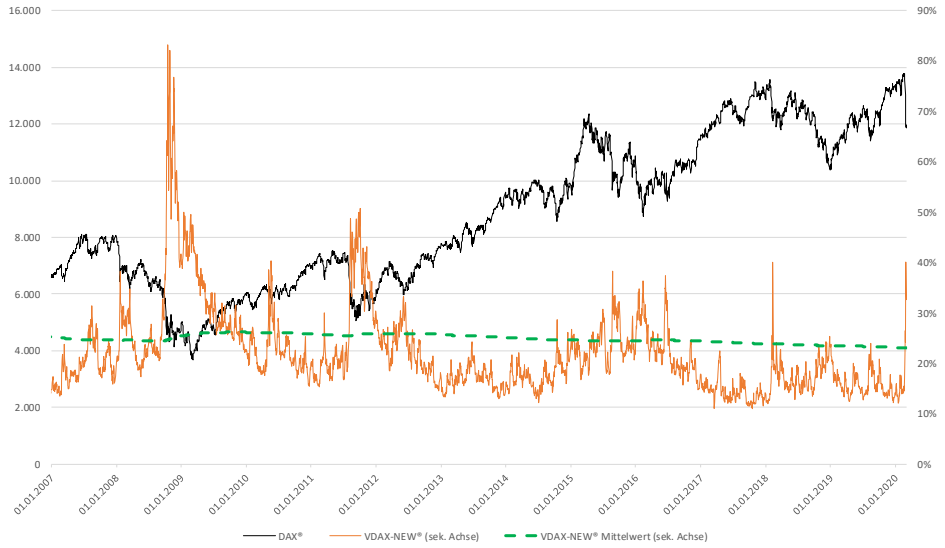
---

69 Vgl. Haugen und Heins, „Risk and the Rate of Return on Financial Assets“, S. 779 ff.; Vgl. Mittnik, „Geldanlage für Angsthasen“.

70 Vgl. Moreira und Muir, „Should Long-Term Investors Time Volatility?“, S. 518.

71 Vgl. Baker und Haugen, „Low Risk Stocks Outperform within All Observable Markets of the World“, S. 16 ff.

Abbildung 2: Entwicklung DAX und Volatilitätsindex VDAX-NEW®



Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[72]

Eine beispielhafte Betrachtung des wichtigsten Deutschen Aktienindexes, des DAX, sowie seinem Volatilitätsabbild VDAX-NEW®, verschafft einen Überblick über diese Behauptungen. Abbildung 2 zeigt beide Indizes. In Zeiten, in denen der VDAX-NEW® über seinem Allzeit-Mittelwert verläuft (sprich: in Zeiten hoher Kursschwankungen des Basiswerts) büßt der DAX regelmäßig stark an Wert ein. Geringe Kursschwankungen scheint der DAX hingegen regelmäßig mit steigender Punktzahl zu „belohnen". Die Volatilität ist demnach ein wertvoller Faktor in der Generierung von Rendite.

„Offizielle" Volatilitätsindizes werden nur für einen kleinen Teil handelbarer Basiswerte berechnet. Regelmäßig bieten Börsen Volatilitätsindizes nur für die jeweils national bedeutendsten Aktienindizes an, z.B. in Deutschland der VDAX-NEW® für den DAX, in Europa der VSTOXX® für den STOXX, und in den *Verei-*

72 Datenquelle: ARIVA, „DAX 30 Historische Kurse"; onvista, „VDAX-NEW Index".

nigten Staaten von Amerika (USA) der VIX® für den S&P. Wie können Anlegerinnen und Anleger also auch solche Basiswerte handeln, für die durch die Börsen kein öffentlich zugänglicher Volatilitätsindex berechnet und ausgewiesen wird?

Volatilitätsindizes wie der VDAX-NEW® indizieren die Schwankung in den Optionspreisen auf den DAX.[73] Einfacher gesagt: Schwanken DAX-Optionen stark im Preis, steigt der VDAX-NEW®; sind die Preise stabil, fällt der VDAX-NEW®.

Durch die Kalkulation auf Basis der Schwankungen von Optionspreisen wird die Schwankung des Basiswerts abgebildet. Der VDAX-NEW® umfasst die Schwankungen der Preise auf Optionen mit Restlaufzeiten von 1 bis 12 Monaten, oder schließt, falls solche Produkte nicht verfügbar sind, per Extrapolation auf Werte dieser Laufzeiten.[74] Steht für einen zu handelnden Basiswert kein Volatilitätsindex zur Verfügung, kann dieser annähernd über die Standardabweichung der Renditen eines Finanzproduktes bestimmt werden. Entsprechend der Definition des VDAX-NEW®, der die Volatilität der Option mit Restlaufzeiten von 30 Tagen darstellt, wird für das Aufstellen eines eigenen Volatilitätsindex auch die auf das Jahr gerechnete Standardabweichung der Rendite der vergangenen 30 Tage betrachtet. Da dieser selbst errechnete Index direkt auf dem Basiswert fußt und nicht etwa auf Preisen expliziter Finanzprodukte, müssen Effekte, die sich „typischerweise kurz vor Laufzeitende in starken Volatilitätsschwankungen“[75] bemerkbar machen, nicht weiter betrachtet werden. Liegt die errechnete 30-Tages-Volatilität über dem Langzeitdurchschnitt[76], befindet sich der Basiswert in einer

---

73 Vgl. Deutsche Börse AG, „Leitfaden zu den Volatilitätsindizes der Deutschen Börse“, S. 19 ff.

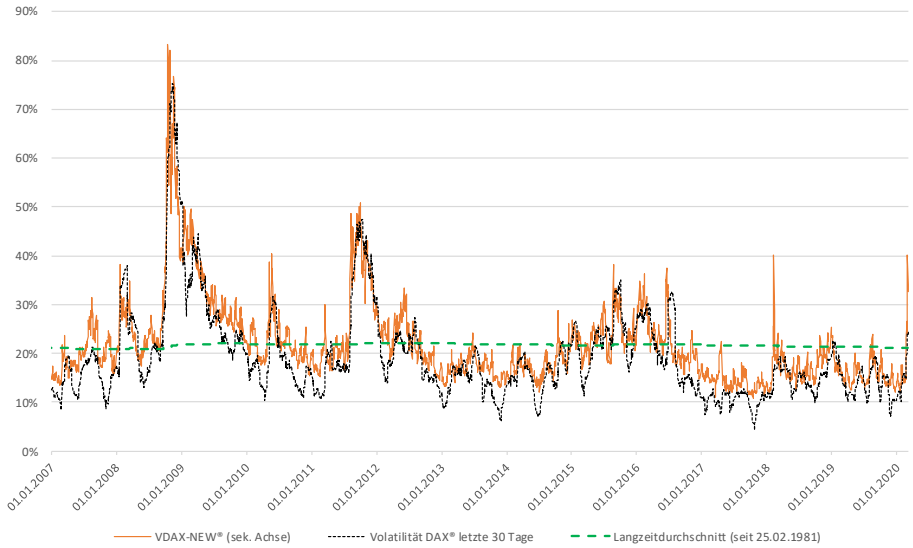
74 Vgl. Deutsche Börse AG, „Leitfaden zu den Volatilitätsindizes der Deutschen Börse“, S. 5.

75 Deutsche Börse AG, „Leitfaden zu den Volatilitätsindizes der Deutschen Börse“, S. 5.

76 Zur Transparenz und Wiederholbarkeit der Berechnung wird nicht mit zum Betrachtungszeitpunkt künftigen Werten gearbeitet (Out-Of-Sample). Der zum Vergleich genutzte Langzeitdurchschnitt bildet sich über die bestimmten historischen Volatilitätsindexwerte bis hin zum jeweiligen Berechnungstag. Liegen dem Anleger bspw. Aufzeichnungen von DAX®-Daten seit dem 01.03.1981 vor, und berechnet er die Volatilität für den Börsentag 18.05.1990, so bildet der Langzeitdurchschnitt lediglich Volatilitätswerte zwischen diesen beiden Daten ab.

Hochvolatilitätsphase. Liegt sie darunter, spricht man von einer Niedrigvolatilitätsphase. Letztere geht, im Einklang mit der oben genannten Forschung, erwartungsgemäß mit steigenden Kursen einher.

Abbildung 3: VDAX-NEW® und selbsterrechnete 30-Tages-Volatilität des DAX



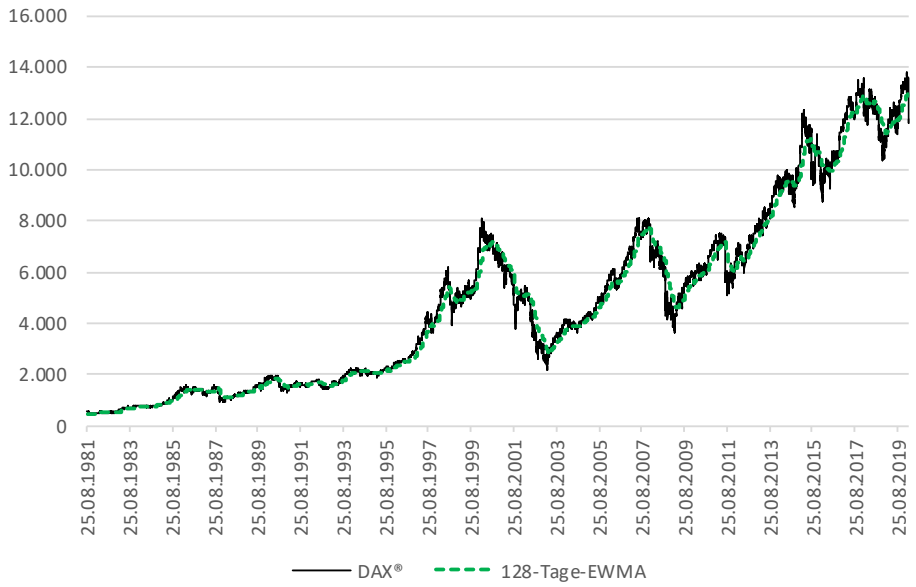
Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[77]

Abbildung 3 zeigt den historischen Verlauf sowohl des VDAX-NEW®, als auch denjenigen des ermittelten Volatilitätsindexes, sowie den Langzeitdurchschnitt des Letzteren. Die beiden Volatilitätsindizes weisen mit einem Korrelationskoeffizienten von $r = 0{,}909492848$ eine hohe Korrelation auf, sodass eine Substitutionsfähigkeit angenommen wird. Korrelationen wie diese lassen sich auch bei der Beobachtung anderer Indizes, über die ein offizieller Volatilitätsindex gebildet wird, feststellen. Anhang 0zeigt die Volatilitäten dieses und weiterer ausgewählter Indizes.

[77] Datenquelle: ARIVA, „DAX 30 Historische Kurse“; onvista, „VDAX-NEW Index“.

Eine weitere Beobachtung, aus der sich eine regelmäßige Verhaltensweise ableiten lässt, ist die Trendvermutung. Abgeleitet aus der gefühlten Abhängigkeit einer Kursveränderung von seiner vorherigen Entwicklung (die Medienberichterstattung spricht hier von „Talfahrt“, „Höhenflug“, und „Stagnation“) lässt sich unter Betrachtung eines EWMA eine solche Vermutung als plausibel darstellen. Innerhalb des in Abbildung 4 abgebildeten Zeitraums ändert der 128-Tage-EWMA lediglich an 477 Börsentagen seine Richtung, wohingegen er an 9.396 Tagen seine jeweilige Vortagestendenz beibehält. Dies entspricht einer Wahrscheinlichkeit von $> 95\%$, dass der EWMA auf einen Tag mit einer Steigung erneut wächst. Bei der Betrachtung des 32-Tage-EWMA liegt diese Wahrscheinlichkeit immerhin noch bei $> 90\%$ (8.959:914).

Abbildung 4: DAX und 128-Tages-EWMA des DAX

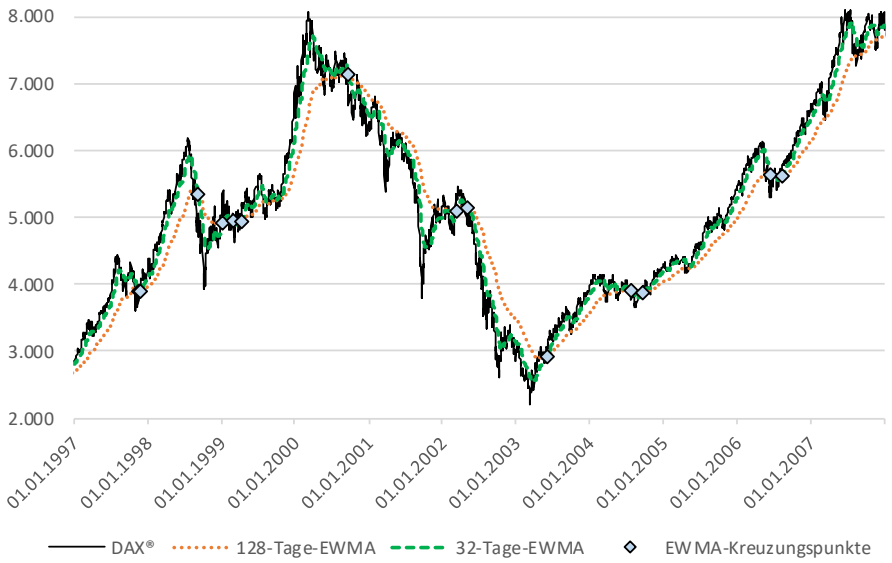


Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[78]

Weiterhin lässt sich anhand der EWMAs beobachten, dass der zugrundeliegende Basiswert zwischen zwei EWMA-Kreuzungspunkten *(EWMAC)*[79] tendenziell steigt, wenn der kurzfristigere EWMA in dieser Phase über seinem längerfristigen Pendant liegt (wenn also im ersten Kreuzungspunkt der kurzfristige den langfristigen EWMA übersteigt); tritt der gegenteilige Fall ein, und der langfristige Schnitt liegt über dem kurzfristigen, fällt der Basiswert zumeist ab.[80] Abbildung 5 zeigt die EWMA-Kreuzungspunkte.

78 Datenquelle: ARIVA, „DAX 30 Historische Kurse“.
79 Aus dem Englischen: *EWMA Crossover* (*EWMAC*).
80 Vgl. Carver, *Systematic Trading*, S. 118 f.

Abbildung 5: DAX, 128-Tage-EWMA, 32-Tage-EWMA und EWMA-Kreuzungspunkte



Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[81]

Aufgrund der hohen Anzahl an Einflussfaktoren lassen sich allein mit statistischen Mitteln freilich keine 100%ig sicheren Prognosen über die Zukunft treffen. Dennoch lässt sich die künftige Entwicklung eines Finanzproduktes über die zwei zuvor ausgeführten Beobachtungen jeweils mit hoher Wahrscheinlichkeit bestimmen. Die folgenden Abschnitte dieser Arbeit bauen daher auf diese beiden Beobachtungen auf. Bisher qualitativ anmutende Aussagen werden im Laufe dieses Abschnitts insoweit quantifiziert, sodass auf ihrer Grundlage ein Tradingsystem aufgebaut werden kann.

[81] Datenquelle: ARIVA, „DAX 30 Historische Kurse“.

4.2.3 Wahl der Anlagestrategie

Die Wahl der grundsätzlichen Anlagestrategie bestimmt wesentlich, wie das Tradingsystem aufgebaut wird. Unterschiedliche Anlageverhalten und -profile weisen spezielle Charakterzüge auf, die im weiteren Aufbau beachtet werden müssen. Unterschieden werden kann dabei zwischen drei grundsätzlichen Arten von Anlegerinnen und Anlegern.

„Kaufen und Halten"-Anleger. Dieser Typus kann nur im weiteren Sinne als Trader bezeichnet werden. Dieser Anleger ist der Ansicht, dass über die künftige Entwicklung von Finanzprodukten keine Vorhersagen getroffen werden können. Er ist zudem skeptisch gegenüber Behauptungen, durch aktives Portfoliomanagement könnten zusätzliche Gewinne erzielt werden. Dementsprechend verwendet der „Kaufen und Halten"-Anleger keine Hebelprodukte, sondern vertraut auf den Basiswert eines Finanzinstrumentes. Dabei setzt er auf günstige Finanzprodukte, wie z.B. passiv verwaltete *Exchange Traded Funds* (börsengehandelte Fonds). Er verwendet die ihm zum Zwecke der Anlage zur Verfügung stehenden liquiden Mittel vollständig zum Kauf dieser Finanzprodukte, welche er für unbestimmte Zeit in sein Depot legt. Der „Kaufen und Halten"-Anleger unterwirft sich keinem Kurslimit und verkauft demnach auch keine Wertpapiere, um den Erlös in andere Finanzinstrumente anzulegen.[82]

Teilautomatisierter Anleger. Der teilautomatisierte Anleger entspricht in etwa einem institutionellen Vermögensverwalter. Seiner Meinung nach unterliegt der Finanzmarkt keinen Mustern, die durch einfache Regeln zu beschreiben sind. Vielmehr vertraut er auf seine eigenen Einschätzungen, die er auf Basis seines eigenen Wissensstandes formuliert. Der teilautomatisierte Anleger handelt jede Art von Finanzinstrumenten, erwägt also bspw. auch Hebelprodukte oder Leerverkäufe. Entsprechende Einschränkungen lassen sich in der Anwendung dennoch treffen.[83]

Vollautomatisierter Anleger. Der vollautomatisierte Anleger verlässt sich nicht auf sein eigenes (angenommenes) Wissen, sondern setzt ausschließlich auf syste-

---

82 Vgl. Carver, *Systematic Trading*, S. ix.
83 Vgl. Carver, *Systematic Trading*, S. ix f.

matische Vorgehensweisen. Hierfür schöpft er das volle Potenzial von Tradingregeln aus, um Vorhersagen für die gehandelten Finanzinstrumente zu treffen. Aufgrund des Vertrauens in die systematisch bestimmten Vorhersagen setzt auch er auf Hebelprodukte und andere Derivate. Je nach Risikoaversion handelt er mehr oder weniger aktiv, wodurch sich auch potenziell versteckte Renditen erwirtschaften lassen. Er verlässt sich zum Aufstellen seiner Forecasts u. U. auf Backtesting, was jedoch nicht zwingend erforderlich ist.[84]

Für die Versicherungsbranche lässt sich basierend auf diesen Angaben kein eindeutiges Anlegerbild festlegen. Basierend auf der Größe des Unternehmens könnte eine Unterscheidung getroffen werden. Diese wird verfeinert, wenn unternehmensspezifische Kennzahlen, wie z.B. das zum Trading zur Verfügung stehende Kapital, die Summe der den Versicherungskunden garantierten Kapitalzahlungen, oder aber auch die Volatilität der Prämienzahlungen einbezogen werden. Die drei Anleger lassen sich jedoch anhand des Implementierungsaufwandes und des Horizonts, in denen sie ihre Anlage betrachten, grob unterscheiden und jeweils für eine Entscheidungssituation empfehlen. Genauere Analysen des jeweiligen Bedarfes sind darüber hinaus dennoch stets anzufertigen.

Die Strategie des *„Kaufen und Halten"-Anlegers* ist prinzipiell diejenige Lösung, die unter dem geringsten Aufwand implementiert werden kann. Der „Kaufen und Halten"-Anleger ist ein langfristig orientierter Investor, der nicht täglich die Wertentwicklung seines Portfolios[85] prüft, und möchte nur geringen Aufwand für die Anlage seines Kapitals aufbringen. Beispiele für diese Art von Anleger sind der durchschnittliche Amateuranleger, oder aber auch Pensionskassen[86] bzw. Direktversicherungen[87]. Die Beiträge sind regelmäßig und berechenbar, der Auszahlungshorizont für jeden Begünstigten ist bekannt. Über diese Parameter lässt

---

84 Vgl. Carver, *Systematic Trading*, S. x.

85 Institutionelle Anleger sollten dies dennoch tun!

86 Vgl. Carver, *Systematic Trading*, S. ix.

87 Pensionskassen und Direktversicherungen unterliegen „seit 2005 … der gleichen steuerlichen Förderung, sodass eine Abgrenzung nur noch über den Träger der Versorgung gegeben ist". (Schumacher, Sobau, und Hänsler, *Entgeltumwandlung*, S. 112). Da eine Unterscheidung der Träger für diese Arbeit irrelevant ist, werden hier beide Begriffe synonym verwendet.

sich ein regelmäßiger Sparbeitrag berechnen, der in weitere Wertpapiere angelegt werden kann. Pensionskassen arbeiten mit dem Ziel, ihre eigenen Kosten zu decken und garantieren lediglich die Auszahlung der eingezahlten Beiträge plus Erträge, abzüglich ggf. anfallender Risikoausgleiche[88]. Bei Renteneintritt eines Begünstigten nimmt die Pensionskasse einen zuvor vereinbarten Betrag aus der Wertanlage heraus und zahlt sie entsprechend der Vereinbarungen aus, oder aber sie zahlt eine Rente, wobei sie den entsprechenden Betrag dann regelmäßig zur Fälligkeit aus dem Depot entnimmt.

Das Investmentprofil der *teilautomatisierten Anlage* ist aufgrund der Willkür durch den Anleger nur bedingt revisionssicher, weshalb sie sich auch nur unter bestimmten Umständen für Versicherer eignet. Teilautomatisierte Vorhersagen könnten z.B. genutzt werden, um durch überschüssige flüssige Mittel Renditen über der risikofreien Anlage zu erwirtschaften. Diese Anlagen sollten nur einen kleinen Teil des Portfolios ausmachen und mit Bedacht eingesetzt werden. Für den Betrieb einer teilautomatisierten Anlage werden regelmäßig eine oder mehrere dedizierte Stellen benötigt, die nicht nur über ausgeprägtes finanzmathematisches Wissen verfügen, sondern auch interessiert und geschult in Realpolitik, Politikhistorie, Wirtschaftshistorie u.Ä. sind. Die teilautomatisierte Anlage dürfte somit nur für große Versicherungsunternehmen realisierbar sein.

Eine Form der Vermögensverwaltung, die prinzipiell zu jedem Versicherer passt[89], ist die *vollautomatisierte Anlage*. Das Festhalten an vordefinierten Regeln macht sie aufgrund der Wiederholbarkeit der Vorhersagen zum designierten Anlageprofil für Versicherer. Die Gründe für jede Handlung sind nachvollziehbar und belegbar; das Trading geschieht revisionssicher und wiederholgenau. Das tatsächliche Abgeben von Ordern kann über ein IT-System abgewickelt werden. Hierbei ist zwar mit initialen Kosten für die Entwicklung und Einrichtung eines

---

88 Gem. § 1 Abs. 2 Nr. 2 BetrAVG.

89 Insofern dieser über ausreichend Ressourcen für die initiale Implementierung verfügt. Darüber hinaus kann ein vollautomatisches System auch daran scheitern, wenn keine ausreichenden finanzmathematischen Fähigkeiten im Mitarbeiterstamm vorhanden sind. Dies kann z.B. auf kleine Fin-Tech-Unternehmen zutreffen, die selbst keine Versicherungsprodukte entwerfen, sondern Produkte von Rückversicherern verkaufen und den Kundenservice realisieren. Zwar sind in Fin-Techs regelmäßig per Definition technikaffine Mitarbeiter angesiedelt; Eine Implementierung eines Tradingsystems ohne ausreichend finanzmathematische Kenntnisse ist jedoch äußerst riskant.

solchen Systems zu rechnen, die Kosten für Betrieb und Wartung eines solchen einmal eingeführten Systems fallen im Vergleich dazu jedoch gering aus. Zum Betrieb benötigt der Versicherer eine eigene Stelle, um neue Regeln zu entwerfen bzw. die Performance des bestehenden Systems zu überwachen.

Für die Zwecke der vorliegenden Arbeit wird das Profil des vollautomatisierten Anlegers weiterverfolgt. Dieser eignet sich den obigen Ausführungen entsprechend für eine breite Anzahl (auch institutioneller) Anwender, wird in der vorliegenden Untersuchung jedoch auf die Versicherungsbranche spezialisiert. Plant ein Anleger eine andere Strategie zu implementieren, wird ein Studieren der Primärliteratur empfohlen.

4.2.4 Festlegen von Gewichtungen

In einem vollautomatisierten Tradingsystem interagieren eine große Anzahl an Variablen, die auf die zu treffenden Anlageentscheidungen Einfluss nehmen. Nicht alle Kennzahlen sollten dabei den gleichen Anteil an der letztlich bezogenen Position haben. Daher schlägt Carver eine tabellarische Umsetzung vor, wie von Korrelationen bestimmter Datenreihen zueinander auf deren Gewichtung geschlossen wird. Die nach dieser Methodik beurteilten Kenngrößen werden dabei rekursiv in Gruppen zu drei zusammengesetzt. Hierfür schlägt er eine Gewichtung der Werte gem. Tabelle 7 vor.

Tabelle 7: Gewichtung innerhalb Gruppen bei der händischen Zusammenstellung eines Portfolios

1	Ein Asset	100% für dieses Asset
2	Zwei Assets	50% für jedes der beiden Assets
3	Beliebige Anzahl Assets identischer Korrelationen zueinander	Gleiche Gewichtung für alle Assets
4	Vier oder mehr Assets unterschiedlicher Korrelationen zueinander	Neuverteilen der Gruppen, bis sie einer der anderen Reihen entsprechen

	Drei Assets mit den Korrelationen			**Gewichtung je Asset**		
	AB	**AC**	**BC**	**A**	**B**	**C**
5	0,0	0,5	0,0	30%	40%	30%
6	0,0	0,9	0,0	27%	46%	27%
7	0,5	0,0	0,5	37%	26%	37%
8	0,0	0,5	0,9	45%	45%	10%
9	0,9	0,0	0,9	39%	22%	39%
10	0,5	0,9	0,5	29%	42%	29%
11	0,9	0,5	0,9	42%	16%	42%

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 79 (Table 8).

Liegen die ermittelten Korrelationen zwischen zweien der in Tabelle 7 aufgeführten Werte, so werden diese jeweils auf den nächsten angegebenen Wert gerundet. Negative Werte werden, zur Vermeidung stark verzerrter Gewichtungen, auf $0{,}0$ gerundet. Eine ermittelte Korrelation von bspw. $0{,}2$ wird demnach auf $0{,}0$, während $0{,}3$ auf $0{,}5$ gerundet wird.[90] Dies hat den Nachteil, dass höchst unterschiedliche Korrelationen auf einen gemeinsamen Wert gerundet werden, sodass die tatsächlichen Korrelationen verfälscht und die entsprechenden Gewichtungen nicht äquivalent vergeben werden. So werden alle Werte $\geq 0{,}25$ und

[90] Vgl. Carver, *Systematic Trading*, S. 79.

$< 0{,}7$ auf $0{,}5$ gerundet – das entspricht knapp der Hälfte des positiven Korrelationsbereichs. Eine solche tabellarische Aufstellung ist zwar ein guter Anhaltspunkt, jedoch für die Nutzung in einem vollautomatisierten System ungeeignet. Daher wird eine Berechnung gem. Formel 3 vorgeschlagen, über die sich an die Werte der obigen Tabelle annähernde Gewichtungen errechnen lassen.

Formel 3: Gewichtung w_x einer Datenreihe x

$$w_x = \frac{r_{x,invers}}{\sum r_{n,invers}}. \tag{3}$$

Die Gewichtung einer Datenreihe w_x entspricht dabei dem Quotienten der inversen Durchschnittskorrelation $r_{x,invers}$ einer Datenreihe x und der Summe der inversen Durchschnittskorrelationen aller relevanten Datenreihen. $r_{x,invers}$ berechnet sich gem. Formel 4 durch die Subtraktion der Durchschnittlichen Korrelation $\overline{r_x}$ einer Datenreihe x von 1, welche sich wiederum gem. Formel 5 aus allen Korrelationen $r_{x,n}$ mit einer Beteiligung der Datenreihe x ergibt.

Formel 4: Inverse Durchschnittskorrelation $r_{x,invers}$ einer Datenreihe x

$$r_{x,invers} = 1 - \overline{r_x}. \tag{4}$$

Formel 5: Durchschnittliche Korrelation $\overline{r_x}$ einer Datenreihe x

$$\overline{r_x} = \frac{1}{n}\sum r_{x,n}. \tag{5}$$

Das beispielhaft ausgewählte Portfolio besteht aus drei Basiswerten. Gem. der Ausführungen in Kapitel 4.2.1 werden die Daten der Jahre 2014–2018 dazu genutzt, die Korrelationen der Basiswerte zu bestimmen. Tabelle 8 zeigt diese Korrelationen.

Tabelle 8: Korrelationen der Basiswerte im Fitting-Zeitraum

Basiswertkombination	Korrelation
DAX/STOXX	$r = 0{,}820458371$
DAX/S&P	$r = 0{,}844285275$
STOXX/S&P	$r = 0{,}497945338$

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[91]

Gem. der obigen Formeln lassen sich die Gewichtungen der Basiswerte bestimmen. Die zweite Spalte in Tabelle 9 zeigt die Gewichtungen. Diese stimmen mit den aus Tabelle 7 abgelesenen Gewichtungen (dritte Spalte) annähernd überein.

Tabelle 9: Gewichtungen der Basiswerte im Beispielportfolio

Basiswert	Gewichtung (berechnet)	Gewichtung (Tabelle 7)
DAX	20,02%	16%
STOXX	40,7%	42%
S&P	39,28%	42%

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[92]

Zum Vergleich mit den Gewichtungen gem. Tabelle 7 lassen sich für die dort definierten Referenzwerte ebenfalls Gewichtungen bestimmen. Tabelle 10 stellt diese errechneten Gewichtungen den von Carver in Tabelle 7 definierten gegenüber.

91 Datenquelle: ARIVA, „DAX 30 Historische Kurse“; ARIVA, „Euro Stoxx 50 Historische Kurse“; ARIVA, „S&P 500 Historische Kurse“.

92 Datenquelle: ARIVA, „DAX 30 Historische Kurse“; ARIVA, „Euro Stoxx 50 Historische Kurse“; ARIVA, „S&P 500 Historische Kurse“.

Tabelle 10: Errechnete Gewichtungen (Gewichtungen gem. Tabelle 7 in Klammern)

Drei Assets mit den Korrelationen			Gewichtung je Asset					
AB	**AC**	**BC**	**A**		**B**		**C**	
0,0	0,5	0,0	30%	(30%)	40%	(40%)	30%	(30%)
0,0	0,9	0,0	26,19%	(27%)	47,62%	(46%)	26,19%	(27%)
0,5	0,0	0,5	37,5%	(37%)	25%	(26%)	37,5%	(37%)
0,0	0,5	0,9	46,875%	(45%)	34,375%	(45%)	18,75%	(10%)
0,9	0,0	0,9	45,83%	(39%)	8,34%	(22%)	45,83%	(39%)
0,5	0,9	0,5	27,27%	(29%)	45,45%	(42%)	27,27%	(29%)
0,9	0,5	0,9	42,86%	(42%)	14,29%	(16%)	42,86%	(42%)

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 79 (Table 8).[93]

Für die Zwecke der vorliegenden Untersuchung entsprechen die errechneten Werte denjenigen aus Tabelle 7 zur Genüge. Zudem überwiegen die zuvor geschilderten Vorteile der vollständigen Berechnung solcher Werte die Nachteile der auftretenden Ungenauigkeiten. An der Aufrundung negativer Korrelationen auf den Wert von 0,0 wird festgehalten, da sich die Verzerrung durch solch extreme Werte auch durch eine stufenlose Berechnung nicht verhindern lässt.

Das in diesem Kapitel gezeigte Gewichtungsverfahren lässt sich nicht nur auf die reinen Basiswerte anwenden, sondern wird auch an anderen Stellen innerhalb des Systems Anwendung finden. Dabei wird sich kontinuierlich auf die Anwendung der Formel 3 bezogen.

[93] Datenquelle: ARIVA, „DAX 30 Historische Kurse“; ARIVA, „Euro Stoxx 50 Historische Kurse“; ARIVA, „S&P 500 Historische Kurse“.

4.2.5 Volatilitätsindizes

Zur Kalkulation der Vorhersage auf Basis der in Kapitel 4.2.2 getätigten Beobachtung basierend auf der Niedrig-Volatilitäts-Anomalie wird ein Volatilitätsindikator benötigt. Für bedeutende Indizes, wie z.B. den DAX in Deutschland oder den S&P in den USA, berechnen die Börsen solche Volatilitätsindizes anhand der Preisschwankungen von Optionen[94], die den Index als Basiswert nutzen: Für den DAX ist das seit 2003 der VDAX-NEW®, die Volatilität des S&P wird im VIX abgebildet. Daten dieser Indizes sind regelmäßig auch bei Anbietern abrufbar, die Kurse der Basiswerte liefern können.

Wird zu einem Basiswert kein Volatilitätsindex ausgewiesen, muss die Volatilität v_n äquivalent bestimmt werden. Diese ergibt sich gem. Formel 6 durch Multiplikation der Standardabweichung der Renditen σ_n mit $\sqrt{252}$, um diese auf das Jahr umzurechnen.[95] Der Berechnung wird zugrunde gelegt, dass ein Börsenjahr 252 Handelstage umfasst.

Formel 6: Volatilität v_n

$$v_n = \sigma_n * \sqrt{252}. \tag{6}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 21.

Die Stichproben-Standardabweichung der Renditen wird gem. Formel 7 berechnet. k_t entspricht dem Kurs des Basiswerts zum Zeitpunkt t, $\bar{k}$ dem Mittel der Kurswerte der Stichprobe, n der Größe der Stichprobe.

---

94 Vgl. Deutsche Börse AG, „Leitfaden zu den Volatilitätsindizes der Deutschen Börse“, S. 19 ff.

95 Vgl. Carver, *Systematic Trading*, S. 21.

Formel 7: Empirische Standardabweichung einer Stichprobe σ_n

$$\sigma_n = \sqrt{\frac{\sum_{t=i}^{n}(k_t - \bar{k})^2}{n-1}}. \quad (7)$$

Quelle: In Anlehnung an Göhler und Ralle, *Formelsammlung höhere Mathematik*, S. 89.

Die in Prozentpunkten angegebene Volatilität v_n der letzten n Beobachtungsperioden gem. Formel 7 weist im Vergleich zu den von den Börsen berechneten Volatilitätsindizes regelmäßig hohe Korrelationen auf ($r \geq 0{,}85$ sind realistisch). Abhängig davon, welche Handelsfrequenz gewählt wird, können Unterschiede zu den Volatilitätsindizes auftreten: Die von den Börsen gelieferten Indizes werden regelmäßig in kurzen Abständen[96] berechnet, während der weniger frequente Trader ggf. nur täglich am Tagesende Volatilitäten berechnet und darauf basierend handelt.

4.2.6 Tradingregeln

Gemäß den Erkenntnissen der Prospect-Theorie handeln Individuen unter Betrachtung des einzugehenden Risikos, dem ihre Entscheidung unterliegt, nicht linear. So schätzt ein Individuum den potenziellen Verlust anders ein als einen Gewinn mit identischer objektiver Wahrscheinlichkeit. Solche Verzerrungen können, bei der Adaption auf realfinanzielle Entscheidungen, große Risiken für das eingesetzte Kapital bedeuten. Um diesem Verhalten zu begegnen, werden klar definierte und messbare Regeln benötigt. So können situationsunabhängige, reelle und rationale Ziele und Handlungsanweisungen definiert werden, die zum Treffen von Kaufs- und Verkaufsentscheidungen herangezogen werden.

Diese Regeln müssen objektiv und rational ausführbar sein und dürfen dabei keinen Entscheidungsspielraum zulassen.[97] Dabei sollten diese Regeln nicht nur

---

96 Der VDAX-NEW® und seine Subindizes werden minütlich aktualisiert. Vgl. Deutsche Börse AG, „Leitfaden zu den Volatilitätsindizes der Deutschen Börse“, S. 4.

97 Vgl. Kaufman, *A Short Course in Technical Trading*, S. 8.

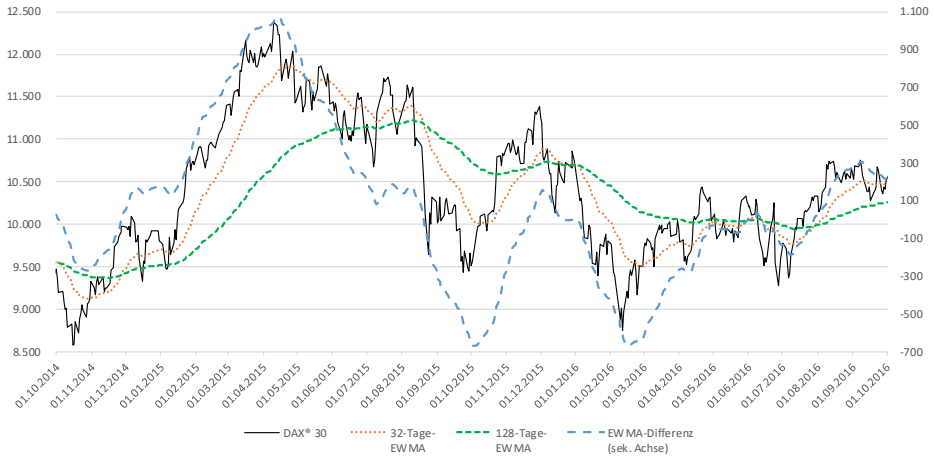
binäre Vorhersagen (also: Long oder Short), sondern Indikatoren für die tatsächlich einzunehmende Positionsgröße ermitteln.

Ein Beispiel für eine positionsgrößenindizierende Regel ist der EWMAC. Über die Differenz zweier EWMA zueinander, wenn diese unterschiedliche Horizonte abbilden, lassen sich sowohl Positionswechsel als auch Größen der jeweils aktuell gehaltenen Positionen bestimmen.[98] Durch die Nutzung des EWMAC lässt sich die in Kapitel 4.2.2 formulierte Idee über das Trendverhalten von Finanzinstrumenten abbilden.

Abbildung 6 verdeutlicht die Funktionsweise des EWMAC. Nach einem langanhaltenden Aufwärtstrend fällt der Basiswert stark ab, wodurch der kurzfristiger betrachtende EWMA unter den langfristigeren fällt (Kreuzung am 21.08.2015). Durch das weitere Abfallen des Basiswertes erhöht sich auch die Differenz zwischen beiden Werten, bis sie am 02.10.2015 bei einem Wert von $-669{,}36$ ein lokales Minimum erreicht. Das anschließende lokale Maximum des Basiswertes sorgt für eine kurzzeitig positive Differenz, bis der folgende Kurssturz diese wieder ins Negative umkehrt. Erst am 29.07.2016 überschreitet der 32-Tage-EWMA sein längerfristiges Pendant wieder und sorgt so für eine positive Differenz zwischen den beiden.

98 Vgl. Carver, *Systematic Trading*, S. 117 f.

Abbildung 6: DAX, 32-Tage-EWMA-, 128-Tage-EWMA, EWMA-Differenz



Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[99]

Gemäß des in Kapitel 4.2.2 beobachteten Verhaltens der EWMA zueinander muss für eine positive Vorhersage der kurzfristige EWMA $E_{k,t}$ größer sein als sein langfristiges Pendant $E_{l,t}$. D.h. wir erhalten nur dann eine Differenz mit korrektem Vorzeichen, wenn $E_{l,t}$ von $E_{k,t}$ abgezogen wird. Der rohe EWMAC $f_{roh,t}$ berechnet sich gem. Formel 8.

Formel 8: Roher EWMA-Forecast $f_{roh,t}$ für einen Zeitpunkt t

$$f_{roh,t} = E_{k,t} - E_{l,t}. \tag{8}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 283.

Die zweite getätigte Beobachtung ist die Niedrig-Volatilitäts-Anomalie. Gem. der in Kapitel 4.2.2 getätigten Beobachtung gehen geringe Volatilitäten regelmäßig mit Kurssteigerungen, hohe Volatilitäten mit Kursgefällen einher. Demnach muss der Forecast ein positives Vorzeichen besitzen, wenn der Volatilitätsindex v_{30} unter seinem Langzeitschnitt v_l liegt, und ein negatives Vorzeichen notieren,

99 Datenquelle: ARIVA, „DAX 30 Historische Kurse“.

sollte der Basiswert kurzfristig volatiler sein. Die Volatilitätsdifferenz $f_{roh,t}$ berechnet sich gem. Formel 9.

Formel 9: Roher Volatilitätsdifferenzforecast $f_{roh,t}$ für einen Zeitpunkt t

$$f_{roh,t} = v_l - v_{30}. \tag{9}$$

Dynamische Regeln wie der EWMAC oder die Volatilitätsdifferenz arbeiten deutlich feiner als binäre Regeln, die nur eine Auskunft darüber geben, welche Position einzunehmen ist, die Größe der Position aber nicht indizieren. Sie geben einen differenzierteren Forecast ab, auf dessen Basis eine exakte Positionsgröße bestimmt werden kann. Zur Kalkulation der tatsächlichen Positionsgröße ist f_{roh} noch nicht geeignet, weshalb dieser unter Beachtung weiterer Voraussetzungen für quantifizierte Vorhersagen im folgenden Kapitel 4.2.7 weiter verfeinert wird.

Über die hier entwickelten Regeln hinaus sollte sich jeder Anleger auch selbst Regeln überlegen, die seine eigenen Ansichten auf dem Finanzmarkt widerspiegeln; wendete jeder Anleger dieselben Regeln zum Kaufen und Verkaufen von Finanzprodukten an, resultierte dies in chaotischen Märkten aufgrund hoher Handelsvolumina an zu kurz aufeinanderfolgenden Zeitpunkten. Einige grundlegende Voraussetzungen, die sich in den obigen Ausführungen bereits widerspiegeln, sollten hierbei beachtet werden[100]:

- Einfach und objektiv. Regeln sollten auf ihre Grundaussage heruntergebrochen werden und objektiv ausführbar sein. Eine Regel, die das Bauchgefühl des Investors zur Durchführung verlangt, ist nicht objektivierbar.
- Fortlaufend statt binär. Binäre Regeln erlauben zwei Aussagen zur Position: Long oder Short. Bewegt sich ein Forecast um den Hold-Wert, werden häufig Transaktionen ausgelöst, wobei jeweils eine komplette Position bezogen wird. Fortlaufende, quantifizierbare Forecasts ermöglichen eine differenziertere Anlage.
- Vorhersagenvariabel. Kauf, Verkauf, und Positionsgröße sollten alle von derselben Regel bestimmt werden. Auch sollte sich der Fo-

[100] Aufzählung vgl. Carver, *Systematic Trading*, S. 121.

> recast mit dem Eingang neuer Basiswertdaten anpassen und unabhängig von der gehaltenen Position sein. Getrennte Start- und Stopp-Regeln sind nicht empfehlenswert.

Damit eine Regel in das Tradingsystem aufgenommen werden kann, sollte sie kostengünstig implementierbar sein und dem System einen Mehrwert bieten. Auch hierbei sollte der Anleger vermeiden, ausschließlich die Rendite als Kriterium heranzuziehen. Eher sollten vom Ertrag unabhängige Parameter zur Bewertung genutzt werden.[101]

Korrelieren zwei Regeln stark miteinander (95% Korrelation oder mehr), bringt das Hinzufügen beider Regeln keinen Mehrwert gegenüber der ausschließlichen Nutzung einer der beiden[102], sondern kann sogar einen unerwünschten gegenteiligen Effekt haben: U. U. werden überproportional große Positionen bezogen, die das Portfolio aus dem Gleichgewicht bringen und anfälliger für starke Kursverluste machen.

Bei der Formulierung von Regeln nach dem Ideas First-Ansatz gilt die Redewendung: „Weniger ist mehr“. Je einfacher die Regel, desto leichter ist mit ihr zu argumentieren, da das durch sie dargestellte Verhalten leichter erklärt werden kann. Auch quantitativ gilt diese sprichwörtliche Maßgabe: Wenige Regeln bedeuten eine verringerte Gefahr des Over-Fittings, da an weniger Stellen potenzielle Fehler auftreten können. Eine Diversifizierung gleicher Regeln auf mehrere Finanzinstrumente ist der Vermehrung von Regeln pro Instrument zu bevorzugen.[103]

Zu Zwecken der Demonstration der Kombination von Regeln wird das ausgewählte Beispielportfolio über die beiden oben beschriebenen Regeln gehandelt, wobei der EWMAC in drei Variationen in das Tradingsystem einfließt: Der kurzfristig orientierte 2-/8-Perioden-EWMAC, der mittelfristige 8-/32-Perioden-EWMAC, und der langfristig betrachtende 32-/128-Perioden-EWMAC werden zusammen mit einer Volatilitätsdifferenz, die aus den letzten vier Beobachtungen die aktuelle Volatilität bestimmt, auf alle drei Produkte angewandt. Im folgenden

---

101 Vgl. Carver, *Systematic Trading*, S. 68.
102 Vgl. Carver, *Systematic Trading*, S. 122.
103 Vgl. Carver, *Systematic Trading*, S. 58.

Kapitel werden die beiden Regeln mit ihren insgesamt vier Variationen quantifiziert und im Tradingsystem gewichtet.

4.2.7 Skalierung des Systems

Das hier zu entwickelnde System basiert auf quantitativen Vorhersagen über die künftige Entwicklung von Basiswerten. Dabei sollen die Vorhersagen vergleichbar sein und sich auf einer verständlichen Skala bewegen. Hierfür wird eine Basis-Skalierung benötigt, anhand derer die Aussagen des hiesigen Systems bemessen werden.

Sind die Forecasts aus einer Regel kalkuliert (siehe dazu auch die Ausführungen im folgenden Kapitel) werden diese entsprechend ihrer Größenordnung skaliert, sodass alle Vorhersagen, unabhängig von Regel und Basiswert, auf dieser Basis-Skalierung verlaufen. Carver schlägt für sein Tradingsystem eine Skalierung um den Basiswert 10 vor.[104] Dies bedeutet, dass die Summe der absoluten Werte einer so skalierten Datenreihe 10 ergibt. Als Maximalwerte setzt er dabei die doppelte Basis fest, d.h. im positiven Bereich wird mit Zahlen ≤ 20, im negativen Bereich ≥ -20 gearbeitet. Die gewählte Größenordnung wird so gewählt, dass sich unterschiedliche Werte in greifbare Relationen zueinander setzen lassen und vom Anleger auch überschlagsweise verglichen werden können. Diese vorgeschlagene Skalierung wird für das hiesige System übernommen. Diese Skalierung ist die Grundlage jeglicher weiteren Berechnung.

4.2.8 Quantifizierbare Vorhersagen

Ein wichtiger Bestandteil der Anlage in Vermögensprodukte ist die Fähigkeit vorherzusagen, wie sich das Portfolio entwickelt. Nur die Anlegerinnen und Anleger, die quantitative Aussagen über künftige Marktverhältnisse treffen, können Gewinne realisieren. Dabei ist nicht erheblich, in welche Richtung sich die Produkte entwickeln: Durch die Nutzung von Derivaten kann auch auf eine Verringerung eines Basiswertes gesetzt werden. Weiterhin führt Quantifizierbarkeit einer Vor-

[104] Vgl. Carver, *Systematic Trading*, S. 112 f.

hersage zu einer Kosteneinsparung im Handeln: Würden nur qualitative Vorhersagen getroffen, würden alle Positionen in voller Höhe, und nicht etwa anhand der erwarteten Renditen bezogen werden. Bei einem indizierten Positionswechsel würde die gesamte Position ver- und die eine volle Gegenposition gekauft werden.[105]

Über die beiden in Kapitel 4.2.2 formulierten und in den darauffolgenden Kapiteln ausgearbeiteten Annahmen können solch quantifizierte Aussagen getroffen werden. Durch das Messen der Differenz zweier EWMAs ist ein gleitender Wert ablesbar, aus dem sich wiederum ein Forecast berechnen lässt. Auch die Differenz zwischen aktueller und historischer Volatilität ist eine quantitative Aussage, sodass keine Anpassung der Regeln erforderlich ist.

Weiterhin sollten die über ein System getroffenen Vorhersagen sowohl dem zeitlichen Vergleich als auch dem Abgleich über Finanzprodukte hinweg standhalten. Hierfür ist eine Normalisierung der Vorhersage um die Volatilität des Basiswertes unabdingbar. Nur so können Vorhersagen eines Basiswertes mit denen anderer Basiswerte und auch mit historischen Vorhersagen verglichen werden.[106]

Um diese Einschränkungen einzuhalten, wird der Forecast um die Volatilität standardisiert. Vorhersagen über die Entwicklung eines Finanzinstrumentes werden somit immer proportional zu risikobereinigten Erträgen aufgestellt. Zwei beispielhafte Finanzprodukte können sich in Ertrag und Standardabweichung unterscheiden, standardisiert werden die zu erwartenden Erträge vergleichbar. Ein beispielhaftes Produkt A hat eine erwartete Rendite von 2% und eine Standardabweichung von 8%, während Produkt B mit einem Ertrag von 1% und einer Volatilität von 2% startet. Obwohl Produkt A eine zweifache Rendite mitbringt, ist der risikobereinigte Ertrag für Produkt B ($\frac{1\%}{2\%} = 0{,}5$) doppelt so hoch wie derjenige für Produkt A ($\frac{2\%}{8\%} = 0{,}25$). Darüber hinaus ermöglicht das regelmäßige Neuberechnen dieser Kennzahl eine Anpassung der Vorhersagen. Sollte sich abzeichnen,

[105] Vgl. Carver, *Systematic Trading*, S. 111.
[106] Vgl. Carver, *Systematic Trading*, S. 112.

dass ein Produkt nicht mehr dem Risikoprofil des Anlegers entspricht, kann dieser agieren und sein Risiko minimieren.[107]

Der über Formel 8 ermittelte rohe EWMAC $f_{roh,t}$ berücksichtigt die Standardabweichung des zugrundeliegenden Basiswertes noch nicht. Um dies zu erreichen, wird $f_{roh,t}$ gem. Formel 10 durch die Standardabweichung der täglichen Kursänderungen $\sigma_{x,E,t}$ dividiert, um die risikobereinigte EWMA-Differenz $f_{ber,t}$ zu erhalten.

Formel 10: Bereinigter Forecast $f_{ber,t}$ zum Zeitpunkt t

$$f_{ber,t} = \frac{f_{roh,t}}{\sigma_{x,E,t}}. \tag{10}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 112.

Die Standardabweichung der Kursänderungen $\sigma_{x,E,t}$ berechnet sich gem. Formel 11 durch Multiplikation des Kurses $k_{x,t}$ des Basiswertes x zum Zeitpunkt t mit der Wurzel des 36-Tage-EWMA der Volatilität des Basiswertes $E_{v,x,t}$. Die Halbwertszeit des 36-Tage-EWMA entspricht derjenigen eines 25-Tage ungewichteten gleitenden Mittelwerts.[108]

Formel 11: EWMA-basierte Standardabweichung $\sigma_{x,E,t}$ der Datenreihe x zum Zeitpunkt t

$$\sigma_{x,E,t} = k_{x,t} * \sqrt{E_{v,x,t}}. \tag{11}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 298 f.

Die Volatilität des Basiswertes $E_{v,x,t}$ errechnet sich als 25-Tages-EWMA gem. Formel 2, wobei der Kurswert k_t durch die quadrierte Rendite ${i_{x,t}}^2$ ersetzt wird. Die Quadratur der Rendite dient dazu, negative Werte zu eliminieren und sie dennoch korrekt gewichtet am EWMA teilhaben zu lassen. Die Rendite $i_{x,t}$ einer

[107] Vgl. Carver, *Systematic Trading*, S. 112.
[108] Vgl. Carver, *Systematic Trading*, S. 298 f.

Datenreihe x zum Zeitpunkt t errechnet sich gem. Formel 12 aus den Kurswerten zu den Zeitpunkten t und $t-1$.

Formel 12: Rendite $i_{x,t}$ einer Datenreihe x zum Zeitpunkt t

$$i_{x,t} = \frac{(k_{x,t} - k_{x,t-1})}{k_{x,t-1}}. \tag{12}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 298 f.

Auch die rohe Differenz zwischen Volatilitätsindex und Langzeitdurschnitt muss zur Erfüllung der oben gestellten Vorgaben um die Volatilität des Basiswertes standardisiert werden. Dies mag kontraproduktiv erscheinen – schließlich ist der Sinn dieser Regel, die Schwankungen eines Basiswertes einzufangen und sie nicht wieder zu bereinigen. Eine Diskussion hierüber findet sich in Kapitel 5. Analog der Berechnungen beim EWMAC wird diese gem. Formel 10 bestimmt.

Zur Erreichung der in Kapitel 4.2.7 festgelegten Größenordnung ist ein Skalar F erforderlich, der gem. Formel 13 aus dem Durchschnitt der absoluten bereinigten Forecasts $\left|\overline{f_{ber}}\right|$ auf eine akzeptable Skala hebt oder senkt.[109] S entspricht dabei dem definierten Basis-Skalawert, hier 10.

Formel 13: Forecast-Skalar F

$$F = \frac{S}{\left|\overline{f_{ber}}\right|}. \tag{13}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 297.

Durch Multiplikation der risikobereinigten Vorhersagen $f_{ber,t}$ mit dem Skalar F wird der skalierte Forecast $f_{scal,t}$ zum Zeitpunkt t gem. Formel 14 ermittelt.

[109] Vgl. dazu die Ausführungen in Kapitel 4.2.7.

Formel 14: Skalierter Forecast f_{scal}

$$f_{scal,t} = f_{ber,t} * F. \tag{14}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 297.

Analog zur Berechnung des resultierenden Forecasts des EWMACs muss auch der rohe Vorhersagewert der Volatilitäten skaliert werden. Hierfür werden Formel 13 und Formel 14 entsprechend angewandt.

Als Abwandlung zu dieser Berechnung könnte ein risikoaverserer Anleger den resultierenden Forecast der Volatilitätsdifferenzen auch als absolutes Limit ansehen. Nähme die Vorhersage f_{scal} einen Wert < 0 an, würde die Position bei 0 verbleiben, bis die Referenz-Volatilität wieder in einen Wertebereich unter ihrem Langzeitschnitt läge. U.U. ist es dem Anleger überhaupt nicht möglich Short-Positionen einzugehen. In diesem Falle würden bei allen Regeln $f_{scal} < 0$ auf 0 angehoben werden.

Trotz angemessener Risikobereinigung und Skalierung sollte der Anleger bei Forecasts jenseits des doppelten Skalawertes, in diesem Falle also bei ± 20 Punkten, Vorsicht walten lassen: Die selbst ermittelte Vorhersage ist keine Garantie für eine tatsächliche Entwicklung des entsprechenden Produktes auf die vorhergesagte Weise. Forecasts dieser Größenordnung könnten, bei Verbreitung des Tradingsystems, überproportionale Kaufs- bzw. Verkaufsstürme auslösen, die in den Finanzmarkt in Zustände wie zur Finanzkrise 2008/2009 stürzten. Carver empfiehlt daher eine Kappung des Forecasts bei ± 20 Punkten, was für diese Arbeit übernommen wird. Neben den bereits genannten Gründen ermöglicht dies eine quantifizierte Beherrschung des Risikos, das mit der Anlage eingegangen wird. Durch das Setzen eines Limits werden zwar mögliche Gewinne gedeckelt; dasselbe Limit begrenzt aber auch maximal mögliche Verluste.[110]

Nach Skalierung und Kappung beim doppelten Skalawert bewegen sich die Forecasts im Bereich zwischen -20 und $+20$, wobei eine Vorhersage von $+10$ einer

[110] Vgl. Carver, *Systematic Trading*, S. 113 f.

durchschnittlichen Long-Empfehlung, -10 einer durchschnittlichen Short-Empfehlung entspricht. Proportional dazu entspricht ein Forecast von $+5$ einer schwachen Long-Empfehlung, während -20 eine starke Short-Empfehlung anzeigt.[111]

4.2.9 Kombination von Vorhersagen

Im Falle des hiesigen Systems werden mehrere Forecasts auf ein Finanzinstrument angewandt. Damit der Gesamtforecast für dieses Instrument auf der zuvor definierten Skala verbleibt, wird eine Möglichkeit benötigt, die einzelnen Forecasts entsprechend miteinander zu kombinieren. Hierfür wird für jeden Forecast einer Regel x eine Gewichtung w_x gem. der Kombinations- und Gewichtungsregeln aus Kapitel 4.2.4 ermittelt und gem. Formel 15 mit dem zugehörigen skalierten Forecast $f_{scal,x,t}$ multipliziert. Die entstehenden Produkte aller Vorhersagen werden aufsummiert und ergeben so den summierten Forecast $f_{sum,t}$ zum Zeitpunkt t. So kann auf jedes Finanzinstrument eine beliebige Anzahl an Tradingregeln angewandt werden. Jede Regel oder Variante ergibt exakt eine Vorhersage pro Finanzinstrument.[112]

Formel 15: Summierter Forecast $f_{sum,t}$ zum Zeitpunkt t

$$f_{sum,t} = \sum w_x * f_{scal,x,t}. \tag{15}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 127.

Für die im Rahmen dieser Arbeit betrachteten Basiswerte ergeben sich nach Kombination der Regeln und Variationen Gewichtungen gem. Tabelle 11.

111 Vgl. Carver, *Systematic Trading*, S. 112 f.
112 Vgl. Carver, *Systematic Trading*, S. 123.

Tabelle 11: Kombination der Tradingregeln im Beispielportfolio

Basis-wert	Regel	Varia-tion	Stufe 1 (gerun-det)	Stufe 2	Gesamt
DAX	EWMAC	2/8	38,71%	50%	19,355%
		8/32	24,57%		12,285%
		32/128	36,72%		18,36%
	Volatilitäts-differenz		100%	50%	50%
STOXX	EWMAC	2/8	38,13%	50%	19,065%
		8/32	25,06%		12,53%
		32/128	36,8%		18,4%
	Volatilitäts-differenz		100%	50%	50%
S&P	EWMAC	2/8	40,15%	50%	20,075%
		8/32	24,51%		12,255%
		32/128	35,34%		17,67%
	Volatilitäts-differenz		100%	50%	50%

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[113]

Unter der Verwendung von mehr als drei Regeln findet, analog zur Gewichtung des Portfolios in Kapitel 4.2.4, eine Verteilung in Gruppen von maximal drei Regeln statt. Auch bei den Forecasts wird vom Speziellen ins Generelle zusammengefasst. So werden in einer ersten Stufe Variationen einer Regel zusammengefasst, anschließend über Regeln hinweg. Bei einer großen Anzahl von Regeln

[113] Datenquelle: ARIVA, „DAX 30 Historische Kurse“; ARIVA, „Euro Stoxx 50 Historische Kurse“; ARIVA, „S&P 500 Historische Kurse“.

kann u.U. noch immer eine Verteilung vonnöten sein, sodass an dieser Stelle entsprechend der Art der Regeln (z.B. trendfolgend) zusammengefasst werden kann.

„Diversification … [is] the only free lunch in investing."[114] Gem. moderner Portfoliotheorie verringert sich das Risiko eines Portfolios, je diversifizierter dieses ist. Diese Schlussfolgerung lässt sich auch auf die Kombination von Regeln übertragen und hat zur Folge, dass kombinierte Forecasts eine geringere Standardabweichung aufweisen als die einzelnen Vorhersagen. Daher bewegt sich der summierte Forecast auch nicht mehr auf der oben gewünschten Skala zwischen -20 und $+20$, sondern verzeichnet im Mittel geringere Ausschläge. Um dies auszugleichen, wird ein Multiplikator benötigt, der mit dem summierten Forecast verrechnet wird.[115]

Hierfür werden die Korrelationen aller Regeln $r_{x,y}$ zueinander in einer Matrix aufgetragen und mit den finalen Gewichtungen w_x bzw. w_y gem. Tabelle 11 (letzte Spalte) beider die Korrelation ergebenden Regeln multipliziert. Die diagonal verlaufenden Selbstkorrelationen werden dabei ebenfalls berücksichtigt. Aus der Summe der entstehenden Produkte werden die Wurzel gezogen und der Kehrwert gebildet. Auf diese Weise wird ein Diversifikationsmultiplikator M_d proportional zur Volatilitätsabschwächung der kombinierten Forecasts gebildet, welcher den finalen Forecast wieder in die oben definierte Skala hebt.[116] Formel 16 zeigt die Berechnung auf.

Formel 16: Diversifikationsfaktor M_d

$$M_d = \frac{1}{\sqrt{\sum w_x * w_y * r_{x,y}}}. \tag{16}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 297 f.

[114] Becker, „Diversification".
[115] Vgl. Carver, *Systematic Trading*, S. 128 ff.
[116] Vgl. Carver, *Systematic Trading*, S. 297 f.

Das Produkt aus diesem Multiplikator und dem zuvor errechneten summierten Forecast ergibt den resultierenden Forecast $f_{res,t}$ gem. Formel 17.

Formel 17: Resultierender Forecast $f_{res,t}$ zu einem Zeitpunkt t

$$f_{res,t} = f_{sum,t} * M_d. \tag{17}$$

Quelle: In Anlehnung an Carver, *Systematic Trading*, S. 169.

Auch $f_{res,t}$ wird bei Werten > 20 bzw. < -20 gekappt, sodass keine extremen Positionen eingegangen werden. $f_{res,t}$ ist der Indikator für das Vertrauen in das Wachstum des Basiswertes: Positive $f_{res,t}$ indizieren, dass der Basiswert steigt, während negative $f_{res,t}$ einen Rückgang des Kurses anzeigen.

4.2.10 Backtest

Die in Kapitel 4.2.9 ermittelten Vorhersagen über die Entwicklung der einzelnen Basiswerte müssen für eine Performancemessung in reale Positionsgrößen übersetzt werden. Um das Übertragen eines Forecasts auf unterschiedliche Kapitale und Finanzprodukte möglich zu machen, muss die zu bestimmende Positionsgröße unabhängig von der zuletzt eingenommenen Position sein, und sich zugleich variabel mit dem zur Verfügung stehenden Kapital ändern. Hierfür wird das zu einem Handelszeitpunkt t zur Verfügung stehende Kapital $c_{ante,t}$ gem. Formel 18 aus dem Kapital nach Handel $c_{post,t-1}$, sowie der Positionsgröße $s_{p,t-1}$ der Vorperiode t_{-1} und den jeweils dazugehörigen Preisen pro Stück $p_{p,t}$ berechnet, wobei der Index p die Positionen Long und Short widerspiegeln kann.

Formel 18: Zur Verfügung stehendes Kapital $c_{ante,t}$ zum Zeitpunkt t

$$c_{ante,t} = c_{post,t-1} + \sum s_{p,t-1} * p_{p,t}. \tag{18}$$

Da nach dem hiesigen Handelssystem immer ausschließlich Long- oder Short-Positionen je Basiswert gehalten werden, würden zur Berechnung von $c_{ante,t}$ regelmäßig die Stückzahlen und Preise nur einer der beiden Positionen (Long bzw. Short) ausreichen, da die Stückzahl der jeweils anderen Position zu diesem Zeit-

punkt 0 beträgt. Zur Einhaltung der oben gestellten Bedingung der Unabhängigkeit und zu Zwecken der besseren Übersichtlichkeit werden im Folgenden stets beide Positionen berücksichtigt, auch wenn eine der beiden keiner realen Positionsgröße entspricht.

Das Kapital $c_{ante,t}$ entspricht demjenigen Geldwert, der zu jedem Handelszeitpunkt erneut in den Finanzmarkt investiert werden kann. Demnach lässt sich proportional aus diesem die volle Positionsgröße $s_{voll,p,t}$ bei einem Forecast von 20 bzw. -20 gem. Formel 19 bestimmen. Diese Berechnung wird pro Handelszeitpunkt nur einmal ausgeführt, abhängig vom Vorzeichen des Forecasts. Bei einer Vorhersage von 0 wird die Positionsgröße nicht bestimmt, sondern ebenfalls auf 0 gesetzt.

Formel 19: Volle Positionsgröße $s_{voll,p,t}$ einer Position p zum Zeitpunkt t

$$s_{voll,p,t} = \frac{c_{ante,t}}{p_{p,t}}. \tag{19}$$

Die volle Positionsgröße wird, um der tatsächlichen Vorhersage $f_{res,t}$ proportional zu entsprechen, auf diese gem. Formel 20 per Dreisatz heruntergerechnet und auf die nächste Ganzzahl abgerundet. S entspricht der in Kapitel 4.2.7 bestimmten Basis-Skala.

Formel 20: Positionsgröße einer Position p nach Forecast $s_{p,t}$ zum Zeitpunkt t

$$s_{p,t} = \left\lfloor \frac{s_{voll,p,t}}{2 * S} * f_{res,t} \right\rfloor. \tag{20}$$

Über die Größe der tatsächlich eingegangenen Position lässt sich gem. Formel 21 letztlich noch das Kapital nach Handel $c_{post,t}$ bestimmen. Dieses entspricht dem Anteil der Geldsumme aus $c_{ante,t}$, der nicht in den Handel investiert wurde.

Formel 21: Kapital nach Handel $c_{post,t}$ zum Zeitpunkt t

$$c_{post,t} = c_{ante,t} - \sum s_{p,t} * p_{p,t}. \tag{21}$$

Zur finanzproduktunabhängigen Berechnung von Positionsgrößen (und daraus resultierend: Performancewerte) werden für beide möglichen Positionen entsprechende Produktpreise $p_{p,t}$ benötigt. Diese werden gem. Formel 22 aus dem Kurswert $k_{x,t}$ zum Zeitpunkt t und dem Produktpreisfaktor $M_{p,x}$ zur Datenreihe x berechnet.

Formel 22: Berechnung des Produktpreises $p_{p,t}$ einer Position p zum Zeitpunkt t

$$p_{p,t} = k_{x,t} * M_{p,x}. \tag{22}$$

Der Produktpreisfaktor $M_{p,x}$ berechnet sich gem. Formel 23 aus dem gekehrten arithmetischen Mittel der Kurswerte $\overline{k_x}$ der Datenreihe x.

Formel 23: Berechnung des Produktpreisfaktors $M_{p,x}$ für die Datenreihe x

$$M_{p,x} = \frac{1}{\overline{k_x}}. \tag{23}$$

Zudem wird für den Backtest eine Möglichkeit benötigt, Short-Positionen darzustellen. Gibt es für einen Basiswert keinen Short-Index, wird dieser analog dem ShortDAX berechnet, ein von der Deutschen Börse Gruppe offiziell kalkulierter Short-Index auf den DAX. Aufbauend auf dem Basiswert wird, beginnend mit dem ersten Tag der vorhandenen Daten, ein entsprechender Short-Index ermittelt. Dieser Short-Index wird, an Stelle eines konkreten Short-Instrumentes, für die Performancemessung genutzt.

Dabei entwickelt sich der ShortDAX gegenläufig zum DAX, d.h. die Renditen des DAX werden negiert und auf den ShortDAX multipliziert. Fällt der DAX in einem Zeitraum um 1%, so steigt der ShortDAX gleichermaßen um 1%.[117] Als Startwert für die Short-Indizes werden 1.000 Punkte vergeben. Tabelle 12 verdeutlicht die

[117] Vgl. Gruppe Deutsche Börse, „ShortDAX® Indizes“, S. 1.

Berechnung eines Short-Indexes. Weiterhin zeigt Tabelle 12 auch, dass sich Basiswert und Short-Index nicht umgekehrt proportional entwickeln. Die Abweichung des Short-Indexes nach $t \geq 2$ Tagen ist proportional zur Volatilität des Basiswertes: Stärkere Kursschwankungen des Basiswertes resultieren in einer größeren Abweichung des Short-Indexes.

Tabelle 12: Entwicklung eines beispielhaften Short-Index

Periode	Basiswert [Punkte]	Rendite des Basiswertes	Short-Index [Punkte]
0	1.000	-	1.000
1	1.100	10%	$900 = 1.000 + 1.000 * (-10\%)$
2	1.000	−9,091%	$981{,}818 = 900 + 900 * (9{,}091\%)$

Rein rechnerisch könnte der Short-Index negative Werte erreichen, und zwar dann, wenn der Basiswert in einer Handelsperiode mehr als 100% Rendite erzielt. Um zu vermeiden, dass ein Short-Index solche nichtgestatteten Werte erreicht, wird die Berechnung bei einer Kursänderung des Basiswertes um mehr als 50% insoweit angepasst, dass seine Berechnungsgrundlage auf den letzten Punkt vor der mehr als 50%igen Veränderung festgesetzt wird. Somit wird die Volatilität der Short-Indizes praktisch bei 50% gekappt.[118] Für die Zwecke des hier vorgestellten Systems wird diese Regel übernommen: Bei einer Rendite von über 50% zwischen zwei Perioden wird diese auf 50% reduziert. Negative Renditen bleiben hiervon unberührt und werden gänzlich berücksichtigt.

Für die Durchführung des Backtests wird schließlich ein Kaufen und Verkaufen der Long- und Short-Produkte simuliert. Dabei wird z.B. bei einem Forecast von 20 die volle Long-Position bezogen, bei einem Forecast von -10 wird die Hälfte des zur Verfügung stehenden Kapitals in Short-Produkte investiert. Das nach der

[118] Vgl. STOXX Ltd., „Guide to the DAX Strategy Indices“, S. 33 f.

letzten Periode vorhandene Kapital, addiert zum Wert der aktuell gehaltenen Position, ergibt dabei den erwirtschafteten Gewinn, abzüglich des ursprünglich investierten Kapitals.

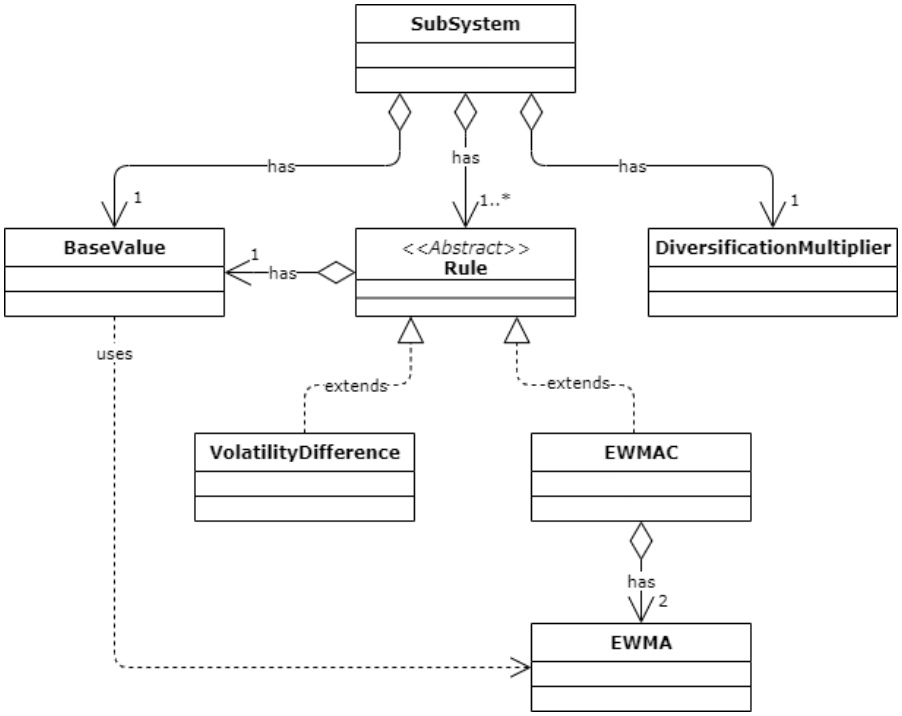
4.3 Systementwurf

Zur Realisierung der oben formulierten Anforderungen an das System wird eine Bibliothek vorgeschlagen, die zum einen ein voll verwendbares System darstellt, zum anderen aber auch eine Erweiterbarkeit durch weitere noch zu entwickelnde Regeln erlaubt.

4.3.1 Hauptkomponenten

Abbildung 7 zeigt die Hauptkomponenten des Systems.

Abbildung 7: Hauptkomponenten des Zielsystems



Die Hauptkomponenten decken den in Kapitel 4.2 definierten Funktionsumfang des Tradingsystems für einen Basiswert vollständig ab. Diese Komponenten sind:

- ***Rule***. Die Klasse *Rule* implementiert die Funktionalität, die für die Erstellung der Forecasts benötigt wird. ***VolatilityDifference*** und ***EWMAC*** sind zwei hiervon abgeleitete Klassen, die die Klasse *Rule* um Spezifika der in Kapitel 4.2.2 formulierten Ideen über den Finanzmarkt erweitern. Die Klasse ***EWMA*** bildet diejenigen Funktionen ab, die für die Berechnung des namensgebenden Mittelwerts benötigt werden.
- ***BaseValue***. *BaseValue* repräsentiert den Basiswert, für den dieses Subsystem Vorhersagen treffen soll.

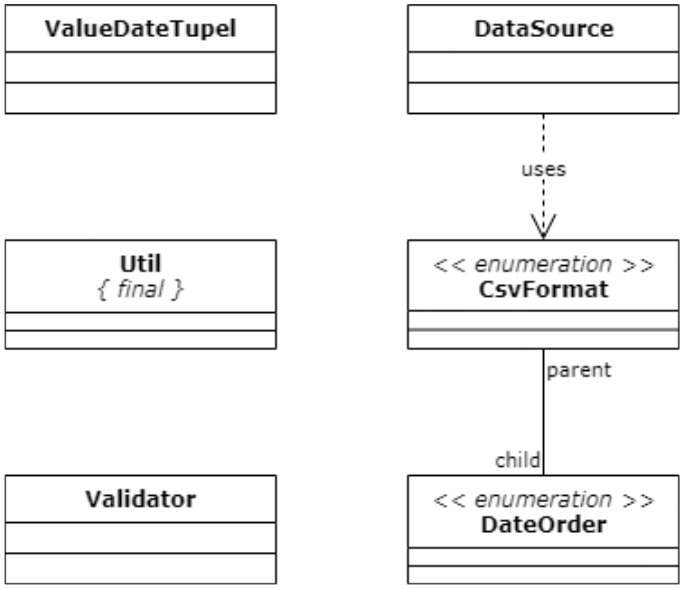
- ***SubSystem***[119]. Die Komponente *SubSystem* nimmt eine steuernde Rolle im System ein und orchestriert das Zusammenspiel der restlichen Komponenten.
- ***DiversificationMultiplier***. Diese Komponente erfüllt die Funktion des Diversifikationsfaktors, der die Forecasts der Regeln auf das gewünschte Skalenniveau hebt.

4.3.2 Hilfskomponenten

Neben den zuvor genannten Hauptkomponenten gibt es noch eine Reihe von Hilfskomponenten, die von den Hauptkomponenten verwendet werden. Diese bilden zumeist wiederverwendete Funktionalität ab, sodass die Implementierung der Hauptkomponenten prägnant und präzise gehalten werden kann. Abbildung 8 zeigt diese Hilfskomponenten.

[119] Das Wort „System“ ist in vielen Programmiersprachen ein reservierter Begriff bzw. stellt eine Referenz auf die Laufzeitumgebung oder das Betriebssystem dar. Weiterhin spricht Carver auf dieser Ebene ebenfalls von „Subsystem“, wobei pro Subsystem alle Regeln zu einem Basiswert zusammengefasst werden. Mehrere Subsysteme werden wiederum zum gesamten Portfolio zusammengefasst. Vgl. Carver, *Systematic Trading*, S. 98.

Abbildung 8: Hilfskomponenten des Zielsystems



Diese Hilfskomponenten decken unterschiedliche Funktionsbereiche ab, die wiederholt im gesamten System verwendet werden. Diese Hilfskomponenten sind:

- ***ValueDateTupel***. *ValueDateTupel* ist ein zentraler Baustein, der in beinahe allen anderen Komponenten verwendet wird. Er repräsentiert einen Gleitkommawert zu einem bestimmten Zeitpunkt.
- ***Util***. *Util* versammelt eine Menge mathematischer Operationen, die den Zweck der Hauptkomponenten verschleiern würden oder an verschiedenen Stellen zum Einsatz kommen. So wird eine doppelte Implementierung vermieden.
- ***Validator***. Die zu implementierende Bibliothek soll eine Reihe an öffentlichen Schnittstellen zur Verfügung stellen, die von anderen Entwicklern genutzt werden können. Die Komponente *Validator* bietet für diesen Zwecke eine Reihe an Validierungsmethoden, über die die Hauptkomponenten eine entsprechende Inputvalidierung durchführen können.
- ***DataSource***. Um mit Finanzdaten arbeiten zu können, müssen diese zuerst importiert und in ein für das System nutzbares Format

überführt werden. *DataSource* bietet eine Schnittstelle zum Import von Finanzdaten, welche in eine für die Hauptkomponenten verarbeitbare Reihe von *ValueDateTupel* gelesen werden. Zum Import von Dateien mit *kommaseparierten Werten (CSV)* muss deren Format bekannt sein. Die Enumeration ***CsvFormat*** bietet eine Unterscheidung der gängigsten CSV-Formate, sodass *DataSource* die eingelesene CSV-Datei korrekt verarbeiten kann. Ein Teil hiervon ist der möglicherweise unterschiedliche Aufbau von Datumsfeldern. Die Enumeration ***DateOrder*** liefert hierbei eine Unterscheidung zwischen diesen Aufbauten.

4.4 Implementierung

Dieser Abschnitt beschreibt, wie die zuvor beschriebene Struktur bzw. die in ihr enthaltenen Komponenten mit Funktion gefüllt werden und miteinander interagieren. Der Fokus wird dabei auf die Hauptkomponenten gelegt, wobei auch die Hilfskomponenten beschrieben werden. Abbildung 18 in Anhang 0zeigt das Klassendiagramm der gesamten Anwendung. In Anhang 0finden sich ebenfalls die Klassenbeschreibungen der einzelnen, im Folgenden näher beschriebenen Komponenten. Anhang A3[120] zeigt den gesamten im Rahmen dieser Arbeit entwickelten Quellcode.

Wie in Kapitel 1.2 erläutert, ist die gesamte Bibliothek in Java entwickelt. Dabei wird auf Funktionalität gesetzt, die erstmals mit Java 11 eingeführt wurde, weshalb zur Umwandlung des Sourcecodes mindestens ein *Java Development Kit (JDK)* 11 benötigt wird. Als Build-Tool wird der Industriestandard Maven in der Version 4.0.0 genutzt. Maven steuert darüber hinaus auch die Test-, Berichts- und Dokumentationserstellung. Listing 12 in Anhang A3[121] zeigt die für Maven relevante steuernde *pom.xml*.

---

[120] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

[121] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

Durch eine parallel mitentwickelte Unit-Test-Suite wird die Funktionalität der einzelnen Komponenten festgestellt. Das Ziel dieser Suite ist es, den Quellcode möglichst gänzlich abzudecken und dabei auch alle möglichen Verzweigungen zu betrachten. Auf diese Weise wird sichergestellt, dass Code-Änderungen keine unerwünschten Nebeneffekte nach sich ziehen und das System leichter weiterentwickelt werden kann. Die Testklassen sind mittels JUnit Jupiter (JUnit 5) erstellt. Die Ausführung der Test-Suite steuert das Maven Surefire Plugin. Das Reporting der Testabdeckung wird über das Maven Plugin JaCoCo (Java Code Coverage) durchgeführt. Weiterhin wird über das Maven Javadoc Plugin die Code-Dokumentation der gesamten Bibliothek erstellt und im resultierenden Bibliotheksfile mit abgelegt.

In den nachfolgenden Abschnitten über die Komponenten wird erläutert, welche der in Kapitel 4.2 formulierten Anforderungen jeweils erfüllt wird, sofern das jeweilige Kapitel einer bestimmten Komponente zugeordnet werden kann. Die in Kapitel 4.2.3 formulierte Anforderung bzgl. vollautomatisierter Anlage findet sich im gesamten Systementwurf wieder und kann nicht direkt einer Komponente zugeordnet werden.

4.4.1 Komponente Rule

Wie in Kapitel 4.2.6 erörtert, stellen die Regeln, nach denen die Entwicklung von Finanzprodukten bewertet wird, das Zentrum eines Tradingsystems dar. Sie entscheiden objektiv, nachvollziehbar und situationsabhängig darüber, ob auf ein Produkt vertraut wird und eine Long-Position eingenommen, oder ob von einer negativen Entwicklung ausgegangen und daher eine Short-Position bezogen wird.

Eine Anforderung an das System ist, dass dieses möglichst leicht durch weitere Regeln ergänzt werden kann. Dabei sollte möglichst nur die Funktionalität implementiert werden müssen, die sich von Regel zu Regel unterscheidet und somit nicht allgemein verwendbar ist. Jegliche weitere Funktionalität sollte bereits bestehen. Weiterhin muss gewährleistet sein, dass an den Stellen im System, an denen eine Regel erwartet wird, auch eine solche verwendet wird, die mit den restlichen Komponenten kompatibel ist.

Daher wird die Komponente ***Rule*** als abstrakte Klasseimplementiert. Dies ermöglicht das Vorgeben von Funktionalität, lässt aber gleichzeitig diejenigen Funktionalitäten offen, die von den erbenden Klassen, also den tatsächlichen Regeln, implementiert werden müssen. Daher fordert die Klasse *Rule* von seinen erbenden Klassen nur eine abstrakte Methode *calculateRawForecast(LocalDateTime forecastDateTime)* zur Berechnung des rohen Forecasts. Weitere, in den erbenden Klassen implementierte Methoden beeinflussen die anderen Komponenten nicht, sondern wirken allenfalls innerhalb der erbenden Klasse. Diese Abstraktion fordert und fördert somit den Ideas First-Ansatz, wie er in Kapitel 4.2.1 festgelegt wurde.

Listing 1 zeigt die Bestimmung der Gewichtungen der Variationen, die eine Regel hat. Weist eine Regel keine Variationen auf, müssen keine Gewichtungen kalkuliert werden. Andernfalls wird unterschieden zwischen einer, zwei, und drei Gewichtungen. Die ersten beiden Fälle sind trivial und ergeben 100% für eine Variation bzw. jeweils 50% für zwei Variationen. Im Falle von drei Variationen hingegen erfolgt eine Kalkulation entsprechend der in Kapitel 4.2.4 festgelegten Vorgehensweise. Die eigentliche Kalkulation der Gewichtungen findet innerhalb der Komponente *Util* (vgl. Listing 6) statt.

Listing 1: weighVariations() zur Bestimmung der Gewichtungen der Variationen einer Regel

```
1.   private void weighVariations() {
2.     Rule[] instanceVariations = this.getVariations();
3.     if (instanceVariations == null)
4.       return;
5.
6.     switch (instanceVariations.length) {
7.     case 1:
8.       /* If there is only 1 variation then its weight is 100% */
9.       instanceVariations[0].setWeight(1d);
10.      break;
11.
12.    case 2:
13.      /* If there are 2 variations their weights are 50% each */
14.      instanceVariations[0].setWeight(0.5d);
15.      instanceVariations[1].setWeight(0.5d);
16.      break;
17.
18.    case 3:
19.      /*
20.       * Extract the values from the forecasts array, as the Dates are not
21.       * needed for correlation calculation.
22.       */
```

```
23.      double[][] variationsForecasts = {};
24.      for (Rule variation : instanceVariations) {
25.        ValueDateTupel[] fcs = variation.extractRelevantForecasts();
26.        variationsForecasts = ArrayUtils.add(variationsForecasts,
27.            ValueDateTupel.getValues(fcs));
28.      }
29.
30.      /* Find the correlations for the given variations. */
31.      double[] correlations = Util
32.          .calculateCorrelationOfRows(variationsForecasts);
33.
34.      if (ArrayUtils.contains(correlations, Double.NaN))
35.        throw new IllegalArgumentException(
36.            "Correlations cannot be calculated due to illegal values"
37.                + " in given variations.");
38.
39.      /* Find the weights corresponding to the calculated correlations. */
40.      double[] weights = Util
41.          .calculateWeightsForThreeCorrelations(correlations);
42.
43.      /* Set the weights of the underlying variations */
44.      for (int i = 0; i < weights.length; i++) {
45.        instanceVariations[i].setWeight(weights[i]);
46.      }
47.      break;
48.
49.    default:
50.      throw new IllegalStateException(
51.          "A rule should not have this many variations: "
52.              + instanceVariations.length);
53.    }
54.  }
```

Durch die Nutzung der abstrakten Klasse *Rule* wird im restlichen System erzwungen, dass nur Regeln verwendet werden können, die diese Klasse erweitern. Klassen, die dem nicht entsprechen, sind für die Nutzung innerhalb der Bibliothek nicht zulässig. Weiterhin trägt die Komponente *Rule* zur Erfüllung der in Kapitel 4.2.7 geforderten Skalierung des Systems bei. Sie zieht für die Kalkulation der

Vorhersagenwerte die an sie übergebene Basisskalierung in Betracht und berechnet die Forecasts entsprechend. Listing 13 in Anhang A3[122] zeigt den gesamten Sourcecode der Komponente *Rule*.

Zwei beispielhaft erbende Klassen sind die Implementierungen in den Komponenten ***VolatilityDifference*** und ***EWMAC***. Diese realisieren die in Kapitel 4.2.2 formulierten Ideen über den Finanzmarkt. Listing 14 und Listing 15 in Anhang A3[123] zeigen jeweils den Sourcecode zu diesen Komponenten. Codiert werden musste hierbei jeweils ausschließlich diejenige Logik, die die Regel auszeichnet und von anderen Regeln unterscheidet. Die Komponente *VolatilityDifference* implementiert indes auch die in Kapitel 4.2.5 definierten Volatilitätsindizes. Diesen beiden Implementierungen stehen nur beispielhaft für die erläuterten finanzmathematischen Beobachtungen und können beliebig ausgetauscht werden. Listing 16 in Anhang A3[124] zeigt den Sourcecode zur Klasse ***EWMA***, die innerhalb von *EWMAC* verwendet wird.

4.4.2 Komponente BaseValue

Die Komponente ***BaseValue*** repräsentiert den Basiswert, für den mittels des hiesigen Systems Vorhersagen getroffen werden. Dieser Basiswert wird wiederum in den anderen Komponenten referenziert, sodass das gesamte System mit demselben Basiswert arbeitet. Darüber hinaus speichert sie auch die Short-Index-Werte. Diese Werte werden über die Methode *calculateShortIndexValues(ValueDateTupel[])* gem. der Vorgaben in Kapitel 4.2.10 ermittelt, sofern sie nicht bei der Instanziierung übergeben werden. Listing 2 zeigt die entsprechende Implementierung.

---

122 Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

123 Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

124 Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

Listing 2: calculateShortIndexValues(ValueDateTupel[]) zur Bestimmung der Short-Index-Werte

```
1.   private static ValueDateTupel[] calculateShortIndexValues(
2.       ValueDateTupel[] values) {
3.     /**
4.      * Declare the return value. There are always as many short index
5.      * values as there are base values.
6.      */
7.     ValueDateTupel[] calculatedShortIndexValues = ValueDateTupel
8.         .createEmptyArray(values.length);
9.     calculatedShortIndexValues[0] = new ValueDateTupel(values[0].getDate(),
10.        SHORT_INDEX_INITIAL_VALUE); // = SHORT_INDEX_INITIAL_VALUE = 1000
11.
12.    ValueDateTupel formerValue;
13.    ValueDateTupel latterValue;
14.
15.    /**
16.     * Loop over the provided values array and calculate the corresponding
17.     * short index value for every time interval t > 0.
18.     */
19.    for (int i = 1; i < values.length; i++) {
20.      formerValue = values[i - 1];
21.      latterValue = values[i];
22.
23.      double returnPercentagePoints = Util
24.          .calculateReturn(formerValue.getValue(), latterValue.getValue());
25.
26.      /**
27.       * If the base value generates more than 50% in returns (and thus
28.       * decreasing the short index value by more than 50%) the return
29.       * percentage is set to 50%.
30.       */
31.      if (returnPercentagePoints > 0.5)
32.        returnPercentagePoints = 0.5;
33.
34.      double shortIndexValue = calculatedShortIndexValues[i - 1].getValue()
35.          - calculatedShortIndexValues[i - 1].getValue()
36.              * returnPercentagePoints;
37.
38.      calculatedShortIndexValues[i] = new ValueDateTupel(
39.          latterValue.getDate(), shortIndexValue);
40.    }
41.
42.    return calculatedShortIndexValues;
43.  }
```

Weiterhin bestimmt die Komponente *BaseValue* die Standardabweichung für jeden Zeitpunkt. Diese berechnen sich gem. der Vorgaben in Kapitel 4.2.8. Die drei in diesem Kapitel genannten Wertereihen werden dabei jeweils durch ein Array

der Komponente *ValueDateTupel* (siehe Kapitel 4.4.5) dargestellt. Listing 17 in Anhang A3[125] zeigt den gesamten Sourcecode der Komponente *BaseValue*.

4.4.3 Komponente SubSystem

Die Komponente ***SubSystem*** kombiniert die Regeln mit einem Basiswert und ermittelt aus ihnen, unter Verwendung einer Instanz der Klasse ***DiversificationMultiplier*** (siehe Kapitel 4.4.4), den kombinierten Forecast. Weiterhin realisiert diese Komponente die in Kapitel 4.2.10 definierten Anforderungen an einen Backtest. Es werden fiktive Produktpreise kalkuliert, anhand derer die Performance der Regeln in Kombination mit dem Basiswert gemessen wird.

Die Klasse *SubSystem* bietet zwei mal zwei *public*-Methoden an (jeweils einmal als Instanzmethode und einmal als statische Methode), die für Backtesting-Zwecke genutzt werden. Die Methoden *calculatePerformanceValues(…)* liefern *ValueDateTupel[]*, die die Entwicklung des eingesetzten Kapitals zu allen aus dem Basiswert ermittelten Zeitpunkten widerspiegelt. Dabei wird ein Verkaufen der gehaltenen Positionen nach jeder Periode simuliert, sodass die ermittelte Performance die Kombination der gehaltenen Finanzprodukte und des nicht eingesetzten Kapitals darstellt. Listing 3 zeigt die statische Variante von *calculatePerformanceValues(…)*, welche über die kombinierten Forecasts die Performance der Regeln im Testzeitraum ermittelt.

Listing 3: calculatePerformanceValues(…) zur Bestimmung der Performance eines spezifischen Regelsatzes

```
1.    public static ValueDateTupel[] calculatePerformanceValues(
2.        BaseValue baseValue, LocalDateTime startOfTestWindow,
3.        LocalDateTime endOfTestWindow, ValueDateTupel[] combinedForecasts,
4.        double baseScale, double capital) {
5.
6.      ...
7.
8.      long longProductsCount = 0;
9.      long shortProductsCount = 0;
10.     for (int i = 0; i < relevantCombinedForecasts.length; i++) {
```

[125] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

```
11.
12.       /*
13.        * Calculate the capital available for this time interval by
14.        * "selling" off all currently held positions at this time interval‘s
15.        * prices.
16.        */
17.       capital += longProductsCount * productPrices[i].getValue();
18.       capital += shortProductsCount * shortProductPrices[i].getValue();
19.
20.       /* Reset the products count as they were sold off */
21.       shortProductsCount = 0;
22.       longProductsCount = 0;
23.
24.       /*
25.        * Add this capital as performance value, as the overall value of
26.        * cash + assets held will not change during buying.
27.        */
28.       ValueDateTupel performanceValue = new ValueDateTupel(
29.           relevantCombinedForecasts[i].getDate(), capital);
30.       performanceValues = ArrayUtils.add(performanceValues,
31.           performanceValue);
32.
33.       if (relevantCombinedForecasts[i].getValue() > 0) {
34.         /* Long position */
35.         longProductsCount = calculateProductsCount(capital,
36.             productPrices[i].getValue(),
37.             relevantCombinedForecasts[i].getValue(), baseScale);
38.
39.         /*
40.          * "Buy" the calculated count of products and thus reduce the cash
41.          * capital
42.          */
43.         capital -= longProductsCount * productPrices[i].getValue();
44.
45.       } else if (relevantCombinedForecasts[i].getValue() < 0) {
46.         /* short position */
47.         shortProductsCount = calculateProductsCount(capital,
48.             shortProductPrices[i].getValue(),
49.             relevantCombinedForecasts[i].getValue(), baseScale);
50.
51.         /*
52.          * "Buy" the calculated count of products and thus reduce the cash
53.          * capital
54.          */
55.         capital -= shortProductsCount * shortProductPrices[i].getValue();
56.       } else {
57.         /*
58.          * If forecast was 0 nothing would be bought so no default-else
59.          * branch is needed.
60.          */
61.       }
62.     }
63.     return performanceValues;
64.   }
```

Die zweite Methode, *backtest(…)*, liefert unter der Verwendung von *calculatePerformanceValues(…)* den Performance-Wert als *double* nach der letzten Tradingperiode. Dieser Wert sollte jedoch nur mit Vorsicht in Entscheidungen einfließen, da er keine Aussage über das Verhalten der Regel (z.B. die Korrelation der Performance zum Basiswert) zulässt. Listing 18 in Anhang A3[126] zeigt den Sourcecode von *SubSystem*.

Auch die Komponente *SubSystem* trägt durch ihre Implementierung zur Einhaltung der Anforderung bzgl. Skalierung gem. Kapitel 4.2.7 bei. Sie wendet die Basisskalierung auf die kombinierten und um die Diversifikation bereinigten Vorhersagen entsprechend an.

4.4.4 Komponente DiversificationMultiplier

Die Komponente ***DiversificationMultiplier*** implementiert das in Kapitel 4.2.9 vorgestellte Konzept des Diversifikationsfaktors. Über die Korrelationen der Forecasts im festgelegten Fitting-Zeitraum und der hieraus resultierenden Gewichtungen der Regeln wird gem. Formel 16 der Diversifikationsfaktor bestimmt. Dadurch wird auch die Anforderung bzgl. Skalierung gem. Kapitel 4.2.7 unterstützt.

Hierfür werden die übergebenen Regeln rekursiv durchschritten, sodass die Variationen jeder Ebene (Variationen können wiederum selbst Variationen haben) mit ihren jeweiligen Gewichtungen einfließen. Damit die Forecasts und Gewichtungen zueinander passen, werden die Regeln über einen gemeinsamen Algorithmus innerhalb der Methode *getWeightsAndForecastsFromRules(Rule[])* (vgl. Listing 4) solange durchschritten, bis alle Blatt-Gewichtungen und -Forecasts extrahiert wurden. Die dafür geschaffene innere Klasse ***WeightsAndForecasts*** nimmt dabei die Gewichtungen und Forecasts jeder Ebene auf und gibt diese an die darüberliegende, aufrufende Ebene weiter.

---

[126] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

Listing 4: getWeightsAndForecastsFromRules(Rule[]) zur rekursiven Ermittlung der Gewichtungen und Forecasts eingegebener Variationen

```
1.  private WeightsAndForecasts getWeightsAndForecastsFromRules(Rule[] rules) {
2.      double[] weightsFromRules = {};
3.      double[][] relevantForecastsFromRules = {};
4.
5.      /* Iterate over the given rules */
6.      for (Rule rule : rules) {
7.
8.          /* If a rule has variations get their weights and forecasts */
9.          if (rule.hasVariations()) {
10.             WeightsAndForecasts wafToAdd = getWeightsAndForecastsFromRules(
11.                     rule.getVariations());
12.             for (double weight : wafToAdd.weights)
13.                 weightsFromRules = ArrayUtils.add(weightsFromRules,
14.                         weight * rule.getWeight());
15.
16.             for (double[] forecasts : wafToAdd.forecasts)
17.                 relevantForecastsFromRules = ArrayUtils
18.                         .add(relevantForecastsFromRules, forecasts);
19.
20.         } else {
21.             double weight = rule.getWeight();
22.             /*
23.              * If a top level rule has no variations its weight has not been
24.              * set. Manually set its weight to be 1/numberOfTopeLevelRules
25.              */
26.             if (weight == 0)
27.                 weight = 1d / rules.length;
28.             weightsFromRules = ArrayUtils.add(weightsFromRules, weight);
29.
30.             relevantForecastsFromRules = ArrayUtils.add(
31.                     relevantForecastsFromRules,
32.                     rule.extractRelevantForecastValues());
33.         }
34.     }
35.     return new WeightsAndForecasts(weightsFromRules,
36.             relevantForecastsFromRules);
37. }
```

Die innere Klasse *WeightsAndForecasts* ist eine reine Containerklasse ohne jegliche Business-Logik, die lediglich aus einer Reihe von *double* zur Speicherung von Gewichtungen (*double[] weights*) und einer Reihe von *double*-Reihen zur Speicherung der relevanten Forecasts (*double[][] forecasts*) besteht. Sie wird ausschließlich innerhalb von *DiversificationMultiplier* verwendet, weshalb sie an

dieser Stelle nicht weiter beschrieben wird. Listing 19 in Anhang A3[127] zeigt den gesamten Sourcecode der Komponente *DiversificationMultiplier*, welcher auch die Klassenbeschreibung für *WeightsAndForecasts* beinhaltet.

4.4.5 Komponente ValueDateTupel

Die Komponente ***ValueDateTupel*** repräsentiert einen Gleitkommawert zu einem definierten Zeitpunkt. Durch ihren hohen Grad an Abstraktion eignet sich diese Klasse zur Darstellung vieler unterschiedlicher Sachverhalte und findet daher auch in allen Hauptkomponenten Anwendung. Durch die Verwendung der seit Java 8 angebotenen time-API (*java.time*) und der hierdurch verfügbaren Klasse *LocalDateTime*, welche die Kombination aus einem Datum (*LocalDate*) und einer Uhrzeit (*LocalTime*) darstellt, ist selbst eine Intratagesbetrachtung möglich, sodass sich bei der Nutzung dieser Library nicht auf den Handel nach Marktschluss beschränkt werden muss.

Neben ihrer simplen Prämisse bringt die Klasse *ValueDateTupel* einige statische Methoden mit, die das Arbeiten mit Arrays dieser Klasse erleichtern und im gesamten Projekt Anwendung finden. So ermöglicht die Methode *alignDates(ValueDateTupel[][])* in Listing 5, dass eigentlich nichtkompatible Datenreihen zueinander kompatibel sind. Hierfür werden die *LocalDateTime*-Werte aller eingegebenen Datenreihen abgemischt, sodass jede Reihe die *LocalDateTime*-Werte aller übergebenen Reihen aufweist.

Listing 5: alignDates(ValueDateTupel[][]) zur Angleichung von LocalDateTime-Werten mehrerer ValueDateTupel-Datenreihen

```
1.   public static ValueDateTupel[][] alignDates(
2.           ValueDateTupel[][] valueDateTupels) {
3.       if (valueDateTupels == null)
4.           throw new IllegalArgumentException(
5.                   "Given array of arrays must not be null");
6.
7.       /* TreeSet (unique and sorted) of all dates in all valueDateTupel[] */
8.       TreeSet<LocalDateTime> uniqueSortedDates = getUniqueDates(
9.               valueDateTupels);
```

[127] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

```
10.
11.     /* Loop over all rows */
12.     for (int rowIndex = 0; rowIndex < valueDateTupels.length; rowIndex++) {
13.
14.         try {
15.             /* Validate if the row contains at least one suitable value. */
16.             Validator.validateRow(valueDateTupels[rowIndex]);
17.         } catch (IllegalArgumentException e) {
18.             throw new IllegalArgumentException(
19.                     "Row at position " + rowIndex + " is not valid.", e);
20.         }
21.
22.         /*
23.          * If the row's length equals the length of uniqueSortedDates no
24.          * Value has to be added as it already contains all dateTimes.
25.          */
26.         if (valueDateTupels[rowIndex].length == uniqueSortedDates.size())
27.             continue;
28.
29.         /* Enhance current row by missing LocalDateTime values. */
30.         valueDateTupels[rowIndex] = enhanceRowByNaNs(
31.                 valueDateTupels[rowIndex], uniqueSortedDates);
32.
33.         /* Replace all values of Double.NaN by real values. */
34.         valueDateTupels[rowIndex] = replaceNansByValues(
35.                 valueDateTupels[rowIndex]);
36.
37.     }
38.     return valueDateTupels;
39. }
```

Der gesamte Sourcecode für die Komponente *ValueDateTupel* wird in Listing 20 in Anhang A3[128] wiedergegeben.

4.4.6 Komponente Util

Die Komponente ***Util*** bringt einige Utility-Funktionalitäten mit sich. In diese Komponente sind mathematische Vorgänge ausgelagert, die entweder in mehreren unterschiedlichen Komponenten oder mehrfach in einer anderen Komponente

[128] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

verwendet werden. Somit wird erreicht, dass diese Methoden, sollten sie verändert werden müssen, zentral gesammelt sind und die Funktionalität nicht an mehreren Stellen angepasst werden muss.

Listing 6 zeigt die Umsetzung der in Kapitel 4.2.4 formulierten Anforderungen an die Gewichtung von drei Datenreihen. Wie in Formel 3 ff. beschrieben, wird über die durchschnittlichen Korrelationen jeder einzelnen Datenreihe ein inverser Wert gebildet, welcher, normalisiert zur Basis 1, die Gewichtung dieser Reihe im Zusammenspiel mit den anderen beiden ergibt. Weiterhin tragen einige Methoden der Komponente ebenfalls zur Erfüllung der Anforderungen gem. 4.2.9 bei, indem sie entsprechende statische Methoden zur Kombination von Vorhersagen anbietet.

Listing 6: calculateWeightsForThreeCorrelations(double[]) zur Berechnung der Gewichtungen von drei Reihen

```
1.   public static double[] calculateWeightsForThreeCorrelations(
2.       double[] correlations) {
3.
4.     ...
5.
6.     /* Get the average correlation each row of values has */
7.     double averageCorrRowA = (correlations[0] + correlations[1]) / 2;
8.     double averageCorrRowB = (correlations[0] + correlations[2]) / 2;
9.     double averageCorrRowC = (correlations[1] + correlations[2]) / 2;
10.
11.    double[] averageCorrelations = { averageCorrRowA, averageCorrRowB,
12.      averageCorrRowC };
13.
14.    /*
15.     * Subtract each average correlation from 1 to get an inverse-ish value
16.     */
17.    for (int i = 0; i < averageCorrelations.length; i++)
18.      averageCorrelations[i] = 1 - averageCorrelations[i];
19.
20.    /* Calculate the sum of average correlations. */
21.    double sumOfAverageCorrelations = DoubleStream.of(averageCorrelations).sum();
22.
23.    /*
24.     * Normalize the average correlations so they sum up to 1. These normalized
25.     * values are the weights.
26.     */
27.    for (int i = 0; i < averageCorrelations.length; i++)
28.      weights =
29.        ArrayUtils.add(weights, averageCorrelations[i] / sumOfAverageCorrelations);
30.
31.    return weights;
32.  }
```

Der gesamte Code der Komponente *Util* ist in Listing 21 in Anhang A3[129] wiedergegeben.

4.4.7 Komponente Validator

Die Komponente ***Validator*** führt die Validierungsaufgaben aller Klassen in sich zusammen. Sobald eine Komponente oder Methode zwei oder mehr Validierungen auf einen Eingabewert durchführt, oder eine Validierung fünf Zeilen Code oder mehr in Anspruch nimmt, wird diese in eine Methode der Komponente *Validator* extrahiert. Die Prüfungen und Fehlermeldungen sind dabei möglichst generisch gehalten, sodass diese auf verschiedene Sachverhalte angewandt werden können. Listing 22 in Anhang A3[130] zeigt den Code für die Komponente *Validator*.

4.4.8 Komponente DataSource

Bevor eine Analyse von Finanzdaten stattfinden kann, müssen diese in eine dem Programm verständliche Form gebracht werden. Hierfür werden bestehende Datenquellen genutzt, sodass aus diesen die gewünschten Daten bezogen werden können.

Die für diese Arbeit hauptsächlich genutzte Plattform ARIVA.de bietet einen Export der historischen Daten als CSV-Datei an. Solche CSV-Daten können, sofern sie einen entsprechenden Aufbau aufweisen, über die Komponente ***DataSource*** eingelesen werden. Diese gibt die eingelesenen Daten wiederum als

---

[129] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

[130] Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

ValueDateTupel[] aus, sodass diese direkt in den übrigen Komponenten verwendet werden können. Listing 23 in Anhang A3[131] zeigt den Sourcecode der Komponente *DataSource*.

Da CSV-Dateien nicht immer gleich aufgebaut sind, sondern sich bspw. durch das genutzte Trennzeichen unterscheiden können, benötigt *DataSource* zum Verarbeiten einer CSV-Datei genau diese Information. Die Enumeration ***CsvFormat*** repräsentiert dabei die für diese Arbeit relevanten CSV-Formate EU und US, die sich u.a. durch die genutzten Trennzeichen, Tausenderzeichen, und Dezimaltrennzeichen voneinander unterscheiden. Zusätzlich unterscheidet sich zumeist auch der Aufbau von Datumsfeldern. Diesem Sachverhalt trägt die Enumeration ***DateOrder*** Rechnung, wobei jeweils eine Ausprägung dieser Enumeration in den möglichen *CsvFormat*-Werten integriert ist. Listing 24 bzw. Listing 25 in Anhang A3[132] zeigen die Sourcecodes dieser Enumerationen.

4.5 Anwendung und Auswertung

Da es sich beim hier vorgestellten System nicht um eine Standalone-Lösung handelt, müssen die einzelnen Komponenten, den fachlichen Anforderungen des Anwenders entsprechend, miteinander kombiniert werden. Begonnen wird mit der Definition der Datenquelle und dem darauf basierenden *BaseValue*. Listing 7 zeigt, wie Index- und Short-Index-Werte aus je einer übergebenen Datenquelle gelesen und in einen *BaseValue* integriert werden.

Listing 7: Definition eines Basiswerts

```
1.  /* Read long values from file */
2.  ValueDateTupel[] baseValues = DataSource.getDataFromCsv(longFileName,
3.      CsvFormat.EU);
4.  /* Read short values from file */
5.  ValueDateTupel[] shortIndexValues = DataSource.getDataFromCsv(shortFileName,
6.      CsvFormat.EU);
7.
```

131 Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

132 Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

```
8.  /* Align and reassign the rows. */
9.  ValueDateTupel[][] aligned = ValueDateTupel
10.     .alignDates(new ValueDateTupel[][] { baseValues, shortIndexValues });
11. baseValues = aligned[0];
12. shortIndexValues = aligned[1];
13.
14. /* Construct the base value */
15. BaseValue baseValue = new BaseValue(baseValueName, baseValues, shortIndexValues);
```

Unter Verwendung dieses Basiswertes werden die Regeln konstruiert und kombiniert. Listing 8 zeigt, wie eine Regel sowohl ohne als auch mit Variationen instanziiert wird.

Listing 8: Instanziieren von Regeln

```
1.  ewmacLong = new EWMAC(baseValue, null,
2.    START_OF_REFERENCE_WINDOW, END_OF_REFERENCE_WINDOW, 32, 8, BASE_SCALE);
3.
4.  ewmacTop = new EWMAC(baseValue, ewmacVariations,
5.    START_OF_REFERENCE_WINDOW, END_OF_REFERENCE_WINDOW, 0, 0, BASE_SCALE);
```

Basiswert und Regeln werden an eine Instanz von *SubSystem* übergeben. Weiterhin wird auch das Kapital übergeben, dass in diesem *SubSystem*, also für diesen einen Basiswert, maximal eingesetzt werden soll. Die Aufteilung dieses Kapitals wird durch das Tradingsystem nicht abgedeckt und muss gem. Tabelle 7, einer Berechnung gem. Formel 3, oder einer vom Anwender selbst zu wählenden Regelung verteilt werden. Listing 9 zeigt, wie das *SubSystem* instanziiert wird. Es zeigt außerdem, wie von einer Instanz von *SubSystem* der Backtest ausgeführt wird.

Listing 9: Instanziieren von SubSystem

```
1.  SubSystem subSystem = new SubSystem(baseValue, rules, capital,
2.    BASE_SCALE);
3.  subSystem.backtest(START_OF_TEST_WINDOW, END_OF_TEST_WINDOW);
```

Unter Verwendung der ermittelten Gewichtungen für das Beispielportfolio (vgl. Tabelle 9) kann für die definierten Parameter die Performance der einzelnen Instanzen von *SubSystem*, sowie auch für das gesamte Portfolio ermittelt werden. Als Gesamtkapital wird eine Summe von 100.000,00 € definiert, wobei die gewählte Summe die Performance (außer bei sehr kleinen Summen aufgrund von Rundungen) nur geringfügig beeinflusst. Sie dient eher dazu, dem Anleger einen

Referenzpunkt in absoluten Zahlen zu geben. Listing 10 zeigt einen beispielhaften Print, der aus den Ausgaben der einzelnen *SubSystem* aufbereitet wurde.

Listing 10: Prüfung der Performance

```
1.  Starting backtest for BaseValue DAX with testing window 2019-01-02T22:00 - 2019-12-
    30T22:00
2.  Done Testing. Value after backtest: 19.240,42 €
3.  With your starting capital of 20.020,00 € that's a net return of -3,894%.
4.  Current position: Long, 16.937664946468317
5.
6.  Starting backtest for BaseValue EURO STOXX 50 with testing window 2019-01-
    02T22:00 - 2019-12-30T22:00
7.  Done Testing. Value after backtest: 41.319,86 €
8.  With your starting capital of 40.700,00 € that's a net return of 1,523%.
9.  Current position: Long, 16.76933238268308
10.
11. Starting backtest for BaseValue S&P 500 with testing window 2019-01-02T22:00 - 2019-12-
    30T22:00
12. Done Testing. Value after backtest: 37.673,30 €
13. With your starting capital of 39.280,00 € that's a net return of -4,09%.
14. Current position: Long, 18.018124430237826
15.
16. Done Testing. Value after backtest: 98.233,59 €
17. With your starting capital of 100.000,00 € that's a net return of -1,766%.
18.
19. Runtime: 2.329 seconds.
```

Aus Listing 10 wird ersichtlich, dass nur das *SubSystem* des Basiswerts EURO STOXX 50 im Betrachtungszeitraum eine positive Rendite erwirtschaften konnte. Allerdings wird ebenfalls ersichtlich, dass trotz der relativ hohen negativen Renditen der verbleibenden Subsysteme von –3,894% bzw. –4,09% insgesamt nur ein Verlust von 1,766% eingefahren wurde. Hier macht sich der Vorteil der Diversifikation eines Portfolios bemerkbar, welche das Risiko streut. Ein weiterer Einflussfaktor hierbei: Der „gewinnende“ Basiswert hatte die größte Gewichtung, sodass seine Rendite stärker ins Gewicht fällt als die der verlierenden Werte. Weiterhin lässt sich aus den Instanzen der Subsysteme auch die zuletzt gehaltene Position ermitteln. Zum letzten Zeitpunkt im Testing-Zeitraum wird bei allen drei Basiswerten mit einem Wachstum gerechnet.

Die Performance-Messungen gem. Listing 10 (und Listing 11) wurden auf einem *Lenovo ThinkPad L580* mit 4 *Intel® Core™ i7-8550U @ 1,80GHz* mit 16 GB Hauptspeicher durchgeführt. Getestet wurde die von Maven gebaute *.jar*-Datei. Dabei waren vor dem Start der Anwendung 1% CPU- und 30% Arbeitsspeicherauslastung als Grundlast vorhanden. Verwendet wurde eine *Java HotSpot™ 64-*

Bit JDK 11.0.3+12-LTS auf einem *Microsoft Windows 10 Pro 10.0.18362 Build 18362* mit aktuellen Treibern und Windows-Updates.

5 Kritische Betrachtung

Die vorliegende Untersuchung verfolgte das Ziel, der Versicherungsbranche im Bereich der Kapitalanlage zu mehr Selbständigkeit zu verhelfen. Unter der Verwendung moderner Technologien sollte es dem anwendenden Versicherer möglich sein, algorithmenbasiertes Trading über seine eigenen Systeme durchzuführen. Zwar ist es mit der in dieser Arbeit entwickelten Programmbibliothek nicht möglich, den gesamten Tradingprozess zentral abzubilden. Dennoch kann mit dieser Bibliothek eine wichtige Säule des Tradings abgedeckt werden: Die Bestimmung des Vertrauens in einen bestimmten Basiswert, basierend auf durch die Anwenderin oder den Anwender entwickelte Regeln.

Hierfür wurde der Ansatz gewählt, ein theoretisch beschriebenes Tradingsystem abstrakt programmatisch umzusetzen. Dieses kann entsprechend der Bedarfe des Anwenders angepasst und von diesem um die einzusetzenden Regeln erweitert werden (bzw. muss, wenn der Anwender nicht die beispielhaft entwickelten Regeln verwendet). Dabei wurden diejenigen Komponenten des Systems übernommen, die für die Versicherungsbranche relevant sind bzw. deren Auslassung revisionstechnische Mängel aufkommen ließen.

Das vorliegende System kann die gestellten Anforderungen größtenteils erfüllen. Unter Anwendung der Programmbibliothek können für einen bestimmten Basiswert, basierend auf von der Anwenderin oder dem Anwender zu gestaltenden Regeln, Vorhersagen über die künftige Entwicklung eines Finanzproduktes getroffen werden, wobei die Art des Finanzproduktes unerheblich ist. Weiterhin ist es dem Anwender möglich, eine von ihm entwickelte Regel (oder ein ganzes System von Regeln) gegen die Vergangenheit zu testen und die historische Performance dieser zu ermitteln. Hierdurch kann dieser das Vertrauen in das System stärken und seine eigens entwickelten Regeln validieren.

Eine hohe Testabdeckung eines Systems bringt großes Vertrauen sowohl in den bereits entwickelten als auch in den zukünftig noch zu entwickelnden Code. Durch frühzeitiges Testen einer Anwendung werden sowohl die Fehlerzahl an

sich, als auch die Auswirkung eines einzelnen Fehlers gering gehalten.[133] Durch die hohe Unit-Testabdeckung des Systems, wobei die Test-Suite für die hier entwickelten Komponenten über die Angaben im *Project Object Model (POM)*[134] bei jedem Bauen des Projektes ausgeführt wird, entsteht hohes Vertrauen in die Weiterentwicklung der hier dargestellten Sourcen. Weiterhin werden so künftige Refactoring-Maßnahmen z.B. zur Performance-Verbesserung in der Art unterstützt, als dass „auf Knopfdruck" die Funktionalität der Komponenten überprüft werden kann. Anhang A4[135] zeigt den Testcode.

Dennoch bringt die angewandte Methode auch Nachteile mit sich. Die Performance des von der Anwenderin oder dem Anwender selbst zusammenzustellenden Portfolios sollte mit entsprechender Skepsis bewertet werden, da eine solch händische Portfolioauswahl zumeist weniger diversifiziert und sub-optimal erfolgt. Falls möglich, sollten die mit diesem System erzielten Resultate mit anderen Out-Of-Sample-Methoden, wie z.B. dem Bootstrapping[136], verglichen werden.[137]

Eine der in dieser Arbeit beispielhaft entwickelten Regeln ist die Volatilitätsdifferenz, bei der der Abstand der aktuellen Volatilität vom historischen Mittelwert als Indikator für die Vorhersage dient. Die Regel, wie sie hier entwickelt wurde, entspricht nicht exakt denjenigen Beobachtungen, die Haugen und die auf ihm aufbauende Forschung tätigten. Sie stellten eher den Zusammenhang zwischen der Veränderung der Volatilität, quasi der Rendite gem. Formel 12 eines Volatilitätsindexes, und der Rendite des dazugehörigen Basiswertes fest. Tabelle 13 zeigt

---

133 Vgl. Beck und Andres, *Extreme Programming Explained*, S. 171.

134 Einen Überblick über die Funktion der POM-Datei gibt https://maven.apache.org/guides/introduction/introduction-to-the-pom.html.

135 Die Anhänge A3 und A4 werden in der digitalen Version des vorliegenden Bandes ausgespielt, siehe https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122.

136 Bootstrapping ist ein Fitting-Verfahren, bei welchem Optimierungen über einen großen Datenzeitraum wiederholt durchgeführt werden, sodass eine große Menge unterschiedlicher Marktgegebenheiten in die Optimierung einfließen. Der Grundgedanke dabei bleibt, dass die Vergangenheit ein guter Indikator für die Zukunft ist, jedoch ist nicht bekannt, welcher Teil der Vergangenheit wiederholt wird. Vgl. Carver, *Systematic Trading*, S. 75.

137 Vgl. Carver, *Systematic Trading*, S. 85.

die Korrelationen zwischen den Renditen der Basiswerte und den Renditen ihres jeweiligen Volatilitätsindexes.

Tabelle 13: Korrelationen der Renditen der Basiswerten mit den Renditen ihrer Volatilitätsindizes

Basiswert/ Volatilitätsindex	Korrelation der Renditen 2014-2018	Korrelation der Renditen 2019
DAX/VDAX-NEW®	$r = -0{,}74965827$	$r = -0{,}82796014$
S&P/VIX®	$r = -0{,}812628599$	$r = -0{,}849983014$
STOXX/VSTOXX®	$r = -0{,}741911611$	$r = -0{,}838134003$

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[138]

Tabelle 13 zeigt, dass die Veränderung der Volatilität stark mit einer gegensätzlichen Änderung des Basiswertes einhergeht. Eine auf diese beobachtete Gesetzmäßigkeit ausgerichtete Volatilitätsdifferenz wurde im Rahmen dieser Arbeit nicht genutzt. Die beispielhaft entwickelten Regeln sollen jedoch nur die Funktionsweise des ansonsten abstrakten Tradingsystems verdeutlichen, weshalb sich die Abweichung der Interpretation der Forschung Haugens nicht negativ auf die Aussagekraft dieser Arbeit auswirkt.

Die in dieser Arbeit entwickelte Volatilitätsdifferenz-Regel lebt davon, dass sie die Schwankungen eines Basiswertes erfasst und in einen Forecast übersetzt. Es ist eine der Voraussetzungen des von Carver entwickelten Systems, dass Vorhersagen, die aus einer Regel entwickelt werden, um die Volatilität bereinigt werden. Dies wird als Voraussetzung bzw. Prozessschritt in Kapitel 4.2.8 festgehalten. Dabei strebt Carver eine Wiederverwendbarkeit und insbesondere auch eine Vergleichbarkeit von Regeln über Basiswerte hinweg an. Die Bereinigung findet daher lediglich um die Standardabweichung des Basiswertes statt, also die

[138] Datenquelle: ARIVA, „DAX 30 Historische Kurse“; onvista, „VDAX-NEW Index“; ARIVA, „S&P 500 Historische Kurse“; Cboe, „VIX Index Historical Data“; ARIVA, „Euro Stoxx 50 Historische Kurse“; onvista, „VSTOXX VOLATILITÄTS Index“.

Abweichung der absoluten Werte voneinander. Die Volatilitätsdifferenz fußt jedoch auf der Volatilität der Renditen – ein relativer Wert, unabhängig von der Höhe des Basiswertes, sondern lediglich von seiner Veränderung.[139] Somit annulliert die Bereinigung um die Volatilität nicht die Aussagekraft der Volatilitätsdifferenz.

Bei der Auswahl der Basiswerte bzw. der Berechnung der Gewichtungen, mit denen die Basiswerte für einen Testzeitraum verwendet werden sollen, wird auf die Korrelation der Basiswerte als Maßgabe zurückgegriffen. Vorgeschlagen wird hierbei, für einen Testzeitraum von einem Jahr (hier: 2019) einen Fünfjahreszeitraum (hier: 2014–2018) für das Fitting zu verwenden. D.h., die Korrelationen der Jahre 2014–2018 werden dazu genutzt, die Gewichtungen für einen Test im Jahre 2019 zu bestimmen. Die dem Backtest zugrundeliegenden Korrelationen sind in Tabelle 8 notiert.

In einer Ex-post-Betrachtung lässt sich feststellen, dass die Korrelationen des Fitting-Zeitraums nicht zwingend einen guten Indikator für die Korrelationen des Test-Zeitraums darstellen. Bei einer der Kombinationen (STOXX/S&P) weichen Fitting- und Testing-Zeitraum-Korrelationen sogar um knapp 50% ab. Die übrigen beiden Kombinationen weichen um 16,2% (DAX/STOXX) bzw. 11,4% (DAX/S&P) ab. Tabelle 14 zeigt die Korrelationen der drei Basiswerte im zeitlichen Vergleich. Dabei wird auch ersichtlich, dass selbst eine stärkere Betrachtung des jüngsten Wertes, also die Korrelationen von 2018, nur in einer der Kombinationen zu einer Verbesserung der Vorhersagen geführt hätten, während die beiden übrigen Kombinationen noch stärker abgewichen wären.

[139] Vgl. Carver, *Systematic Trading*, S. 112.

Tabelle 14: Ex-Post-Betrachtung der Korrelationen des Beispielportfolios im zeitlichen Vergleich

Korrelationen	DAX/STOXX	DAX/S&P	STOXX/S&P
2014	$r = 0{,}889224501$	$r = 0{,}308755904$	$r = 0{,}373999552$
2015	$r = 0{,}983682278$	$r = 0{,}789387639$	$r = 0{,}787803603$
2016	$r = 0{,}822409639$	$r = 0{,}856403449$	$r = 0{,}51219828$
2017	$r = 0{,}933843371$	$r = 0{,}861089748$	$r = 0{,}711953686$
2018	$r = 0{,}989206697$	$r = 0{,}405135941$	$r = 0{,}385056132$
01.01.2014 – 31.12.2018	$r = 0{,}820458371$	$r = 0{,}844285275$	$r = 0{,}497945338$
2019	$r = 0{,}979092078$	$r = 0{,}953202826$	$r = 0{,}976981128$

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[140]

Die Anlegerin bzw. der Anleger sollte diese Unsicherheiten in die Risikobetrachtungen stets einbeziehen. Die Unsicherheit der tatsächlichen Entwicklungen der Basiswerte könnte zu einer starken Veränderung der künftigen Kursverläufe führen, sodass selbst in einem kurzen Fitting-Zeitraum formulierte Annahmen über die Gewichtungen in einen direkt anschließenden Testing-Zeitraum nicht mehr angemessen sind.

Fitting- und Testing-Zeitraum fielen auf Zeiten vergleichsweise „ruhiger Gewässer“ im Finanzsektor. Die Börsen legten tendenziell einen Aufwärtstrend an den Tag, wobei auch innerhalb der jeweiligen Perioden nur geringe Schwankungen wahrzunehmen waren. In dieser Arbeit konnte daher explizit keine Aussage darüber getroffen werden, wie sich die beispielhaft entwickelten Regeln in Extremsituationen verhalten würden. Unter Verwendung des Beispielportfolios ließ sich für einen Testing-Zeitraum 01.01.2020–31.03.2020, wobei die Gewichtungen über einen Zeitraum von 2015–2019 gefittet wurden, eine Rendite von 15,542%

[140] Datenquelle: ARIVA, „DAX 30 Historische Kurse“; ARIVA, „Euro Stoxx 50 Historische Kurse“; ARIVA, „S&P 500 Historische Kurse“.

erwirtschaften. Unter dem Betrachtungspunkt, dass Aktienindizes bereits ein diversifiziertes Portfolio darstellen, kann angenommen werden, dass sich durch die gezielte Anlage in Derivate einzelner Basiswerte noch höhere Erträge hätten erwirtschaften lassen. Ein solcher Extremfall ist jedoch nur schwerlich mit „regulären“ Bedingungen vergleichbar, sodass Betrachtungen, die einen solchen Zeitraum beinhalten, genauerer Analyse unterzogen werden sollten.

Unter Verwendung des hier vorgestellten Systems können nicht alle in Kapitel 2.3 genannten psychologischen Faktoren angegangen werden. Insbesondere denjenigen Verzerrungen, die im operativen Tätigkeitsbereich auftreten, wie Kaufs- bzw. Verkaufsentscheidungen einzelner Basiswerte basierend auf deren historischer Entwicklung, kann über ein objektiv und reproduzierbar agierendes System wie das hier gezeigte effektiv begegnet werden. Bei strategischen Entscheidungen, wie der Auswahl des Portfolios oder der Definition der Fitting- und Testing-Zeiträume, kann das System zwar indirekt Hilfestellung geben, die tatsächlichen Entscheidungen müssen dennoch extern getroffen werden. So können mit diesem System Verzerrungen wie der Home-Bias nicht überwunden werden.

Der Umfang an Fitting, der unter Zuhilfenahme der hier entwickelten Bibliothek durchgeführt werden kann, fällt gering aus. Es wird insbesondere kein Fitting der Regeln an sich anhand von Performance- oder ähnlichen Parametern vorgenommen. Ein Wert, der im Rahmen dieser Arbeit gefittet wurde, sind die Gewichtungen der Basiswerte anhand ihrer Korrelationen im Fitting-Zeitraum. In einer dem *SubSystem* übergeordneten Komponente, die mehrere Instanzen von *SubSystem* aufnimmt und anhand der Korrelationen entweder ihrer Basiswerte oder ihrer kombinierten Forecasts Gewichtungen für diese bestimmt, könnte ein automatisiertes Fitting durchgeführt werden. Somit entfiele das händische Kalkulieren der Gewichtungen, wie es bei der hier entwickelten Lösung durchgeführt werden muss. Eine weitere Möglichkeit, das Fitting auszubauen, liegt im Design der Regeln. Insbesondere Regeln wie die Volatilitätsdifferenz, die Parameter mit deren historischen Mitteln vergleichen, könnten eine Nutzung der bereits an die Komponente *Rule* übergebenen Daten für den Fitting-Zeitraum in der Art nutzen, dass das für den Vergleich benötigte historische Mittel lediglich aus dem Fitting-Zeitraum bestimmt wird.

Mit einer Laufzeit von ca. 2,5 Sekunden (vgl. Listing 10) für das hier gezeigte beispielhafte Portfolio mit den ebenfalls für diese Arbeit erstellten Regeln erreicht das Tradingsystem eine akzeptable Performance. Für den Bereich des Hochfrequenzhandels, wobei Transaktionen in Sekundenbruchteilen ausgelöst werden, ist es damit sicherlich nicht geeignet. Dies war jedoch nicht das Ziel. Die zeitliche Performance könnte unter Verwendung eines dedizierten Servers, auf welchem die Anwendung eigenständig betrieben wird, u.U. verbessert werden. Weiterhin könnte durch die Serialisierung der Komponenten, sowie durch die „Erweiterbarkeit“ durch das Hinzufügen von Daten anstatt der Neuberechnung bei jedem neuen Zeitpunkt die Performance ebenfalls verbessert werden. So müssten beim Bekanntwerden eines neuen Datenpunktes lediglich die Vorhersagen für den Zeitpunkt dieses Datenpunktes neu berechnet werden, wobei die restlichen, bereits ermittelten Forecasts, nicht erneut kalkuliert würden.[141]

Ein einzelner Basiswert könnte über das System sicherlich auch schneller als nur einmal täglich gehandelt werden. Mit einer Laufzeit von ca. 1 Sekunde für einen Basiswert unter vergleichbaren Bedingungen wie Listing 10 könnte auch untertägig Handel betrieben werden. Listing 11 zeigt die Performance unter der Verwendung eines einzelnen Basiswertes. Unter Anwendung der oben beschriebenen Performanceoptimierungen könnte auch hier die Laufzeit verbessert werden.

Listing 11: Performance und Laufzeit eines einzelnen Basiswertes

```
1.  Starting backtest for BaseValue EURO STOXX 50 with testing window 2019-01-
    02T22:00 - 2019-12-30T22:00
2.  Done Testing. Value after backtest: 101.523,03 €
3.  With your starting capital of 100.000,00 € that's a net return of 1,523%.
4.  Current position: Long, 16.76933238268308
5.
6.  Runtime: 0.964 seconds.
```

[141] Die Erweiterbarkeit bringt keinen Performancevorteil, wenn ein In-Sample-Backtest-Verfahren angewandt wird und sich der neue Datenpunkt im Fitting-Zeitraum befindet.

6 Fazit und Ausblick

Die Versicherungsbranche muss ihre Kapitalanlage wieder in die eigene Hand nehmen, um nachhaltig mit ihrem Geld wirtschaften zu können. Dafür wird eine Lösung benötigt, über die die Versicherungsbranche objektive Investment- und Tradingentscheidungen treffen kann, wobei die Lösung revisionssicher und wiederholgenau arbeiten muss.

Die hier entwickelte Lösung schafft es, die in den vorigen Kapiteln formulierten Anforderungen an ein solches System zu erfüllen. Durch die Umsetzung als computergesteuerten Algorithmus basieren die getätigten Vorhersagen ausschließlich auf den vom Betreiber zur Verfügung gestellten Rohdaten. Freilich muss hier seitens des Anwenders entsprechende Qualitätssicherung der Eingaben getätigt werden.

Die in Kapitel 1.2 gestellten Fragen wurden im Verlauf dieser Arbeit beantwortet. Über algorithmenbasiertes Trading können Anleger computergesteuert Börsenorder auslösen und so ihre Kapitalanlage durch automatisierte Systeme verwalten. Insbesondere neue Technologien wie die der KI oder Big Data eröffnen ungeahnte Möglichkeiten in diesem Bereich. Die Versicherungsbranche ist durch ihre Reglementierung stark eingeschränkt, doch auch sie kann unter bestimmten Voraussetzungen vollautomatisierten Kapitalhandel betreiben. Dabei müssen die einzugehenden (Kapital-)Risiken definierbar und über die Zeit hinweg anpassbar sein. Weiterhin muss der Eingriff durch menschliche Benutzer auf einem kontrollierten Minimum gehalten werden. Das hier entwickelte System ermöglicht der Versicherungsbranche, diese Voraussetzungen zu erfüllen. Die Grundsteine für den vollautomatisierten Handel sind gelegt.

Dennoch stellt die hier entwickelte Bibliothek keinen Anspruch auf Vollständigkeit dar. Durch die Veröffentlichung des Codes auf GitHub[142] kann dieser durch Anwenderinnen und Anwender verbessert werden. Open Source-Projekte leben vom Wandel und von der Erweiterung, sodass die Gemeinschaft sicherlich nur davon profitiert, wenn verschiedene Anwender ihre Erfahrungen und Ideen in die Bibliothek einbringen und diese weiterentwickeln.

[142] Rumford, „Tradingsystem-Repository“.

Das Fitting nimmt eine entscheidende Rolle im Backtest ein. Nur solche Systeme, die in der Fitting-Phase korrekt skaliert und auf die zu handelnden Marktgegebenheiten eingestellt werden, können realistische Ergebnisse in der Testing-Phase erzielen. In weiterführender Arbeit, ob wissenschaftlich oder ökonomisch motiviert, sollten die Fitting-Fähigkeiten der hier entwickelten Bibliothek erweitert werden. So könnte, wie in Kapitel 5 beschrieben, eine Komponente entwickelt werden, die sich strukturell über die Komponente *SubSystem* stülpt und mehrere Instanzen dieser Komponente beherbergt. Diese Komponente repräsentiert dann das gesamte oder ein Teilportfolio, wobei es auch die Gewichtungen der einzelnen *SubSysteme* bestimmt. Vorgeschlagen werden hierbei risikobereinigte Parameter, wie z.B. der Sharpe-Quotient[143], wobei in einer Ex-post-Betrachtung (hier: des Fitting-Zeitraums) die erwirtschafteten Renditen um das eingegangene Risiko bereinigt werden.

Generell erlaubt das Nutzen technischer Tradingsysteme einen einfachen Einstieg in die Diversifikation der Kapitalanlage.[144] Einen Aspekt der Anlage, den diese Bibliothek jedoch nicht automatisiert, ist eben diese Auswahl der zu handelnden Basiswerte. Über den Einsatz algorithmengesteuerter Auswahlprozesse kann dem in dieser Arbeit nicht betrachteten menschlichen Bias wirkungsvoll begegnet werden. Durch die Verwendung eines Machine-Learning-Algorithmus, der die Historie eines Finanzproduktes (bzw. des Unternehmens, welches durch das Produkt repräsentiert wird) und aktuelle Geschehnisse interpretiert, können den Marktwert betreffende Faktoren ausfindig gemacht und in die Investment-Entscheidungen einbezogen werden. Das diversifiziert zudem das Portfolio weiter, sodass das mit der Anlage eingegangene Risiko weiter gestreut wird. Beispiele können in der Branche der Robo-Advisor gesehen werden. Solche algorithmengesteuerten Portfolio-Berater stellen regelmäßig diversifiziertere Portfolios zusammen als menschliche Berater.[145]

Weitere Entwicklungsmöglichkeiten bieten auch neue Technologien wie die der KI. Über einen Data First-Ansatz könnten über einen entsprechenden Algorithmus neue Ideen über den Finanzmarkt gewonnen werden, welche sich wiederum

---

143 Vgl. Sharpe, „The Sharpe Ratio“.
144 Vgl. Kaufman, *A Short Course in Technical Trading*, S. 8.
145 Vgl. D’Acunto, Prabhala, und Rossi, „The Promises and Pitfalls of Robo-Advising“, S. 16 f.

in quantifizierbare Regeln übersetzen lassen. Eingebettet in einen kontinuierlichen Optimierungsprozess, der entweder dauerhaft neue Regeln vorschlägt oder entwickelte Ideen über die jeweils neuesten Daten prüft (z.B. am Jahresende) wird das System aktuell gehalten und repräsentiert stets den Stand und das Verhalten des Marktes. Über solche Prozesse ist es auch möglich, die bestehenden Regeln anzupassen, sie also einem anhaltenden Fitting zu unterziehen.

Literatur

Aigner, Tobias, und Francesco D'Acunto. „Robo-Advisor-Studie". Money, Markes & Machines, o. J. https://soundcloud.com/scalablecapital/robo-advisor-studie-english-only [Zugegriffen am 26.06.2020].

ARIVA. „DAX 30 Historische Kurse", o. J. https://www.ariva.de/dax-index/historische_kurse [Zugegriffen am 18.12.2020].

———. „Euro Stoxx 50 Historische Kurse", o. J. https://www.ariva.de/eurostoxx-50-index/historische_kurse [Zugegriffen am 18.12.2020].

———. „S&P 500 Historische Kurse", o. J. https://www.ariva.de/s-p_500-index/historische_kurse [Zugegriffen am 18.12.2020].

Ausschuss Enterprise Risk Management. „Emerging Risks 2020". Ergebnisbericht. Köln: Deutsche Aktuarvereinigung e.V., 28. Januar 2020. https://aktuar.de/unsere-themen/fachgrundsaetze-oeffentlich/2020-01-28_Ergebnisbericht_Emerging_Risks_Update_2020.pdf [Zugegriffen am 26.06.2020].

Baker, Nardin L., und Robert A. Haugen. „Low Risk Stocks Outperform within All Observable Markets of the World". *SSRN Electronic Journal*, April 2012. https://doi.org/10.2139/ssrn.2055431.

Beck, Kent, und Cynthia Andres. *Extreme Programming Explained: Embrace Change*. 2nd ed. Boston, MA: Addison-Wesley, 2005.

Becker, Matt. „Diversification: The Only Free Lunch in Investing". *The Simple Dollar* (blog), 13. Dezember 2015. https://www.thesimpledollar.com/investing/blog/limit-investment-risk-without-limiting-returns/ [Zugegriffen am 26.06.2020].

Bitkom e.V. und Deutsches Forschungszentrum für Künstliche Intelligenz GmbH. „Künstliche Intelligenz: Wirtschaftliche Bedeutung, gesellschaftliche Herausforderungen, menschliche Verantwortung". Berlin, Kaiserslautern, 2017. https://www.bitkom.org/sites/default/files/file/import/171012-KI-Gipfelpapier-online.pdf [Zugegriffen am 26.06.2020].

Brewer, Eric, Lawrence Ying, Lawrence Greenfield, Robert Cypher, und Theodore Ts'o. „Disks for Data Centers". White paper. White Papers for FAST 2016. Google, Inc., 29. Februar 2016. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44830.pdf [Zugegriffen am 26.06.2020].

Bundesanstalt für Finanzdienstleistungsaufsicht. „Solvency II: Rechtsgrundlagen, Leitlinien und Auslegungsentscheidungen der BaFin", 7. Januar 2020. https://www.bafin.de/DE/Aufsicht/VersichererPensionsfonds/Aufsichtsregime/RechtsgrundlagenLeitlinien/rechtsgrundlagen_node.html [Zugegriffen am 26.06.2020].

———. „Versicherungsaufsicht". BaFin, 19. März 2016. https://www.bafin.de/DE/DieBaFin/AufgabenGeschichte/Versicherungsaufsicht/versicherungsaufsicht_node.html [Zugegriffen am 26.06.2020].

Carver, Robert. *Systematic Trading: A Unique New Method for Designing Trading and Investing Systems*. Petersfield, Hampshire, Vereinigtes Königreich: Harriman House Ltd., 2015.

Cboe. „VIX Index Historical Data", o. J. https://ww2.cboe.com/products/vix-index-volatility/vix-options-and-futures/vix-index/vix-historical-data [Zugegriffen am 18.12.2020].

CFA Institute. „EWMA (Exponentially Weighted Moving Average)". *WallStreetMojo* (blog), 15. August 2019. https://www.wallstreetmojo.com/ewma/ [Zugegriffen am 26.06.2020].

D'Acunto, Francesco, Nagpurnanand Prabhala, und Alberto Rossi. „The Promises and Pitfalls of Robo-Advising". *SSRN Electronic Journal*, 10. Oktober 2017. https://doi.org/10.2139/ssrn.3122577.

Deutsche Börse. „Preisverzeichnis für die Nutzung der Börsen-EDV der Frankfurter Wertpapierbörse und der EDV XONTRO", 1. Januar 2020. https://www.xetra.com/resource/blob/62002/aeea35c142ed3d79c5388b6f2bd52a87/data/20200101_preisverzeichnis-fuer-die-nutzung-der-boersen-edv-der-fwb-und-der-edv-xontro.pdf [Zugegriffen am 26.06.2020].

———. „Preliminary Results Q4 and FY 2008". Geschäftsbericht, 25. Februar 2009.

Deutsche Börse AG. „Leitfaden zu den Volatilitätsindizes der Deutschen Börse“. Leitfaden. Frankfurt am Main, Januar 2007. Aufgerufen über die WayBack Machine: https://web.archive.org/web/20091122151635/http:/www.boerse-frankfurt.de/DE/MediaLibrary/Document/Wissen/vdax_guide.pdf [Zugegriffen am 26.06.2020].

Finanzen.net. „Vapiano zahlungsunfähig - Aktie halbiert sich“. Blog. finanzen.net, 20. März 2020. https://www.finanzen.net/nachricht/aktien/umsatzrueckgang-vapiano-zahlungsunfaehig-aktie-halbiert-sich-8651990 [Zugegriffen am 26.06.2020].

Gesamtverband der Deutschen Versicherungswirtschaft e.V. „Fakten zur Versicherungswirtschaft“. Berlin, 24. September 2019. https://www.gdv.de/resource/blob/24006/c161f408c5bb9505e03440df468f1f53/keyfact-booklet---die-versicherungswirtschaft-fakten-im-ueberblick-download-data.pdf [Zugegriffen am 26.06.2020].

———. „Säule I: Kapitalanforderungen unter Solvency II“, 29. Juli 2015. https://www.gdv.de/de/themen/news/saeule-i--kapitalanforderungen-unter-solvency-ii-17224 [Zugegriffen am 26.06.2020].

———. „Säule II: Governance und Risikomanagement“, 12. Oktober 2015. https://www.gdv.de/de/themen/news/saeule-ii--governance-und-risikomanagement-17226 [Zugegriffen am 26.06.2020].

———. „Säule III: Berichtspflichten unter Solvency II“, 12. Oktober 2015. https://www.gdv.de/de/themen/news/saeule-iii--berichtspflichten-unter-solvency-ii-17228 [Zugegriffen am 26.06.2020].

Göhler, Wilhelm, und Barbara Ralle. *Formelsammlung höhere Mathematik*. Nachdr. der 16., Überarb. Aufl. Frankfurt am Main: Deutsch, 2007.

Gruppe Deutsche Börse. „Deutsche Börse stellt IT auf Cloud-Nutzung um“. Deutsche Börse stellt IT auf Cloud-Nutzung um, 14. Mai 2020. https://deutsche-boerse.com/dbg-de/produkte-services/insights-innovation-new-technologies-de/Deutsche-B-rse-stellt-IT-auf-Cloud-Nutzung-um-2000656 [Zugegriffen am 26.06.2020].

———. „Geschäftsbericht 2018“. Geschäftsbericht. Frankfurt am Main, 15. März 2019. https://deutsche-boerse.com/resource/blob/1443022/b6de6a2ebe7efd65fc6e9b882db2ecd1/data/GDB-Geschaeftsbericht-2018.pdf [Zugegriffen am 26.06.2020].

———. „ShortDAX® Indizes“, Oktober 2010. Aufgerufen über die WayBack Machine: https://web.archive.org/web/20160703143517/http://dax-indices.com/DE/MediaLibrary/Document/FS%20ShortDAX_d.pdf [Zugegriffen am 26.06.2020].

Gsell, Markus. „Technological Innovations in Securites Trading: The Adoption of Algorithmic Trading“. In *Proceedings of the 13th Pacific-Asia Conference on Information Systems (PACIS)*, 51–75. Hyderabad, Indien: ibidem, 2009.

Haugen, Robert A., und A. James Heins. „Risk and the Rate of Return on Financial Assets: Some Old Wine in New Bottles“. *The Journal of Financial and Quantitative Analysis* 10, Nr. 5 (Dezember 1975): 775–84. https://doi.org/10.2307/2330270.

Kaufman, Perry J. *A Short Course in Technical Trading*. New York: Wiley, 2003.

Kröger, Carolin. „Digitalisierung in der Versicherungsbranche: Diese 5 Technologien treiben den Wandel“. IT Finanzmagazin, 20. Februar 2018. https://www.it-finanzmagazin.de/digitalisierung-versicherungsbranche-5-technologien-66199/ [Zugegriffen am 26.06.2020].

Kruger, Justin, und David Dunning. „Unskilled and Unaware of It: How Difficulties in Recognizing One’s Own Incompetence Lead to Inflated Self-Assessments.“ *Journal of Personality and Social Psychology* 77, Nr. 6 (1999): 1121–34. https://doi.org/10.1037/0022-3514.77.6.1121.

Lewis, Michael. *Flash Boys: A Wall Street Revolt*. First Edition. New York: W.W. Norton & Company, 2014.

Markowitz, Harry. „Portfolio Selection“. *The Journal of Finance* 7, Nr. 1 (März 1952): 77–91. https://doi.org/10.2307/2975974.

Mittnik, Stefan. „Geldanlage für Angsthasen: Wer wenig wagt, gewinnt“. *FAZ.NET*, 2. August 2017. https://www.faz.net/1.5128359 [Zugegriffen am 26.06.2020].

Moreira, Alan, und Tyler Muir. „Should Long-Term Investors Time Volatility?“ *Journal of Financial Economics* 131, Nr. 3 (März 2019): 507–27. https://doi.org/10.1016/j.jfineco.2018.09.011.

Odean, Terrance. „Are Investors Reluctant to Realize Their Losses?“ *The Journal of Finance* LIII, Nr. 5 (Oktober 1998): 1775–98.

onvista. „VDAX-NEW Index: Kurs, Chart & News (A0DMX9 | DE000A0DMX99)“, o. J. https://www.onvista.de/index/VDAX-NEW-Index-12105789 [Zugegriffen am 18.12.2020].

———. „VSTOXX VOLATILITÄTS Index: Kurs, Chart & News (A0C3QF | DE000A0C3QF1)“, o. J. https://www.onvista.de/index/VSTOXX-VOLATILITAeTS-Index-12105800 [Zugegriffen am 18.12.2020].

Pulham, Susan, und Michael Deeken. *Zur Rationalität von Anlageentscheidungen: Begriffsklärung und Implikationen für Kapitalmarktakteure*. essentials. Wiesbaden: Springer Fachmedien Wiesbaden, 2015. https://doi.org/10.1007/978-3-658-10806-9.

Rumford, Max. „Blitzableiter/Tradingsystem“, 25. Juni 2020. https://github.com/Blitzableiter/Tradingsystem [Zugegriffen am 18.12.2020].

Schumacher, Hans-Georg, Markus Sobau, und Felix Hänsler. *Entgeltumwandlung*. Wiesbaden: Gabler Verlag, 2013. https://doi.org/10.1007/978-3-8349-4557-0.

Schweizerischer Versicherungsverband SVV. „Emerging Risks: Eine Wertung der Arbeitsgruppe Emerging Risks der Fachkommission Haftpflicht des SVV“. Zürich, Schweiz: Schweizerischer Versicherungsverband SVV, 9. November 2018. https://www.svv.ch/sites/default/files/2018-11/SVV_Brosch%C3%BCre_Emerging_Risks_2018_DE.pdf.

Sharpe, William F. „The Sharpe Ratio“. *The Journal of Portfolio Management* 21, Nr. 1 (31. Oktober 1994): 49–58. https://doi.org/10.3905/jpm.1994.409501.

Shefrin, Hersh, und Meir Statman. „The Disposition to Sell Winners Too Early and Ride Losers Too Long: Theory and Evidence“. *The Journal of Finance* 40, Nr. 3 (Juli 1985): 777–90. https://doi.org/10.1111/j.1540-6261.1985.tb05002.x.

Spiegel Online. „Erster Fall für den Protektor“. Spiegel Online, 26. Juni 2003. https://www.spiegel.de/wirtschaft/mannheimer-erster-fall-fuer-den-protektor-a-254769.html [Zugegriffen am 26.06.2020].

STOXX Ltd. „Guide to the DAX Strategy Indices“. Guide Strategy Indices, März 2020. https://www.dax-indices.com/document/News/2020/March/Guide%20to%20the%20DAX%20Strategy%20Indices_I_3_3_d_20200310.pdf [Zugegriffen am 26.06.2020].

Süddeutsche Zeitung. „Kurssturz an der Wall Street - Handel war unterbrochen“. Blog. Süddeutsche.de. Zugegriffen 20. März 2020. https://www.sueddeutsche.de/wirtschaft/wall-street-dow-jones-kurseinbruch-1.4837937 [Zugegriffen am 26.06.2020].

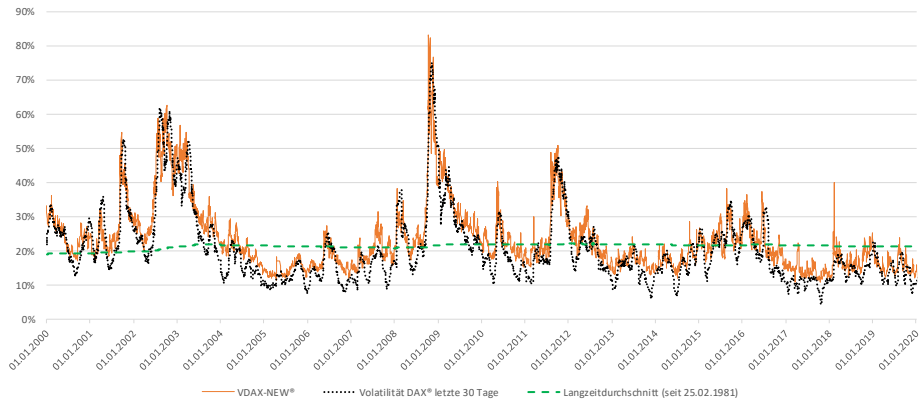
Taleb, Nassim Nicholas. *Fooled by Randomness: The Hidden Role of Chance in the Markets and in Life*. Second edition, Updated by the author. Random House Trade Paperbacks. New York: Random House Trade Paperbacks, 2005.

Anhang

A1 Bedeutende Indizes

DAX 30

Abbildung 9: VDAX-NEW® und selbsterrechnete 30-Tages-Volatilität des DAX 30

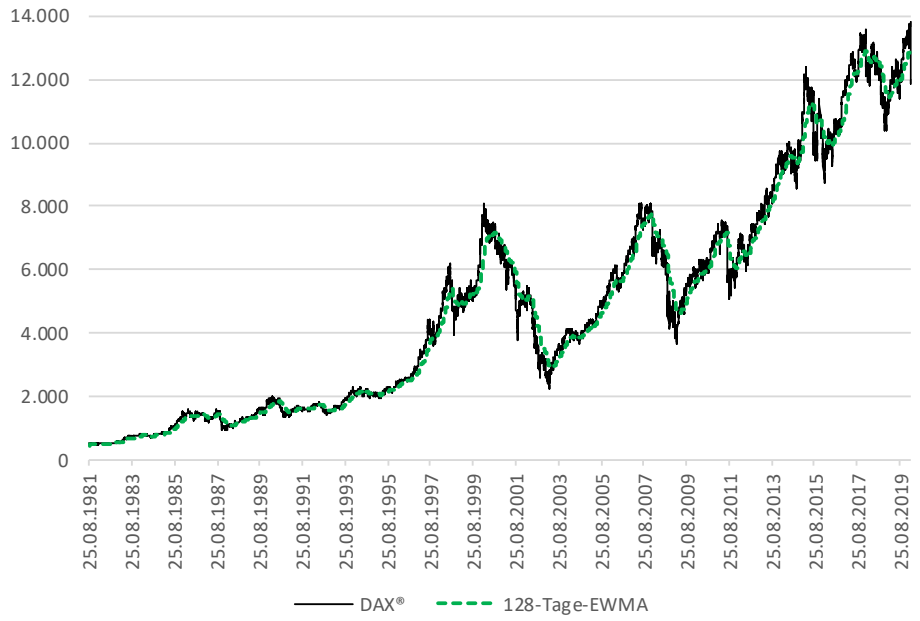


Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[146]

$$r = 0{,}909492848$$

[146] Datenquelle: ARIVA, „DAX 30 Historische Kurse“; onvista, „VDAX-NEW Index“.

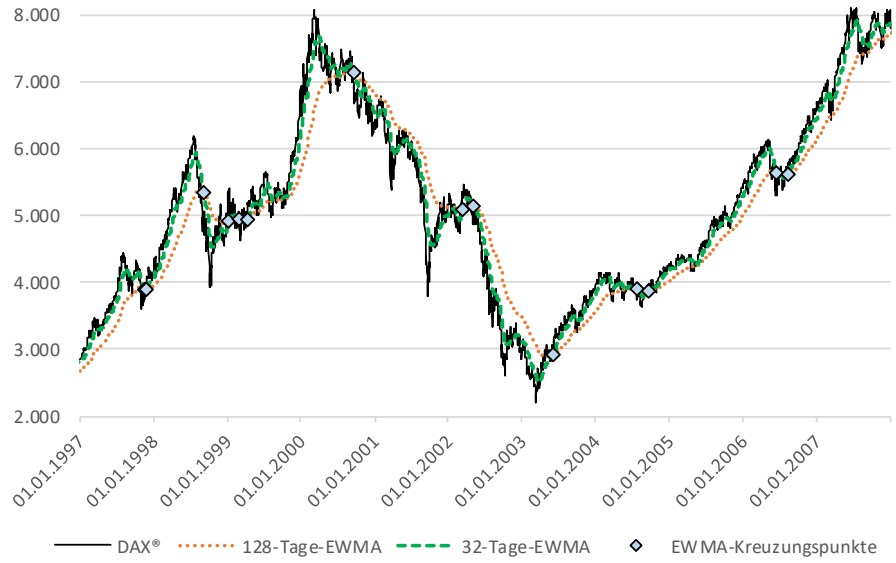
Abbildung 10: DAX und 128-Tages-EWMA des DAX



Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[147]

[147] Datenquelle: ARIVA, „DAX 30 Historische Kurse“.

Abbildung 11: DAX, 128-Tage-EWMA, 32-Tage-EWMA und EWMA-Kreuzungspunkte

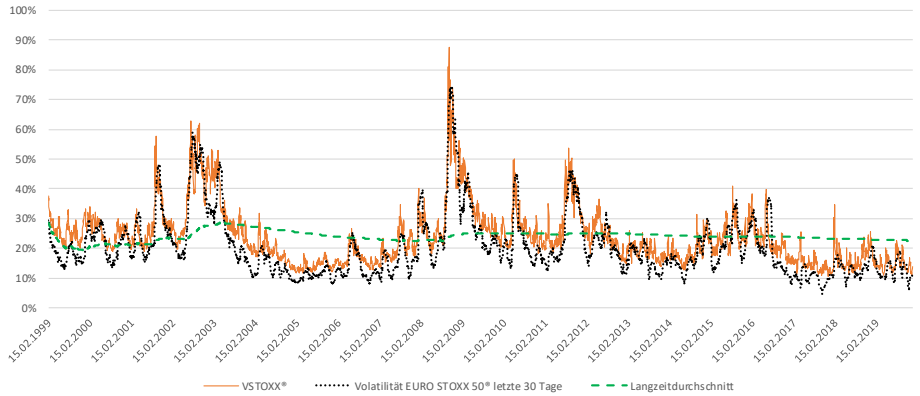


Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[148]

[148] Datenquelle: ARIVA, „DAX 30 Historische Kurse“.

EURO STOXX 50

Abbildung 12: VSTOXX® und selbsterrechnete 30-Tages-Volatilität des STOXX

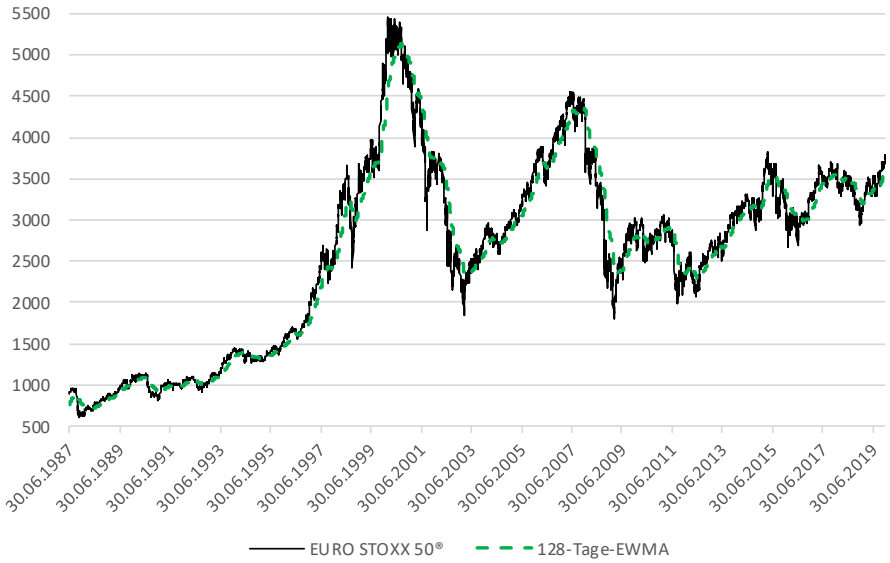


Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[149]

$$r = 0{,}8866639$$

[149] Datenquelle: ARIVA, „Euro Stoxx 50 Historische Kurse“; onvista, „VSTOXX VOLATILITÄTS Index“.

Abbildung 13: STOXX und 128-Tages-EWMA des STOXX



Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[150]

[150] Datenquelle: ARIVA, „Euro Stoxx 50 Historische Kurse“.

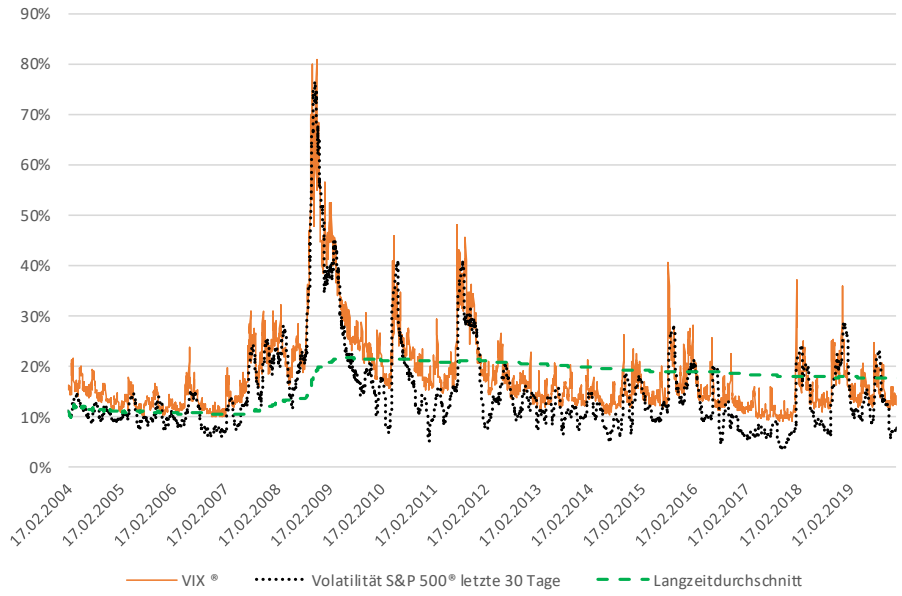
Abbildung 14: STOXX, 128-Tage-EWMA, 32-Tage-EWMA und EWMA-Kreuzungspunkte

Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[151]

[151] Datenquelle: ARIVA, „Euro Stoxx 50 Historische Kurse“.

S&P 500

Abbildung 15: VIX® und selbsterrechnete 30-Tages-Volatilität des S&P

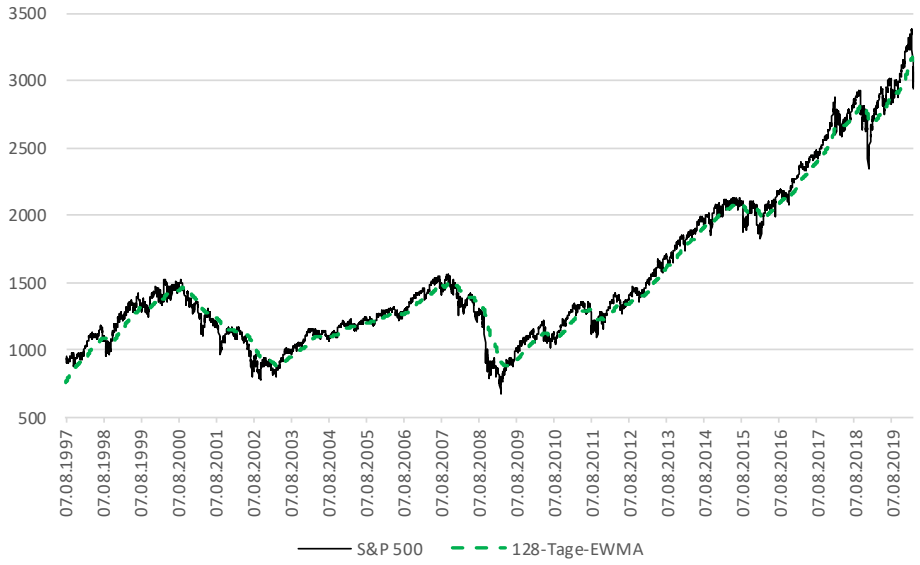


Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[152]

$$r = 0{,}9070979$$

[152] Datenquelle: ARIVA, „S&P 500 Historische Kurse“; Cboe, „VIX Index Historical Data“.

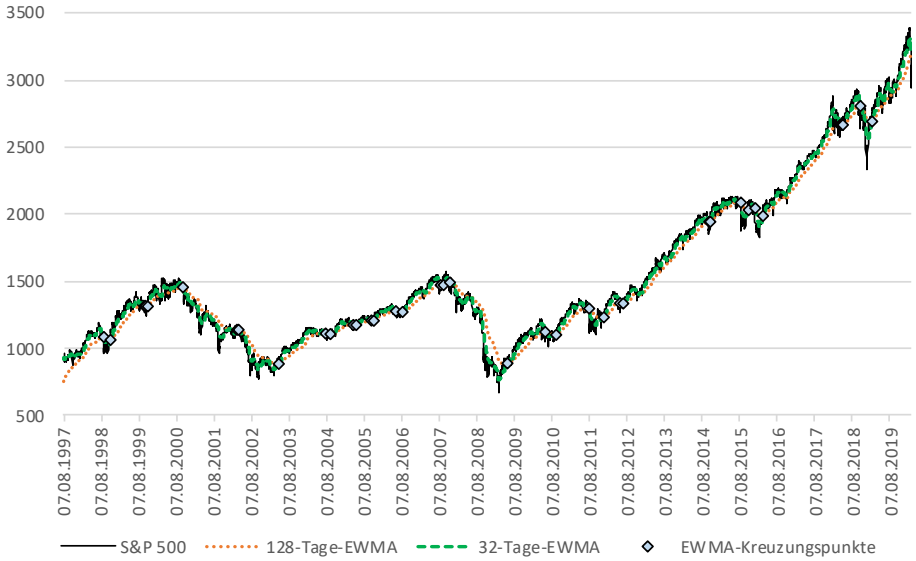
Abbildung 16: S&P und 128-Tages-EWMA des S&P



Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[153]

[153] Datenquelle: ARIVA, „S&P 500 Historische Kurse“.

Abbildung 17: S&P, 128-Tage-EWMA, 32-Tage-EWMA und EWMA-Kreuzungspunkte



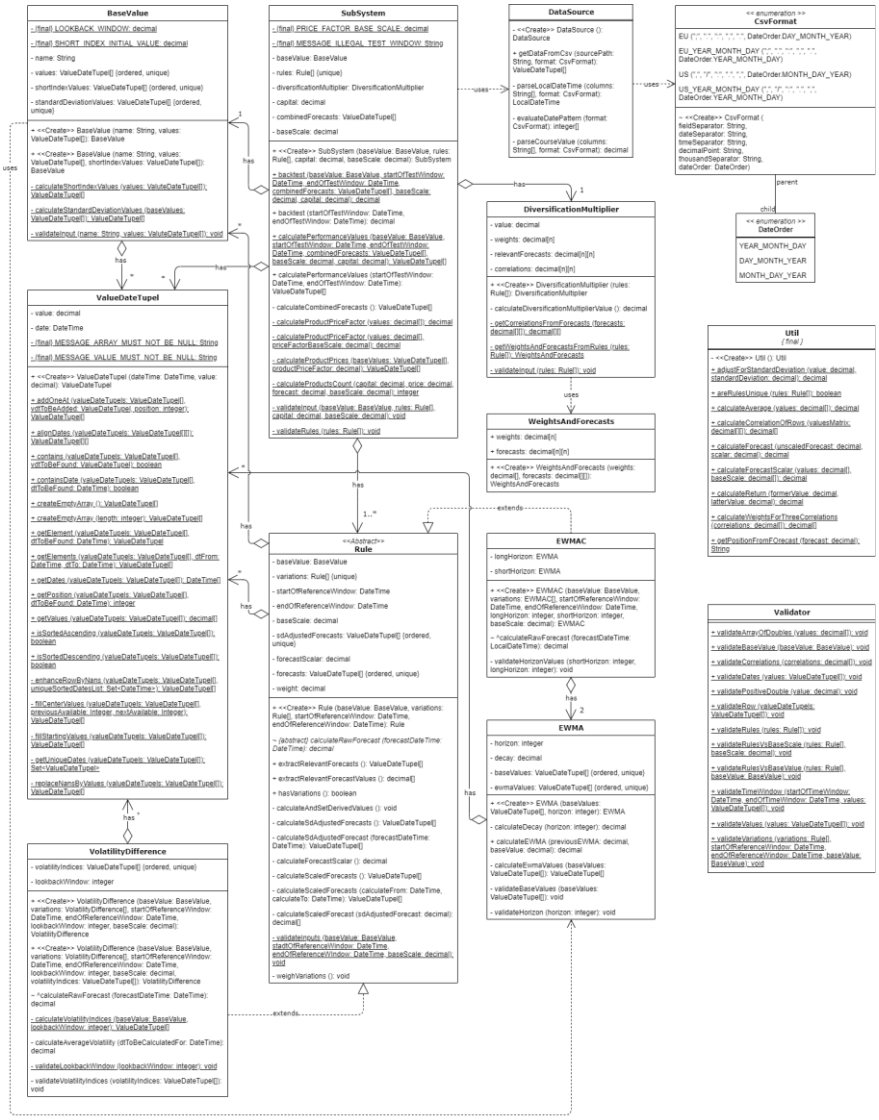
Quelle: Eigene Darstellung basierend auf Datenquellen, siehe Fußnote.[154]

[154] Datenquelle: ARIVA, „S&P 500 Historische Kurse“.

A2 Diagramme

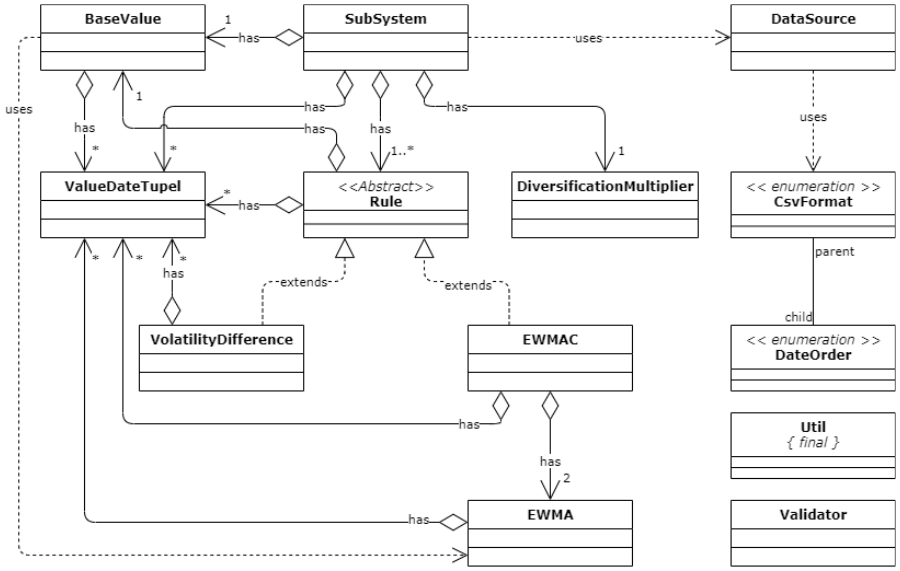
Gesamtklassendiagramm

Abbildung 18: Gesamtklassendiagramm



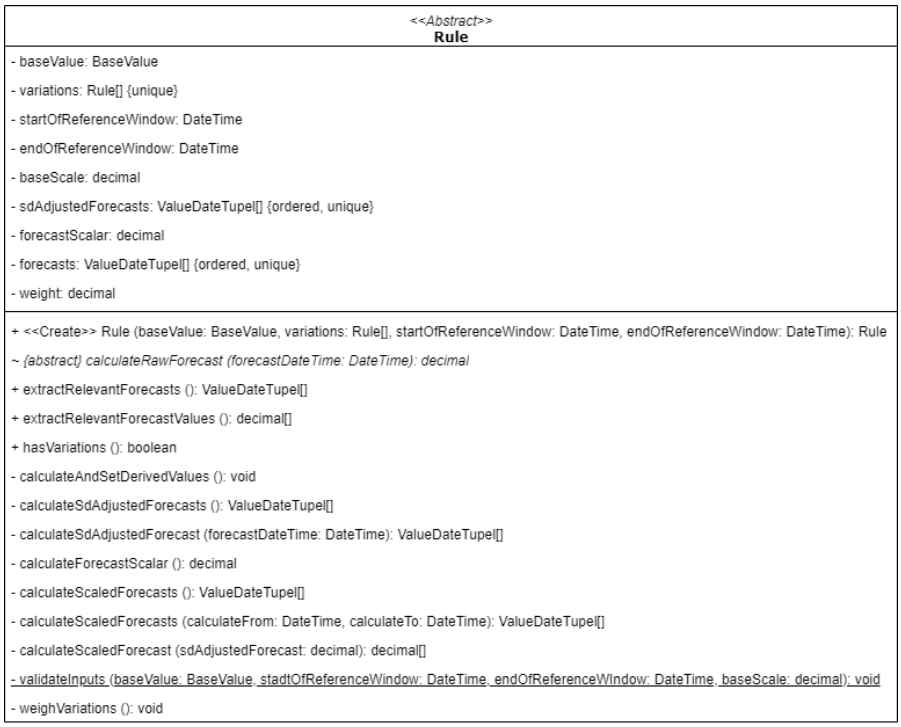
Gesamtklassendiagramm ohne Parameter und Methoden

Abbildung 19: Gesamtklassendiagramme ohne Parameter und Methoden



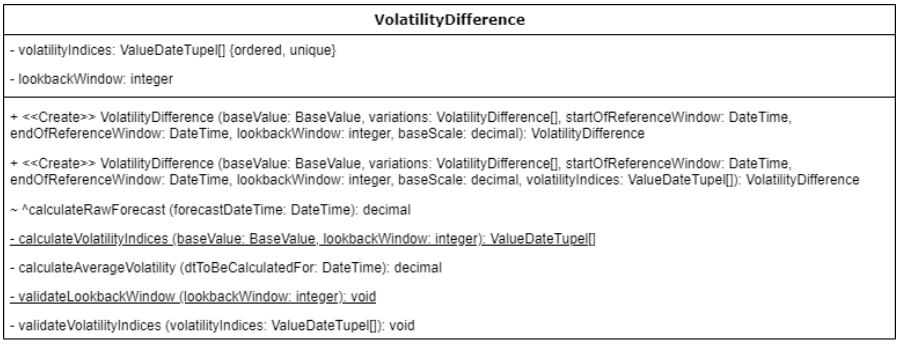
Komponente Rule

Abbildung 20: Klassendiagramm Komponente Rule



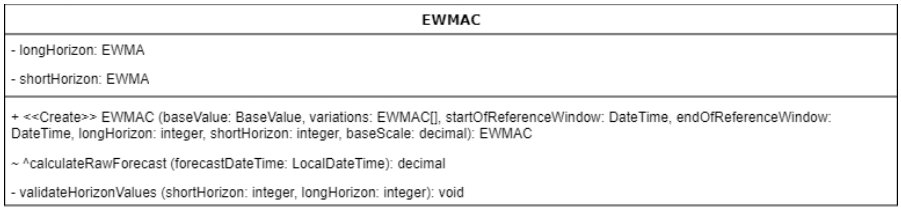
Komponente VolatilityDifference

Abbildung 21: Klassendiagramm Komponente VolatilityDifference



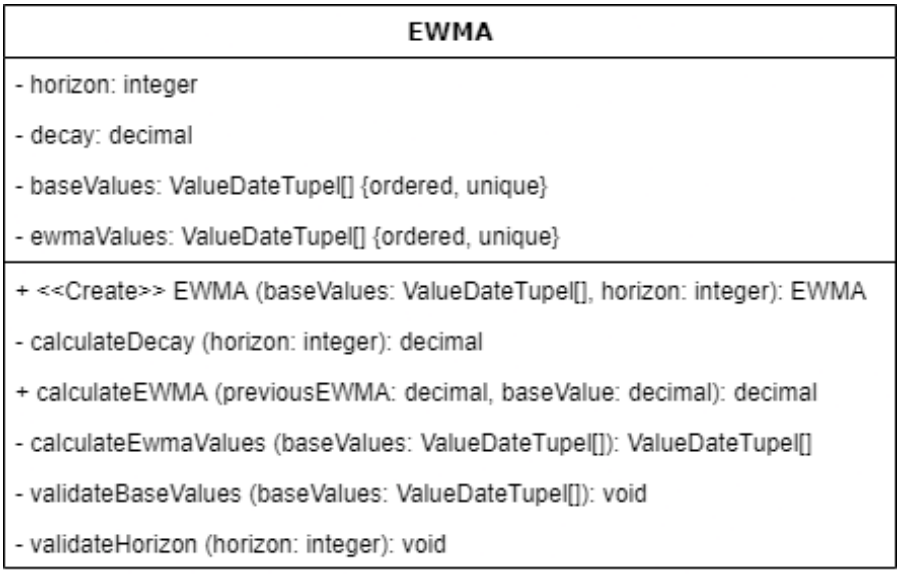
Komponente EWMAC

Abbildung 22: Klassendiagramm Komponente EWMAC



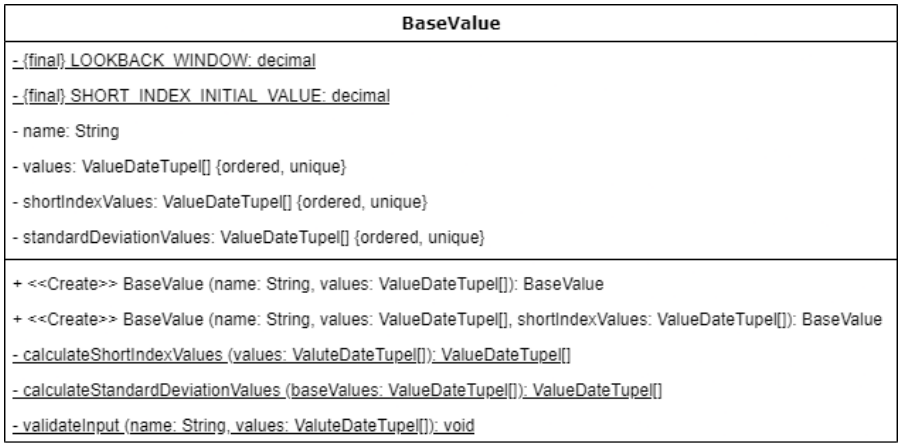
Komponente EWMA

Abbildung 23: Klassendiagramm Komponente EWMA



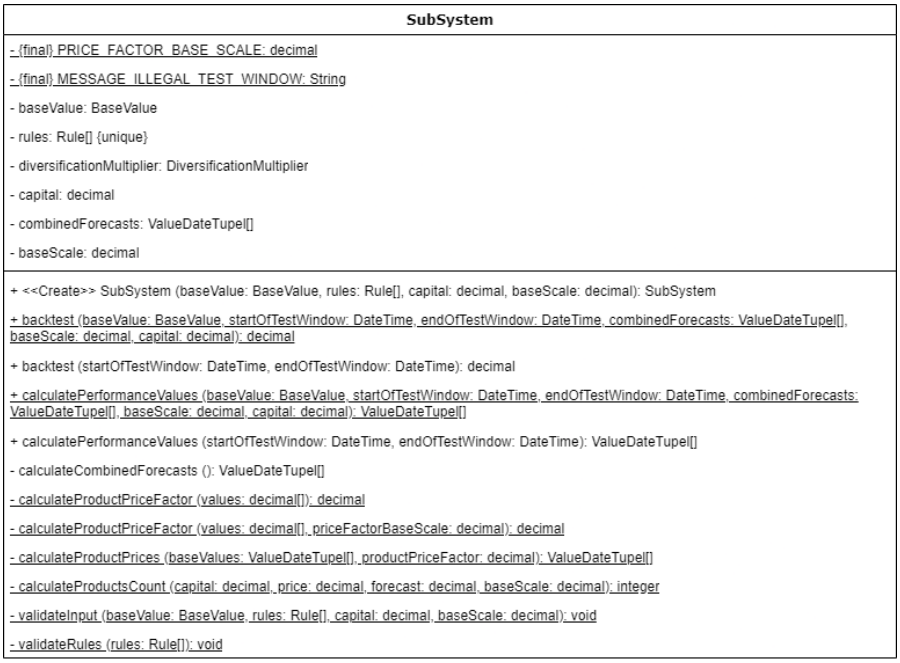
Komponente BaseValue

Abbildung 24: Klassendiagramm Komponente BaseValue



Komponente SubSystem

Abbildung 25: Klassendiagramm Komponente SubSystem



Komponente DiversificationMultiplier

Abbildung 26: Klassendiagramm Komponente DiversificationMultiplier

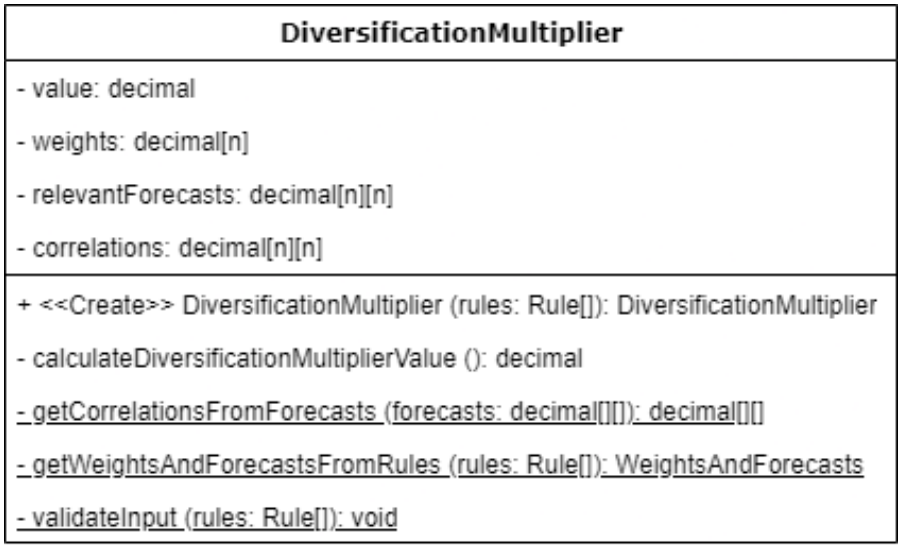
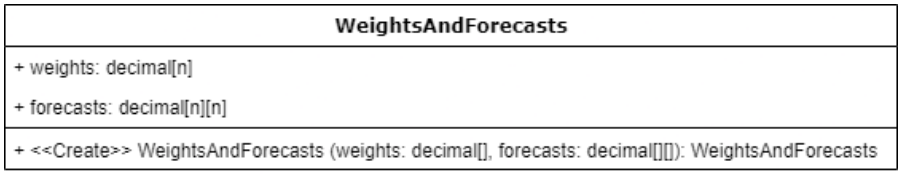


Abbildung 27: Klassendiagramm WeightsAndForecasts

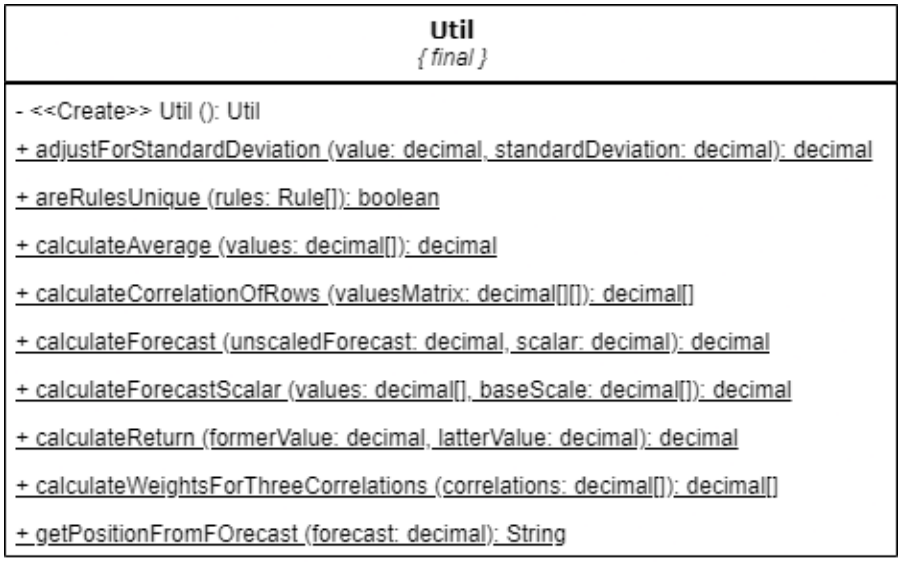


Komponente ValueDateTupel

Abbildung 28: Klassendiagramm Komponente ValueDateTupel

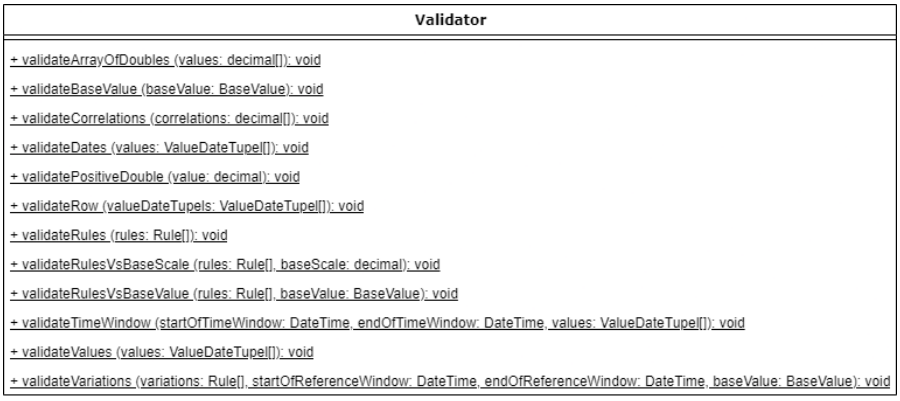
Komponente Util

Abbildung 29: Klassendiagramm Komponente Util



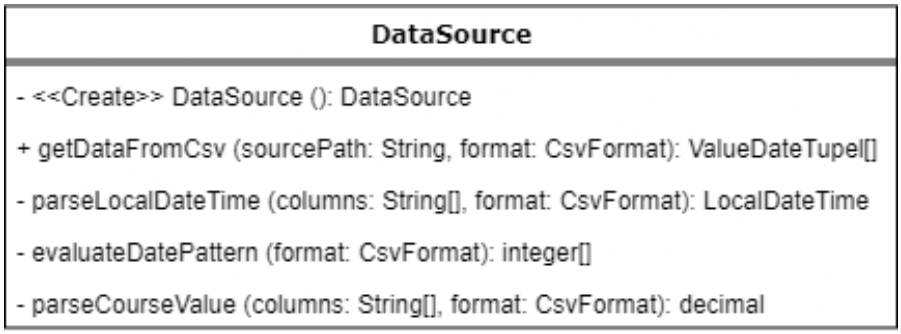
Komponente Validator

Abbildung 30: Klassendiagramm Komponente Validator



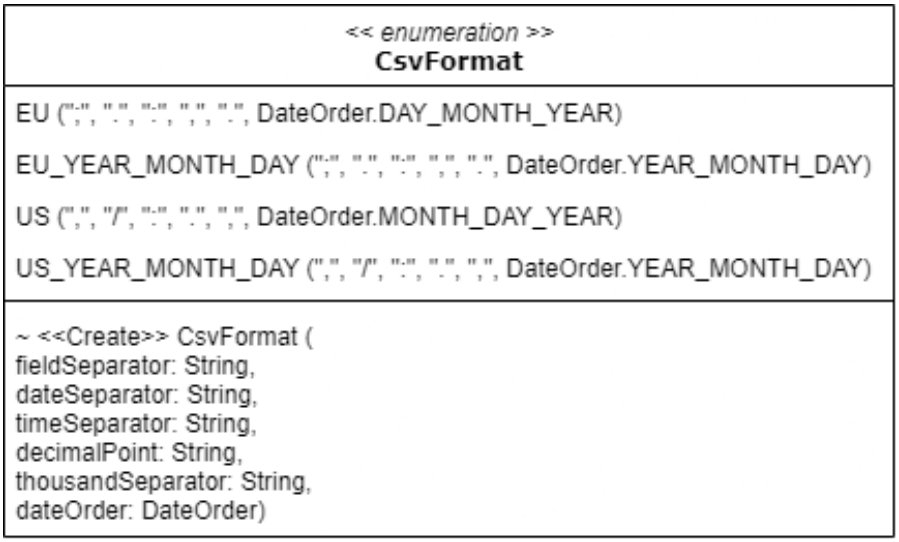
Komponente DataSource

Abbildung 31: Klassendiagramm Komponente DataSource



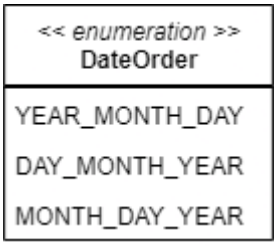
Komponente CsvFormat

Abbildung 32: Klassendiagramm Komponente CsvFormat



Komponente DateOrder

Abbildung 33: Klassendiagramm Komponente DateOrder



Abweichung der Druck- und Online-Versionen:

Die Anhänge *A3 (Programmcode)* und *A4 (Testcode)* auf den Seiten 123–350 werden aufgrund ihres Umfangs in der Druckversion des vorliegenden Bandes nicht angezeigt. Sie sind allen Interessierten jedoch in der Online-Version zugänglich, u. a. auf der Internetseite der Abteilung Publikationen der FOM Hochschule für Oekonomie & Management: https://www.fom.de/forschung/publikationen.html#!acc=arbeitspapiere-der-fom/accid=9122

A3 Programmcode

pom.xml

Listing 12: pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>de.rumford.tradingsystem</groupId>
  <artifactId>Tradingsystem</artifactId>
  <version>1.0.1</version>
  <packaging>jar</packaging>
  <profiles>
    <profile>
      <id>test</id>
      <properties>
        <env>test</env>
        <gebEnv>test</gebEnv>
        <jacoco.skip>false</jacoco.skip>
        <maven.test.skip>false</maven.test.skip>
        <skip.unit.tests>false</skip.unit.tests>
      </properties>
    </profile>
  </profiles>
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>
      UTF-8
    </project.reporting.outputEncoding>
  </properties>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.21.0</version>
        <dependencies>
          <dependency>
            <groupId>org.junit.platform</groupId>
            <artifactId>
              junit-platform-surefire-provider
            </artifactId>
            <version>1.2.0-M1</version>
          </dependency>
          <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
```

```
48.               <version>5.2.0-M1</version>
49.             </dependency>
50.           </dependencies>
51.         </plugin>
52.         <plugin>
53.           <groupId>org.jacoco</groupId>
54.           <artifactId>jacoco-maven-plugin</artifactId>
55.           <version>0.8.5</version>
56.           <executions>
57.             <execution>
58.               <goals>
59.                 <goal>prepare-agent</goal>
60.               </goals>
61.             </execution>
62.             <execution>
63.               <id>report</id>
64.               <phase>prepare-package</phase>
65.               <goals>
66.                 <goal>report</goal>
67.               </goals>
68.             </execution>
69.           </executions>
70.         </plugin>
71.         <plugin>
72.           <groupId>org.apache.maven.plugins</groupId>
73.           <artifactId>maven-javadoc-plugin</artifactId>
74.           <version>3.2.0</version>
75.         </plugin>
76.       </plugins>
77.     </build>
78.     <dependencies>
79.       <dependency>
80.         <groupId>org.junit.jupiter</groupId>
81.         <artifactId>junit-jupiter-api</artifactId>
82.         <version>5.6.2</version>
83.         <scope>test</scope>
84.       </dependency>
85.       <dependency>
86.         <groupId>org.apache.commons</groupId>
87.         <artifactId>commons-lang3</artifactId>
88.         <version>3.10</version>
89.       </dependency>
90.       <dependency>
91.         <groupId>org.apache.commons</groupId>
92.         <artifactId>commons-math3</artifactId>
93.         <version>3.6.1</version>
94.       </dependency>
95.       <dependency>
96.         <groupId>log4j</groupId>
97.         <artifactId>log4j</artifactId>
98.         <version>1.2.17</version>
99.       </dependency>
100.    </dependencies>
101.    <reporting>
```

```
      <plugins>
        <plugin>
          <groupId>org.jacoco</groupId>
          <artifactId>jacoco-maven-plugin</artifactId>
          <version>0.8.5</version>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-project-info-reports-plugin</artifactId>
          <version>3.0.0</version>
        </plugin>
      </plugins>
    </reporting>
  </project>
```

Komponente Rule

Listing 13: Komponente Rule

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.time.LocalDateTime;
4.  import java.util.Arrays;
5.
6.  import org.apache.commons.lang3.ArrayUtils;
7.
8.  import de.rumford.tradingsystem.helper.GeneratedCode;
9.  import de.rumford.tradingsystem.helper.Util;
10. import de.rumford.tradingsystem.helper.Validator;
11. import de.rumford.tradingsystem.helper.ValueDateTupel;
12.
13. /**
14.  * The Rules class provides the functionality necessary for every rule to
15.  * be used by the other classes of this library. Rules are the centerpiece
16.  * of a trading system. Based on these rules a system tries to forecast
17.  * future developments of a given asset and thus advises its user.
18.  * <p>
19.  * Although every investor should develop their own, these rules need to
20.  * share some functionality so they can actually be used in this trading
21.  * system. As soon as the forecast determining calculation is done (done
22.  * inside the implementation of
23.  * {@link #calculateRawForecast(LocalDateTime)}) all rules are treated
24.  * equally. This ensures compatibility and comparability between rules and
25.  * between trading systems.
26.  *
27.  * Abstract class to be extend on developing new rules for the trading
28.  * system.
29.  *
30.  * {@link #calculateAndSetDerivedValues()} is called on first invocation of
31.  * {@link #getSdAdjustedForecasts()} and {@link #getForecastScalar()}
32.  * respectively.
33.  *
34.  * @author Max Rumford
35.  */
36. public abstract class Rule {
37.
38.   /* The base value used for forecast calculation. */
39.   private BaseValue baseValue;
40.   /* The variations this rule might have. */
41.   private Rule[] variations;
42.   /*
43.    * A datetime representing the start of the reference window for forecast
44.    * scaling.
45.    */
46.   private LocalDateTime startOfReferenceWindow;
47.   /*
48.    * A datetime representing the end of the reference window for forecast
49.    * scaling.
```

```
50.       */
51.      private LocalDateTime endOfReferenceWindow;
52.      /* The value to which the forecasts will be scaled. */
53.      private double baseScale;
54.
55.      /* The standard deviation adjusted forecasts. */
56.      private ValueDateTupel[] sdAdjustedForecasts = null;
57.      /*
58.       * The scalar used to scale theses rules' forecasts to the given base
59.       * scale.
60.       */
61.      private double forecastScalar;
62.      /* The scaled forecasts. */
63.      private ValueDateTupel[] forecasts;
64.      /* The weights assigned to this rule. */
65.      private double weight;
66.
67.      /**
68.       * Public constructor for class Rule. Rule is an abstract class and
69.       * depends on the way of working of the extending class.
70.       *
71.       * @param baseValue              {@link BaseValue} The base value to be
72.       *                               used in this rule's calculations. See
73.       *                               {@link #validateInputs(BaseValue, Rule[],
74.       *                               LocalDateTime, LocalDateTime, double)}
75.       *                               for limitations.
76.       * @param variations             {@code Rule[]} An array of up to 3 rules
77.       *                               (or null). See
78.       *                               {@link #validateInputs(BaseValue, Rule[],
79.       *                               LocalDateTime, LocalDateTime, double)}
80.       *                               for limitations.
81.       * @param startOfReferenceWindow {@link LocalDateTime} The first
82.       *                               LocalDateTime to be considered in
83.       *                               calculations such as forecast scalar.
84.       *                               See
85.       *                               {@link #validateInputs(BaseValue, Rule[],
86.       *                               LocalDateTime, LocalDateTime, double)}
87.       *                               for limitations.
88.       * @param endOfReferenceWindow   {@link LocalDateTime} The last
89.       *                               LocalDateTime to be considered in
90.       *                               calculations such as forecast scalar.
91.       *                               See
92.       *                               {@link #validateInputs(BaseValue, Rule[],
93.       *                               LocalDateTime, LocalDateTime, double)}
94.       *                               for limitations.
95.       * @param baseScale              {@code double} How the forecasts shall
96.       *                               be scaled. See
97.       *                               {@link #validateInputs(BaseValue, Rule[],
98.       *                               LocalDateTime, LocalDateTime, double)}
99.       *                               for limitations.
100.      */
101.     public Rule(BaseValue baseValue, Rule[] variations,
102.         LocalDateTime startOfReferenceWindow,
103.         LocalDateTime endOfReferenceWindow, double baseScale) {
```

```
104.
105.       validateInputs(baseValue, variations, startOfReferenceWindow,
106.           endOfReferenceWindow, baseScale);
107.
108.       this.setBaseValue(baseValue);
109.       this.setStartOfReferenceWindow(startOfReferenceWindow);
110.       this.setEndOfReferenceWindow(endOfReferenceWindow);
111.       this.setVariations(variations);
112.       this.weighVariations();
113.       this.setBaseScale(baseScale);
114.     }
115.
116.     /**
117.      * Calculates the raw forecast of this rule for a given LocalDateTime.
118.      * The calculation of this value will heavily depend on the type of rule
119.      * extending this abstract class.
120.      *
121.      * @param forecastDateTime {@link LocalDateTime} The dateTime the raw
122.      *                         forecast shall be calculated for.
123.      * @return {@code double} The raw forecast value for the given
124.      *         LocalDateTime.
125.      */
126.     abstract double calculateRawForecast(LocalDateTime forecastDateTime);
127.
128.     /**
129.      * Extract the relevant forecasts for this rule.
130.      *
131.      * @return {@code ValueDateTupel[]} An array of the relevant forecasts
132.      *         for this rule.
133.      */
134.     public final ValueDateTupel[] extractRelevantForecasts() {
135.       return ValueDateTupel.getElements(this.getForecasts(),
136.           this.getStartOfReferenceWindow(), this.getEndOfReferenceWindow());
137.     }
138.
139.     /**
140.      * Extract the relevant forecast values for this rule.
141.      *
142.      * @return {@code double[]} An array of the relevant forecast values for
143.      *         this rule.
144.      */
145.     public final double[] extractRelevantForecastValues() {
146.       return ValueDateTupel.getValues(this.extractRelevantForecasts());
147.     }
148.
149.     /**
150.      * Evaluates if the current rule has variations.
151.      *
152.      * @return {@code boolean} True, if the rule has variations. False
153.      *         otherwise.
154.      */
155.     public final boolean hasVariations() {
156.       return this.getVariations() != null;
157.     }
```

```
158.
159.     /**
160.      * Calculates all values derived from raw forecasts. This takes into
161.      * consideration that not all relevant values might be known upon call of
162.      * Rule constructor.
163.      */
164.     private void calculateAndSetDerivedValues() {
165.       this.setSdAdjustedForecasts(this.calculateSdAdjustedForecasts());
166.       this.setForecastScalar(this.calculateForecastScalar());
167.       this.setForecasts(this.calculateScaledForecasts());
168.     }
169.
170.     /**
171.      * Calculates the standard deviation adjusted forecasts for this rule,
172.      * beginning from the start of the instance's reference window.
173.      *
174.      * @return {@code ValueDateTupel[]} An array of standard deviation
175.      *         adjusted forecasts.
176.      */
177.     private ValueDateTupel[] calculateSdAdjustedForecasts() {
178.       /*
179.        * All dates from startOfReferenceWindow are relevant for the
180.        * calculation
181.        */
182.       LocalDateTime[] relevantDates = ValueDateTupel.getDates(
183.           ValueDateTupel.getElements(this.getBaseValue().getValues(),
184.               this.getStartOfReferenceWindow(), null));
185.
186.       ValueDateTupel[] calculatedSdAdjustedForecasts = {};
187.
188.       /* For all relevant dates: Calculate the sd adjusted forecast */
189.       for (LocalDateTime dt : relevantDates) {
190.         calculatedSdAdjustedForecasts = ArrayUtils.add(
191.             calculatedSdAdjustedForecasts,
192.             new ValueDateTupel(dt, this.calculateSdAdjustedForecast(dt)));
193.       }
194.       return calculatedSdAdjustedForecasts;
195.     }
196.
197.     /**
198.      * Calculates the standard deviation adjusted Forecast for a given
199.      * LocalDateTime.
200.      *
201.      * @param forecastDateTime {@link LocalDateTime} The LocalDateTime the
202.      *                         forecast is to be calculated of.
203.      * @return {@code double} The standard deviation adjusted value.
204.      *         Double.NaN if the standard deviation at the given
205.      *         LocalDateTime is zero.
206.      */
207.     private double calculateSdAdjustedForecast(
208.         LocalDateTime forecastDateTime) {
209.       if (this.hasVariations()) {
210.         return Double.NaN;
211.       }
```

```
212.
213.       double rawForecast = this.calculateRawForecast(forecastDateTime);
214.
215.       double sdValue = ValueDateTupel
216.           .getElement(this.getBaseValue().getStandardDeviationValues(),
217.               forecastDateTime)
218.           .getValue();
219.
220.       return Util.adjustForStandardDeviation(rawForecast, sdValue);
221.     }
222.
223.     /**
224.      * Calculates the forecast scalar. If this rule has variations the
225.      * variations' forecasts and respective weights are used to calculate the
226.      * forecast scalar. Else this rule's standard deviation adjusted values
227.      * are used.
228.      *
229.      * @param baseScale {@code double} The base scale to which the forecast
230.      *                  scalar should scale the forecasts.
231.      * @return {@code double} The calculated forecast scalar.
232.      */
233.     private double calculateForecastScalar() {
234.
235.       double instanceBaseScale = this.getBaseScale();
236.       Rule[] instanceVariations = this.getVariations();
237.       ValueDateTupel[] relevantForecastValues;
238.
239.       /*
240.        * If the rule has variations, use the variations' forecasts multiplied
241.        * with their respective weights method to get the base for the
242.        * forecast scalar.
243.        */
244.       if (instanceVariations != null) {
245.         /* local array of weighted and combined variations' forecasts. */
246.         relevantForecastValues = ValueDateTupel
247.             .createEmptyArray(instanceVariations[0].getForecasts().length);
248.
249.         /* Loop over each variation */
250.         for (Rule variation : instanceVariations) {
251.           /* Loop over each forecast value inside each variation. */
252.           for (int i = 0; i < variation.getForecasts().length; i++) {
253.
254.             /*
255.              * Calculate the value to be added to the current weighted
256.              * forecast value for this rule
257.              */
258.             double valueToBeAdded = variation.getForecasts()[i].getValue()
259.                 * variation.getWeight();
260.
261.             /*
262.              * If the variations forecast value at this position is null
263.              * (i.e. when we're in the first variation's loop) create a new
264.              * ValueDateTupel
265.              */
```

```
266.            if (relevantForecastValues[i] == null) {
267.              relevantForecastValues[i] = new ValueDateTupel(
268.                  variation.getForecasts()[i].getDate(), valueToBeAdded);
269.            } else {
270.              /*
271.               * If there already is a value at position i add the value to
272.               * the existing value
273.               */
274.              relevantForecastValues[i].setValue(
275.                  relevantForecastValues[i].getValue() + valueToBeAdded);
276.            }
277.          }
278.        }
279.
280.      } else {
281.        /*
282.         * If the rule doesn't have variations use this rules sd adjusted
283.         * forecast values
284.         */
285.        relevantForecastValues = this.getSdAdjustedForecasts();
286.      }
287.
288.      relevantForecastValues = ValueDateTupel.getElements(
289.          relevantForecastValues, this.getStartOfReferenceWindow(),
290.          this.getEndOfReferenceWindow());
291.
292.      double calculatedForecastScalar = Util.calculateForecastScalar(
293.          ValueDateTupel.getValues(relevantForecastValues),
294.          instanceBaseScale);
295.      if (Double.isNaN(calculatedForecastScalar))
296.        throw new IllegalArgumentException(
297.            "Illegal values in calulated forecast values."
298.                + " Adjust reference window.");
299.
300.      return calculatedForecastScalar;
301.    }
302.
303.    /**
304.     * Calculates the scaled forecasts for this rule, starting from this
305.     * rule's start of reference window.
306.     *
307.     * @return {@code ValueDateTupel[]} This rules scaled forecasts.
308.     */
309.    private ValueDateTupel[] calculateScaledForecasts() {
310.      return this.calculateScaledForecasts(this.getStartOfReferenceWindow(),
311.          null);
312.    }
313.
314.    /**
315.     * Calculates the scaled forecasts for a given window of time. If this
316.     * rule has variations, their forecasts are used being weighted and
317.     * scaled. If this rule has no variations, this rule's standard deviation
318.     * adjusted forecasts are used.
319.     *
```

```
320.      * @param calculateFrom {@link LocalDateTime} The starting dateTime.
321.      * @param calculateTo   {@link LocalDateTime} The ending dateTime.
322.      * @return {@code ValueDateTupel[]} An array of scaled forecasts.
323.      */
324.     private ValueDateTupel[] calculateScaledForecasts(
325.         LocalDateTime calculateFrom, LocalDateTime calculateTo) {
326.       ValueDateTupel[] calculatedScaledForecasts = null;
327.
328.       Rule[] instanceVariations = this.getVariations();
329.       /*
330.        * If a rule has variations only their forecasts matter. This rule's
331.        * forecasts are the set to equal the forecasts of its variations.
332.        */
333.       if (instanceVariations != null) {
334.         ValueDateTupel[][] variationsForecasts = {};
335.         double[] variationsWeights = {};
336.
337.         /* Extract forecasts and weights of all variations. */
338.         for (Rule variation : instanceVariations) {
339.           variationsForecasts = ArrayUtils.add(variationsForecasts,
340.               variation.getForecasts());
341.           variationsWeights = ArrayUtils.add(variationsWeights,
342.               variation.getWeight());
343.         }
344.
345.         calculatedScaledForecasts = ValueDateTupel
346.             .createEmptyArray(variationsForecasts[0].length);
347.
348.         /* Loop over all variations */
349.         for (int variationsIndex = 0;
350.             variationsIndex < variationsForecasts.length;
351.             variationsIndex++) {
352.
353.           /* Loop over all forecasts for each variation. */
354.           for (int i = 0;
355.               i < variationsForecasts[variationsIndex].length;
356.               i++) {
357.
358.             /* Add the weighted and scaled variation's forecast */
359.             double valueToBeAdded = variationsForecasts[variationsIndex][i]
360.                 .getValue() * variationsWeights[variationsIndex];
361.
362.             /*
363.              * If the scaled and weighted forecast value at this position is
364.              * null (i.e. when we're in the first variation's loop) create a
365.              * new ValueDateTupel
366.              */
367.             if (calculatedScaledForecasts[i] == null) {
368.               calculatedScaledForecasts[i] = new ValueDateTupel(
369.                   variationsForecasts[variationsIndex][i].getDate(),
370.                   valueToBeAdded);
371.             } else {
372.               /*
373.                * If there already is a value at position i add the weighted
```

```
374.                * and scaled forecast.
375.                */
376.               calculatedScaledForecasts[i].setValue(
377.                   calculatedScaledForecasts[i].getValue() + valueToBeAdded);
378.             }
379.           }
380.         }
381.         return calculatedScaledForecasts;
382.       }
383.       /*
384.        * If the Rule does not have variations, use the sd adjusted forecasts
385.        * of this Rule alone.
386.        */
387.       LocalDateTime[] relevantDates = ValueDateTupel.getDates(
388.           ValueDateTupel.getElements(this.getBaseValue().getValues(),
389.               calculateFrom, calculateTo));
390.
391.       ValueDateTupel[] instanceSdAdjustedForecasts = this
392.           .getSdAdjustedForecasts();
393.
394.       for (int i = 0; i < relevantDates.length; i++) {
395.         LocalDateTime dt = relevantDates[i];
396.         calculatedScaledForecasts = ArrayUtils.add(calculatedScaledForecasts,
397.             new ValueDateTupel(dt, this.calculateScaledForecast(
398.                 instanceSdAdjustedForecasts[i].getValue())));
399.       }
400.       return calculatedScaledForecasts;
401.     }
402.
403.     /**
404.      * Calculate the scaled forecast for the given LocalDateTime. Cut off
405.      * forecast values if they exceed 2 * base scale positively or -2 * base
406.      * scale negatively.
407.      *
408.      * @param sdAdjustedForecast {@link double} The standard deviation
409.      *                           adjusted value to be scaled.
410.      * @return {@code double} the scaled forecast value.
411.      */
412.     private double calculateScaledForecast(double sdAdjustedForecast) {
413.       double instanceBaseScale = this.getBaseScale();
414.       double instanceForecastScalar = this.getForecastScalar();
415.
416.       final double MAX_FORECAST = instanceBaseScale * 2;
417.       final double MIN_FORECAST = 0 - MAX_FORECAST;
418.
419.       double scaledForecast = sdAdjustedForecast * instanceForecastScalar;
420.
421.       if (scaledForecast > MAX_FORECAST)
422.         return MAX_FORECAST;
423.
424.       if (scaledForecast < MIN_FORECAST)
425.         return MIN_FORECAST;
426.
427.       return scaledForecast;
```

```
428.     }
429.
430.     /**
431.      * Validates if the given instance variables meet specifications.
432.      *
433.      * @param baseValue              {@link BaseValue} The base value to be
434.      *                               used in this rule's calculations. Must
435.      *                               pass
436.      *                               {@link Validator# validateBaseValue(
437.      *                               BaseValue)}.
438.      *                               Its values must pass
439.      *                               {@link Validator#validateTimeWindow(
440.      *                               LocalDateTime, LocalDateTime,
441.      *                               ValueDateTupel[])}
442.      * @param variations             {@code Rule[]} Can be null. If not
443.      *                               <ul>
444.      *                               <li>Must not contain more than 3
445.      *                               elements.</li>
446.      *                               <li>Must not contain 0 elements.</li>
447.      *                               <li>Must not contain null.</li>
448.      *                               <li>All elements must have matching
449.      *                               startOfReferenceWindow and
450.      *                               endOfReferenceWindow and must be the
451.      *                               same as given startOfReferenceWindow and
452.      *                               endOfReferenceWindow.</li>
453.      *                               </ul>
454.      * @param startOfReferenceWindow {@link LocalDateTime} The first
455.      *                               LocalDateTime to be considered in
456.      *                               calculations such as forecast scalar.
457.      *                               Must pass
458.      *                               {@link Validator#validateTimeWindow(
459.      *                               LocalDateTime, LocalDateTime,
460.      *                               ValueDateTupel[])}
461.      * @param endOfReferenceWindow   {@link LocalDateTime} The last
462.      *                               LocalDateTime to be considered in
463.      *                               calculations such as forecast scalar.
464.      *                               Must pass
465.      *                               {@link Validator#validateTimeWindow(
466.      *                               LocalDateTime, LocalDateTime,
467.      *                               ValueDateTupel[])}
468.      * @param baseScale              {@code double} How the forecasts shall
469.      *                               be scaled. Must pass
470.      *                               {@link Validator#
471.      *                               validatePositiveDouble(double)}.
472.      * @throws IllegalArgumentException if the above specifications are not
473.      *                                  met.
474.      */
475.     private static void validateInputs(BaseValue baseValue,
476.         Rule[] variations, LocalDateTime startOfReferenceWindow,
477.         LocalDateTime endOfReferenceWindow, double baseScale) {
478.
479.       Validator.validateBaseValue(baseValue);
480.
481.       try {
```

```
482.         Validator.validateTimeWindow(startOfReferenceWindow,
483.             endOfReferenceWindow, baseValue.getValues());
484.       } catch (IllegalArgumentException e) {
485.         /*
486.          * If the message contains "values" the message references an error
487.          * in the given base values in combination with the given reference
488.          * window.
489.          */
490.         if (e.getMessage().contains("values"))
491.           throw new IllegalArgumentException(
492.               "Given base value and reference window do not fit.", e);
493.
494.         throw new IllegalArgumentException(
495.             "The given reference window does not meet specifications.", e);
496.       }
497.
498.       /*
499.        * The first time interval of the base values cannot have all derived
500.        * values correctly calculated, as there will be no returns (due to
501.        * lacking former value).
502.        */
503.       if (baseValue.getValues()[0].getDate().equals(startOfReferenceWindow))
504.         throw new IllegalArgumentException(
505.             "Reference window must not start on first time interval of"
506.                 + " base value data.");
507.
508.       /*
509.        * A rule can have no variations, so variations == null is acceptable.
510.        */
511.       if (variations != null) {
512.         Validator.validateVariations(variations, startOfReferenceWindow,
513.             endOfReferenceWindow, baseValue);
514.       }
515.
516.       try {
517.         Validator.validatePositiveDouble(baseScale);
518.       } catch (IllegalArgumentException e) {
519.         throw new IllegalArgumentException(
520.             "The given base scale does not meet specifications.", e);
521.       }
522.     }
523.
524.     /**
525.      * Calculates and sets the weights for this rule's variations based on
526.      * their correlations. This calculation is an approximation of (Robert
527.      * Carver, Systematic Trading (2015), p. 79, Table 8). Using the actual
528.      * table would muddy the weights and render them inaccurate.
529.      */
530.     private void weighVariations() {
531.       Rule[] instanceVariations = this.getVariations();
532.       if (instanceVariations == null)
533.         return;
534.
535.       switch (instanceVariations.length) {
```

```
536.      case 1:
537.        /* If there is only 1 variation then its weight is 100% */
538.        instanceVariations[0].setWeight(1d);
539.        break;
540.
541.      case 2:
542.        /* If there are 2 variations their weights are 50% each */
543.        instanceVariations[0].setWeight(0.5d);
544.        instanceVariations[1].setWeight(0.5d);
545.        break;
546.
547.      case 3:
548.        /*
549.         * Extract the values from the forecasts array, as the Dates are not
550.         * needed for correlation calculation.
551.         */
552.        double[][] variationsForecasts = {};
553.        for (Rule variation : instanceVariations) {
554.          ValueDateTupel[] fcs = variation.extractRelevantForecasts();
555.          variationsForecasts = ArrayUtils.add(variationsForecasts,
556.              ValueDateTupel.getValues(fcs));
557.        }
558.
559.        /* Find the correlations for the given variations. */
560.        double[] correlations = Util
561.            .calculateCorrelationOfRows(variationsForecasts);
562.
563.        if (ArrayUtils.contains(correlations, Double.NaN))
564.          throw new IllegalArgumentException(
565.              "Correlations cannot be calculated due to illegal values"
566.                  + " in given variations.");
567.
568.        /* Find the weights corresponding to the calculated correlations. */
569.        double[] weights = Util
570.            .calculateWeightsForThreeCorrelations(correlations);
571.
572.        /* Set the weights of the underlying variations */
573.        for (int i = 0; i < weights.length; i++) {
574.          instanceVariations[i].setWeight(weights[i]);
575.        }
576.        break;
577.
578.      default:
579.        throw new IllegalStateException(
580.            "A rule should not have this many variations: "
581.                + instanceVariations.length);
582.      }
583.    }
584.
585.    /**
586.     * ========================================================================
587.     * OVERRIDES
588.     * ========================================================================
589.     */
```

```
590.
591.     /* A hash code for this Rule. */
592.     @GeneratedCode
593.     @Override
594.     public int hashCode() {
595.       final int prime = 31;
596.       int result = 1;
597.       long temp;
598.       temp = Double.doubleToLongBits(baseScale);
599.       result = prime * result + (int) (temp ^ (temp >>> 32));
600.       result = prime * result
601.           + ((baseValue == null) ? 0 : baseValue.hashCode());
602.       result = prime * result + ((endOfReferenceWindow == null) ? 0
603.           : endOfReferenceWindow.hashCode());
604.       temp = Double.doubleToLongBits(forecastScalar);
605.       result = prime * result + (int) (temp ^ (temp >>> 32));
606.       result = prime * result + Arrays.hashCode(forecasts);
607.       result = prime * result + Arrays.hashCode(sdAdjustedForecasts);
608.       result = prime * result + ((startOfReferenceWindow == null) ? 0
609.           : startOfReferenceWindow.hashCode());
610.       result = prime * result + Arrays.hashCode(variations);
611.       temp = Double.doubleToLongBits(weight);
612.       result = prime * result + (int) (temp ^ (temp >>> 32));
613.       return result;
614.     }
615.
616.     /**
617.      * Checks if this Rule is equal to another Rule.
618.      */
619.     @GeneratedCode
620.     @Override
621.     public boolean equals(Object obj) {
622.       if (this == obj)
623.         return true;
624.       if (obj == null)
625.         return false;
626.       if (getClass() != obj.getClass())
627.         return false;
628.       Rule other = (Rule) obj;
629.       if (Double.doubleToLongBits(baseScale) != Double
630.           .doubleToLongBits(other.baseScale))
631.         return false;
632.       if (baseValue == null) {
633.         if (other.baseValue != null)
634.           return false;
635.       } else if (!baseValue.equals(other.baseValue))
636.         return false;
637.       if (endOfReferenceWindow == null) {
638.         if (other.endOfReferenceWindow != null)
639.           return false;
640.       } else if (!endOfReferenceWindow.equals(other.endOfReferenceWindow))
641.         return false;
642.       if (Double.doubleToLongBits(forecastScalar) != Double
643.           .doubleToLongBits(other.forecastScalar))
```

```
        return false;
      if (!Arrays.equals(forecasts, other.forecasts))
        return false;
      if (!Arrays.equals(sdAdjustedForecasts, other.sdAdjustedForecasts))
        return false;
      if (startOfReferenceWindow == null) {
        if (other.startOfReferenceWindow != null)
          return false;
      } else if (!startOfReferenceWindow
          .equals(other.startOfReferenceWindow))
        return false;
      if (!Arrays.equals(variations, other.variations))
        return false;
      if (Double.doubleToLongBits(weight) != Double
          .doubleToLongBits(other.weight))
        return false;
      return true;
    }

    /**
     * Outputs the fields of this Rule as a {@code String}.
     */
    @GeneratedCode
    @Override
    public String toString() {
      StringBuilder builder = new StringBuilder();
      builder.append("Rule [baseValue=");
      builder.append(baseValue);
      builder.append(", variations=");
      builder.append(Arrays.toString(variations));
      builder.append(", startOfReferenceWindow=");
      builder.append(startOfReferenceWindow);
      builder.append(", endOfReferenceWindow=");
      builder.append(endOfReferenceWindow);
      builder.append(", baseScale=");
      builder.append(baseScale);
      builder.append(", sdAdjustedForecasts=");
      builder.append(Arrays.toString(sdAdjustedForecasts));
      builder.append(", forecastScalar=");
      builder.append(forecastScalar);
      builder.append(", forecasts=");
      builder.append(Arrays.toString(forecasts));
      builder.append(", weight=");
      builder.append(weight);
      builder.append("]");
      return builder.toString();
    }

    /**
     * ========================================================================
     * GETTERS AND SETTERS
     * ========================================================================
     */

```

```
698.     /**
699.      * Get the base value of this rule.
700.      *
701.      * @return baseValue {@link BaseValue} The base value of this instance of
702.      *         rule.
703.      */
704.     public final BaseValue getBaseValue() {
705.       return baseValue;
706.     }
707.
708.     /**
709.      * Set the base value of this rule.
710.      *
711.      * @param baseValue {@link BaseValue} the baseValue to bet set for this
712.      *                  instance of rule.
713.      */
714.     private void setBaseValue(BaseValue baseValue) {
715.       this.baseValue = baseValue;
716.     }
717.
718.     /**
719.      * Get the forecast scalar of this rule. Invokes
720.      * {@link #calculateAndSetDerivedValues()} if
721.      * {@code (this.sdAdjustedForecasts == null)} evaluates to {@code true}.
722.      *
723.      * @return {@code double} forecast scalar of this rule
724.      */
725.     public final double getForecastScalar() {
726.       if (sdAdjustedForecasts == null)
727.         this.calculateAndSetDerivedValues();
728.       return forecastScalar;
729.     }
730.
731.     /**
732.      * Set the forecast scalar of this rule
733.      *
734.      * @param forecastScalar {@code double} forecast scalar to be set for
735.      *                       this rule
736.      */
737.     private void setForecastScalar(double forecastScalar) {
738.       this.forecastScalar = forecastScalar;
739.     }
740.
741.     /**
742.      * Get the weight of this rule
743.      *
744.      * @return {@code double} the weight of this rule
745.      */
746.     public final double getWeight() {
747.       return weight;
748.     }
749.
750.     /**
751.      * Set the weight of this rule
```

```
     *
     * @param weight {@code double} the weight to be set for this rule
     */
    private void setWeight(double weight) {
      this.weight = weight;
    }

    /**
     * Get the variations for this rule.
     *
     * @return variations Rule
     */
    public final Rule[] getVariations() {
      return variations;
    }

    /**
     * Set the variations for this rule.
     *
     * @param variations the variations to set
     */
    private void setVariations(Rule[] variations) {
      this.variations = variations;
    }

    /**
     * Get the end of reference window for this rule.
     *
     * @return startOfReferenceWindow Rule
     */
    public final LocalDateTime getStartOfReferenceWindow() {
      return startOfReferenceWindow;
    }

    /**
     * Set the start of reference window for this rule.
     *
     * @param startOfReferenceWindow the startOfReferenceWindow to set
     */
    private void setStartOfReferenceWindow(
        LocalDateTime startOfReferenceWindow) {
      this.startOfReferenceWindow = startOfReferenceWindow;
    }

    /**
     * Get the end of reference window for this rule.
     *
     * @return endOfReferenceWindow Rule
     */
    public final LocalDateTime getEndOfReferenceWindow() {
      return endOfReferenceWindow;
    }

    /**
```

```
806.      * Set the end of reference window for this rule.
807.      *
808.      * @param endOfReferenceWindow the endOfReferenceWindow to set
809.      */
810.     private void setEndOfReferenceWindow(
811.         LocalDateTime endOfReferenceWindow) {
812.       this.endOfReferenceWindow = endOfReferenceWindow;
813.     }
814.
815.     /**
816.      * Get the base scale for this rule.
817.      *
818.      * @return baseScale Rule
819.      */
820.     public final double getBaseScale() {
821.       return baseScale;
822.     }
823.
824.     /**
825.      * Set the base scale for this rule.
826.      *
827.      * @param baseScale the baseScale to set
828.      */
829.     private void setBaseScale(double baseScale) {
830.       this.baseScale = baseScale;
831.     }
832.
833.     /**
834.      * Get the adjusted and scaled forecasts of this Rule. Invokes
835.      * {@link #calculateAndSetDerivedValues()} if
836.      * {@code (this.sdAdjustedForecasts == null)} evaluates to {@code true}.
837.      *
838.      * @return forecasts {@code ValueDateuTupel[]} The adjusted and scaled
839.      *         forecasts of this Rule.
840.      */
841.     public final ValueDateTupel[] getForecasts() {
842.       if (sdAdjustedForecasts == null)
843.         this.calculateAndSetDerivedValues();
844.       return forecasts;
845.     }
846.
847.     /**
848.      * Set the standard deviation adjusted forecasts for this rule.
849.      *
850.      * @param forecasts the forecasts to set
851.      */
852.     private void setForecasts(ValueDateTupel[] forecasts) {
853.       this.forecasts = forecasts;
854.     }
855.
856.     /**
857.      * Get the standard deviation adjusted forecasts for this rule.
858.      *
859.      * @return sdAdjustedForecasts Rule
```

```
860.      */
861.     private ValueDateTupel[] getSdAdjustedForecasts() {
862.       return sdAdjustedForecasts;
863.     }
864.
865.     /**
866.      * Set the standard deviation adjusted forecasts of this Rule.
867.      *
868.      * @param sdAdjustedForecasts the sdAdjustedForecasts to set
869.      */
870.     private void setSdAdjustedForecasts(
871.         ValueDateTupel[] sdAdjustedForecasts) {
872.       this.sdAdjustedForecasts = sdAdjustedForecasts;
873.     }
874.   }
```

Komponente VolatilityDifference

Listing 14: Komponente VolatilityDifference

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.time.LocalDateTime;
4.  import java.util.Arrays;
5.  import java.util.DoubleSummaryStatistics;
6.  import java.util.HashSet;
7.  import java.util.Set;
8.
9.  import org.apache.commons.lang3.ArrayUtils;
10. import org.apache.commons.math3.stat.descriptive.moment.StandardDeviation;
11.
12. import de.rumford.tradingsystem.helper.GeneratedCode;
13. import de.rumford.tradingsystem.helper.Util;
14. import de.rumford.tradingsystem.helper.Validator;
15. import de.rumford.tradingsystem.helper.ValueDateTupel;
16.
17. /**
18.  * The VolatilityDifference is a {@link Rule} providing forecasts based on
19.  * historic and recent volatility of the base value.
20.  * <p>
21.  * Historically, times of higher volatility in a asset tended to be
22.  * accompanied by falls in course value. This rule exploits this behavior
23.  * by comparing the current volatility of an asset with its historic
24.  * counterpart.
25.  *
26.  * @author Max Rumford
27.  *
28.  */
29. public class VolatilityDifference extends Rule {
30.
31.   /*
32.    * An array of values representing volatility values for a given base
33.    * value.
34.    */
35.   private ValueDateTupel[] volatilityIndices;
36.   /* The lookback window used for volatility calculation. */
37.   private int lookbackWindow;
38.
39.   /**
40.    * Creates a new VolatilityDifference instance using the passed
41.    * {@link BaseValue} to calculate the volatility indices and the average
42.    * volatility.
43.    *
44.    * @param baseValue              Same as in
45.    *                               {@link Rule#Rule(BaseValue, Rule[],
46.    *                               LocalDateTime, LocalDateTime, double)}.
47.    * @param variations             {@code VolatilityDifference[]} An array
48.    *                               of three or less rules. Represents the
49.    *                               variations of this rule. Same
```

```
50.       *                                 limitations as in
51.       *                                 {@link Rule#Rule(BaseValue, Rule[],
52.       *                                 LocalDateTime, LocalDateTime, double)}.
53.       * @param startOfReferenceWindow Same as in
54.       *                                 {@link Rule#Rule(BaseValue, Rule[],
55.       *                                 LocalDateTime, LocalDateTime, double)}.
56.       * @param endOfReferenceWindow   Same as in
57.       *                                 {@link Rule#Rule(BaseValue, Rule[],
58.       *                                 LocalDateTime, LocalDateTime, double)}.
59.       * @param lookbackWindow         {@code int} The lookback window to be
60.       *                                 used for this VolatilityDifference. See
61.       *                                 {@link #validateLookbackWindow(int)} for
62.       *                                 limitations.
63.       * @param baseScale              Same as in
64.       *                                 {@link Rule#Rule(BaseValue, Rule[],
65.       *                                 LocalDateTime, LocalDateTime, double)}.
66.       */
67.      public VolatilityDifference(BaseValue baseValue,
68.          VolatilityDifference[] variations,
69.          LocalDateTime startOfReferenceWindow,
70.          LocalDateTime endOfReferenceWindow, int lookbackWindow,
71.          double baseScale) {
72.        super(baseValue, variations, startOfReferenceWindow,
73.            endOfReferenceWindow, baseScale);
74.
75.        if (variations == null) {
76.          /*
77.           * Lookback window and volatility indices are only needed when a rule
78.           * has no variations.
79.           */
80.          /* Check if lookback window fulfills requirements. */
81.          validateLookbackWindow(lookbackWindow);
82.          this.setLookbackWindow(lookbackWindow);
83.
84.          /*
85.           * Calculate volatility index values based on the base value and set
86.           * it
87.           */
88.          ValueDateTupel[] calculatedVolatilityIndices =
89.              calculateVolatilityIndices(baseValue, lookbackWindow);
90.          this.validateVolatilityIndices(calculatedVolatilityIndices);
91.          this.setVolatilityIndices(calculatedVolatilityIndices);
92.        }
93.      }
94.
95.      /**
96.       * Creates a new VolatilityDifference instance using the passed
97.       * {@link BaseValue} to calculate the volatility indices and the average
98.       * volatility.
99.       *
100.      * @param baseValue              Same as in
101.      *                                 {@link Rule#Rule(BaseValue, Rule[],
102.      *                                 LocalDateTime, LocalDateTime, double)}.
103.      * @param variations             {@code VolatilityDifference[]} An array
```

```
104.      *                                of three or less rules. Represents the
105.      *                                variations of this rule. Same
106.      *                                limitations as in
107.      *                                {@link Rule#Rule(BaseValue, Rule[],
108.      *                                LocalDateTime, LocalDateTime, double)}.
109.      * @param startOfReferenceWindow Same as in
110.      *                                {@link Rule#Rule(BaseValue, Rule[],
111.      *                                LocalDateTime, LocalDateTime, double)}.
112.      * @param endOfReferenceWindow   Same as in
113.      *                                {@link Rule#Rule(BaseValue, Rule[],
114.      *                                LocalDateTime, LocalDateTime, double)}.
115.      * @param lookbackWindow         {@code int} The lookback window to be
116.      *                                used for this VolatilityDifference. See
117.      *                                {@link #validateLookbackWindow(int)} for
118.      *                                limitations.
119.      * @param baseScale              Same as in
120.      *                                {@link Rule#Rule(BaseValue, Rule[],
121.      *                                LocalDateTime, LocalDateTime, double)}.
122.      * @param volatilityIndices      {@code ValueDateTupel[]} The volatility
123.      *                                indices used for forecast calculations.
124.      *                                See
125.      *                                {@link #validateVolatilityIndices(
126.      *                                ValueDateTupel[])}
127.      *                                for limitations.
128.      */
129.     public VolatilityDifference(BaseValue baseValue,
130.         VolatilityDifference[] variations,
131.         LocalDateTime startOfReferenceWindow,
132.         LocalDateTime endOfReferenceWindow, int lookbackWindow,
133.         double baseScale, ValueDateTupel[] volatilityIndices) {
134.       super(baseValue, variations, startOfReferenceWindow,
135.           endOfReferenceWindow, baseScale);
136.
137.       if (variations == null) {
138.         /*
139.          * Lookback window and volatility indices are only needed when a rule
140.          * has no variations.
141.          */
142.         /* Check if lookback window fulfills requirements. */
143.         validateLookbackWindow(lookbackWindow);
144.         this.setLookbackWindow(lookbackWindow);
145.
146.         /*
147.          * Calculate volatility index values based on the base value and set
148.          * it
149.          */
150.         this.validateVolatilityIndices(volatilityIndices);
151.         this.setVolatilityIndices(volatilityIndices);
152.       }
153.     }
154.
155.     /**
156.      * Calculates the raw forecast by subtracting the forecast for the given
157.      * LocalDateTIme from the average volatility at that same given point in
```

```
158.      * time. Positive results result in a positive forecast.
159.      */
160.     @Override
161.     double calculateRawForecast(LocalDateTime forecastDateTime) {
162.       double currentVolatilty = ValueDateTupel
163.           .getElement(this.getVolatilityIndices(), forecastDateTime)
164.           .getValue();
165.       return calculateAverageVolatility(forecastDateTime) - currentVolatilty;
166.     }
167.
168.     /**
169.      * Calculate the volatility index values for this VolatilityDifference.
170.      * If the lookback window is longer than there are base values, an empty
171.      * {@code ValueDateTupel[]} is returned.
172.      *
173.      * @return {@code ValueDateTupel[]} calculated volatility indices. If the
174.      *         number of values in the instance base value is smaller than
175.      *         the lookback window the returned array only contains
176.      *         {@code Double.NaN}. Else, all values until the lookback window
177.      *         is reached contain {@code Double.NaN}, the rest contains real
178.      *         volatility index values.
179.      * @throws IllegalArgumentException if the number of base values is
180.      *                                  smaller than the given lookback
181.      *                                  window.
182.      */
183.     private static ValueDateTupel[] calculateVolatilityIndices(
184.         BaseValue baseValue, int lookbackWindow) {
185.       ValueDateTupel[] baseValues = baseValue.getValues();
186.
187.       ValueDateTupel[] volatilityIndices = null;
188.
189.       /**
190.        * If there are less base values than the lookback window is long no
191.        * volatility values can be calculated. The volatility values only have
192.        * any true meaning, when the lookback window is used in its entirety.
193.        */
194.       if (baseValues.length < lookbackWindow)
195.         throw new IllegalArgumentException(
196.             "The amount of base values must not be smaller than the lookback"
197.                 + " window. Number of base values: " + baseValues.length
198.                 + ", lookback window: " + lookbackWindow + ".");
199.
200.       /**
201.        * Fill the spaces before reaching lookbackWindow with NaN
202.        */
203.       for (int i = 0; i < lookbackWindow; i++) {
204.         ValueDateTupel volatilityIndexNaN = new ValueDateTupel(
205.             baseValues[i].getDate(), Double.NaN);
206.         volatilityIndices = ArrayUtils.add(volatilityIndices,
207.             volatilityIndexNaN);
208.       }
209.
210.       /**
211.        * Start calculation with first adequate time value (after lookback
```

```
212.        * window is reached), e.g. lookbackWindow = 4, start with index 2 (4th
213.        * element), as the returns of each day will be needed.
214.        */
215.       for (int i = lookbackWindow; i < baseValues.length; i++) {
216.         /* Copy the relevant values into a temporary array */
217.         ValueDateTupel[] tempBaseValues = ValueDateTupel
218.             .createEmptyArray(lookbackWindow + 1);
219.         System.arraycopy(baseValues, /* source array */
220.             i - (lookbackWindow), /*
221.                                    * source array position (starting position
222.                                    * for copy)
223.                                    */
224.             tempBaseValues, /* destination array */
225.             0, /* destination position (starting position for paste) */
226.             lookbackWindow + 1/* length (number of values to copy */
227.         );
228.
229.         /* Calculate relevant returns and store them into an array. */
230.         double[] tempDoubleValues = {};
231.         for (int j = 1; j < tempBaseValues.length; j++) {
232.           double returnForDayI = Util.calculateReturn(
233.               tempBaseValues[j - 1].getValue(),
234.               tempBaseValues[j].getValue());
235.           tempDoubleValues = ArrayUtils.add(tempDoubleValues, returnForDayI);
236.         }
237.
238.         /* Calculate standard deviation and save into local variable */
239.         StandardDeviation sd = new StandardDeviation();
240.         double volatilityIndexValue = sd.evaluate(tempDoubleValues);
241.
242.         ValueDateTupel volatilityIndexValueDateTupel = new ValueDateTupel(
243.             baseValues[i].getDate(), volatilityIndexValue);
244.
245.         /* Add calculated standard deviation to volatility indices */
246.         volatilityIndices = ArrayUtils.add(volatilityIndices,
247.             volatilityIndexValueDateTupel);
248.       }
249.
250.       return volatilityIndices;
251.     }
252.
253.     /**
254.      * Calculate the average volatility for a given {@link LocalDateTime}.
255.      *
256.      * @param dateToBeCalculatedFor {@link LocalDateTime} The LocalDateTime
257.      *                              the average volatility is to be
258.      *                              calculated for.
259.      * @return {@code double} The average volatility up until the given
260.      *         LocalDateTime.
261.      */
262.     private double calculateAverageVolatility(
263.         LocalDateTime dateToBeCalculatedFor) {
264.       /*
265.        * Starting point is the first DateTime that exceeds the lookback
```

```
266.        * window.
267.        */
268.       LocalDateTime startingDateTime = this.getVolatilityIndices()[this
269.           .getLookbackWindow()].getDate();
270.
271.       /* Get all relevant volatility index values */
272.       ValueDateTupel[] relevantVolatilityIndices = ValueDateTupel
273.           .getElements(this.getVolatilityIndices(), startingDateTime,
274.               dateToBeCalculatedFor);
275.
276.       DoubleSummaryStatistics stats = new DoubleSummaryStatistics();
277.       /* Extract all relevant values into statistics object */
278.       for (ValueDateTupel volatilityIndex : relevantVolatilityIndices)
279.         stats.accept(volatilityIndex.getValue());
280.
281.       /* Put average value of relevant values into class variable */
282.       return stats.getAverage();
283.     }
284.
285.     /**
286.      * Validates the given lookback window. The lookback window must be
287.      * greater than or equal to 1.
288.      *
289.      * @param lookbackWindow {@code int} The lookback window to be validated.
290.      * @throws IllegalArgumentException if the given lookbackWindow does not
291.      *                                  meet specifications
292.      */
293.     public static void validateLookbackWindow(int lookbackWindow) {
294.       if (lookbackWindow <= 1)
295.         throw new IllegalArgumentException(
296.             "Lookback window must be at least 2");
297.     }
298.
299.     /**
300.      * Validates the given volatility indices.
301.      *
302.      * @param volatilityIndices {@code ValueDateTupel[]} the array of
303.      *                          volatility indices to be validated. Must
304.      *                          comply as follows:
305.      *                          <ul>
306.      *                          <li>Must not be null.</li>
307.      *                          <li>Must not be an empty array.</li>
308.      *                          <li>Must be sorted in ascending order.</li>
309.      *                          <li>Must contain this instances
310.      *                          startOfReferenceWindow and
311.      *                          endOfReferenceWindow.</li>
312.      *                          <li>Must not contain Double.NaN values</li>
313.      *                          <li>Must be aligned with this instance's base
314.      *                          value's values. See
315.      *                          {@link ValueDateTupel#
316.      *                          alignDates(ValueDateTupel[][])}.</li>
317.      *                          </ul>
318.      * @throws IllegalArgumentException if the given volatility indices do
319.      *                                  not meet specifications.
```

```
320.      */
321.     private void validateVolatilityIndices(
322.         ValueDateTupel[] volatilityIndices) {
323.       /* Check if passed volatility indices are null */
324.       if (volatilityIndices == null)
325.         throw new IllegalArgumentException(
326.             "Volatility indices must not be null.");
327.
328.       /* Check if passed values array contains elements */
329.       if (volatilityIndices.length == 0)
330.         throw new IllegalArgumentException(
331.             "Volatility indices must not be an empty array");
332.
333.       /*
334.        * The values cannot be used if they are not in ascending order or if
335.        * they contain duplicate LocalDateTimes.
336.        */
337.       if (!ValueDateTupel.isSortedAscending(volatilityIndices))
338.         throw new IllegalArgumentException(
339.             "Given volatility indices are not properly sorted or there are"
340.                 + " duplicate LocalDateTime values");
341.
342.       try {
343.         Validator.validateTimeWindow(this.getStartOfReferenceWindow(),
344.             this.getEndOfReferenceWindow(), volatilityIndices);
345.       } catch (IllegalArgumentException e) {
346.         throw new IllegalArgumentException(
347.             "Giving volatility indices do not meet specificiation.", e);
348.       }
349.
350.       /*
351.        * The given volatility indices value must not contain NaNs in the area
352.        * delimited by startOfReferenceWindow and endOfReferenceWindow.
353.        */
354.       int startOfReferencePosition = ValueDateTupel
355.           .getPosition(volatilityIndices, this.getStartOfReferenceWindow());
356.       int endOfReferencePosition = ValueDateTupel
357.           .getPosition(volatilityIndices, this.getEndOfReferenceWindow());
358.
359.       for (int i = startOfReferencePosition;
360.           i <= endOfReferencePosition;
361.           i++) {
362.         if (Double.isNaN(volatilityIndices[i].getValue())) {
363.           throw new IllegalArgumentException(
364.               "There must not be NaN-Values in the given volatility indices "
365.                   + "values in the area delimited by startOfReferenceWindow"
366.                   + " and endOfReferenceWindow");
367.         }
368.       }
369.
370.       /*
371.        * Extract dates out of the base value's values array and add it to a
372.        * HashSet
373.        */
```

```
374.      Set<LocalDateTime> baseValueSet = new HashSet<>();
375.      for (ValueDateTupel value : this.getBaseValue().getValues())
376.        baseValueSet.add(value.getDate());
377.      /*
378.       * Extract dates out of the volatility indices values array and add
379.       * them to to a copy of the same HashSet
380.       */
381.      Set<LocalDateTime> volatilityIndicesSet = new HashSet<>(baseValueSet);
382.      for (ValueDateTupel value : volatilityIndices)
383.        volatilityIndicesSet.add(value.getDate());
384.      /*
385.       * If the Sets differ in length, the volatility indices added unique
386.       * LocalDateTimes to the set. Therefore, the both are not properly
387.       * aligned.
388.       */
389.      if (baseValueSet.size() != volatilityIndicesSet.size())
390.        throw new IllegalArgumentException(
391.            "Base value and volatility index values are not properly"
392.                + " aligned. Utilize"
393.                + " ValueDateTupel.alignDates(ValueDateTupel[][])"
394.                + " before creating a new VolatilityDifference.");
395.    }
396.
397.    /**
398.     * ========================================================================
399.     * OVERRIDES
400.     * ========================================================================
401.     */
402.
403.    /**
404.     * A hash code for this VolatilityDifference.
405.     */
406.    @GeneratedCode
407.    @Override
408.    public int hashCode() {
409.      final int prime = 31;
410.      int result = super.hashCode();
411.      result = prime * result + lookbackWindow;
412.      result = prime * result + Arrays.hashCode(volatilityIndices);
413.      return result;
414.    }
415.
416.    /**
417.     * Checks if this VolatilityDifference is equal to another
418.     * VolatilityDifference.
419.     */
420.    @GeneratedCode
421.    @Override
422.    public boolean equals(Object obj) {
423.      if (this == obj)
424.        return true;
425.      if (!super.equals(obj))
426.        return false;
427.      if (getClass() != obj.getClass())
```

```
428.         return false;
429.       VolatilityDifference other = (VolatilityDifference) obj;
430.       if (lookbackWindow != other.lookbackWindow)
431.         return false;
432.       if (!Arrays.equals(volatilityIndices, other.volatilityIndices))
433.         return false;
434.       return true;
435.     }
436.
437.     /**
438.      * Outputs the fields of this VolatilityDifference as a {@code String}.
439.      */
440.     @GeneratedCode
441.     @Override
442.     public String toString() {
443.       StringBuilder builder = new StringBuilder();
444.       builder.append("VolatilityDifference [volatilityIndices=");
445.       builder.append(Arrays.toString(volatilityIndices));
446.       builder.append(", lookbackWindow=");
447.       builder.append(lookbackWindow);
448.       builder.append("]");
449.       return builder.toString();
450.     }
451.
452.     /**
453.      * ========================================================================
454.      * GETTERS AND SETTERS
455.      * ========================================================================
456.      */
457.
458.     /**
459.      * Get the volatility indices for this VolatilityDifference.
460.      *
461.      * @return {@code ValueDateTupel[]} The volatility indices for this
462.      *         VolatilityDifference
463.      */
464.     public ValueDateTupel[] getVolatilityIndices() {
465.       return volatilityIndices;
466.     }
467.
468.     /**
469.      * Set the volatilityIndices
470.      *
471.      * @param volatilityIndices {@code ValueDateTupel[]} the
472.      *                          volatilityIndices to set
473.      */
474.     private void setVolatilityIndices(ValueDateTupel[] volatilityIndices) {
475.       this.volatilityIndices = volatilityIndices;
476.     }
477.
478.     /**
479.      * Get the lookbackWindow for this VolatilityDifference.
480.      *
481.      * @return {@code int} The lookback window for this VolatilityDifference
```

```
482.      */
483.     public int getLookbackWindow() {
484.       return lookbackWindow;
485.     }
486.
487.     /**
488.      * Set the lookback window for this VolatilityDifference.
489.      *
490.      * @param lookbackWindow {@code int} The lookback window to be set for
491.      *                       this VolatilityDifference
492.      */
493.     private void setLookbackWindow(int lookbackWindow) {
494.       this.lookbackWindow = lookbackWindow;
495.     }
496.   }
```

Komponente EWMAC

Listing 15: Komponente EWMAC

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.time.LocalDateTime;
4.
5.  import de.rumford.tradingsystem.helper.GeneratedCode;
6.  import de.rumford.tradingsystem.helper.ValueDateTupel;
7.
8.  /**
9.   * The EWMAC is a non-binary {@link Rule} utilizing the different horizons
10.  * of its 2 underlying {@link EWMA}. When the shorter horizon EWMA raises
11.  * above the longer horizon EWMA in value the underlying asset has been
12.  * rising in the not-so-distant past and is thus expected to rise further,
13.  * and vice-versa.
14.  *
15.  * @author Max Rumford
16.  *
17.  */
18. public class EWMAC extends Rule {
19.
20.   /* The longer horizon EWMA. */
21.   private EWMA longHorizonEwma;
22.   /* The shorter horizon EWMA. */
23.   private EWMA shortHorizonEwma;
24.
25.   /**
26.    * A rule based on the relation between two exponentially weighted moving
27.    * averages - EWMAs. The EWMAC assumes that an asset is rising in value,
28.    * if its short term values are greater than its long term values. If the
29.    * short horizon EWMA is greater than the long horizon one, a positive
30.    * forecast is given. Else, a negative forecast is produced.
31.    * <p>
32.    * This non-binary rule allows for a base value scale independent
33.    * detailed forecast calculation. The greater the difference between the
34.    * two EWMAs the greater in extent the forecast will be.
35.    *
36.    * @param baseValue               Same as in
37.    *                                {@link Rule#Rule(BaseValue, Rule[],
38.    *                                LocalDateTime, LocalDateTime, double)}.
39.    * @param variations              {@code EWMAC[]} An array of three or
40.    *                                less rules. Represents the variations of
41.    *                                this rule. Same limitations as in
42.    *                                {@link Rule#Rule(BaseValue, Rule[],
43.    *                                LocalDateTime, LocalDateTime, double)}.
44.    * @param startOfReferenceWindow Same as in
45.    *                                {@link Rule#Rule(BaseValue, Rule[],
46.    *                                LocalDateTime, LocalDateTime, double)}.
47.    * @param endOfReferenceWindow   Same as in
48.    *                                {@link Rule#Rule(BaseValue, Rule[],
49.    *                                LocalDateTime, LocalDateTime, double)}.
```

```
50.       * @param longHorizon            {@code int} The horizon of the long
51.       *                               horizon EWMA. Should be 4* shortHorizon,
52.       *                               but can be anything greater than
53.       *                               shortHorizon.
54.       * @param shortHorizon           {@code int} The horizon of the short
55.       *                               horizon EWMA. Must be greater or equal
56.       *                               to 2;
57.       * @param baseScale              Same as in
58.       *                               {@link Rule#Rule(BaseValue, Rule[],
59.       *                               LocalDateTime, LocalDateTime, double)}.
60.       */
61.      public EWMAC(BaseValue baseValue, EWMAC[] variations,
62.          LocalDateTime startOfReferenceWindow,
63.          LocalDateTime endOfReferenceWindow, int longHorizon,
64.          int shortHorizon, double baseScale) {
65.        super(baseValue, variations, startOfReferenceWindow,
66.            endOfReferenceWindow, baseScale);
67.
68.        this.validateHorizonValues(longHorizon, shortHorizon);
69.
70.        if (variations == null) {
71.          EWMA localLongHorizonEwma = new EWMA(this.getBaseValue().getValues(),
72.              longHorizon);
73.          EWMA localShortHorizonEwma = new EWMA(
74.              this.getBaseValue().getValues(), shortHorizon);
75.          this.setLongHorizonEwma(localLongHorizonEwma);
76.          this.setShortHorizonEwma(localShortHorizonEwma);
77.        }
78.      }
79.
80.      /**
81.       * Calculates the raw forecast for a given LocalDateTime by subtracting
82.       * the long horizon EWMA value from the short horizon EWMA value for this
83.       * LocalDateTime.
84.       */
85.      @Override
86.      double calculateRawForecast(LocalDateTime forecastDateTime) {
87.        double longHorizonEwmaValue = ValueDateTupel
88.            .getElement(this.getLongHorizonEwma().getEwmaValues(),
89.                forecastDateTime)
90.            .getValue();
91.        double shortHorizonEwmaValue = ValueDateTupel
92.            .getElement(this.getShortHorizonEwma().getEwmaValues(),
93.                forecastDateTime)
94.            .getValue();
95.
96.        return shortHorizonEwmaValue - longHorizonEwmaValue;
97.      }
98.
99.      /**
100.      * Validate the given longHorizon and shortHorizon values.
101.      *
102.      * @param longHorizon  {@code int} The given long horizon. Must be >
103.      *                     shortHorizon.
```

```
     * @param shortHorizon {@code int} The given short horizon. Must be >= 2.
     * @throws IllegalArgumentException if the above specifications are not
     *                                  met.
     */
    private void validateHorizonValues(int longHorizon, int shortHorizon) {

      /* The horizons are not used when this rule has variations. */
      if (this.getVariations() == null) {
        if (longHorizon <= shortHorizon)
          throw new IllegalArgumentException(
              "The long horizon must be greater than the short horizon");

        if (shortHorizon < 2)
          throw new IllegalArgumentException(
              "The short horizon must not be < 2");
      }
    }

    /**
     * ========================================================================
     * OVERRIDES
     * ========================================================================
     */
    /**
     * A hash code for this EWMAC.
     */
    @GeneratedCode
    @Override
    public int hashCode() {
      final int prime = 31;
      int result = super.hashCode();
      result = prime * result
          + ((longHorizonEwma == null) ? 0 : longHorizonEwma.hashCode());
      result = prime * result
          + ((shortHorizonEwma == null) ? 0 : shortHorizonEwma.hashCode());
      return result;
    }

    /**
     * Checks if this EWMAC is equal to another EWMAC.
     */
    @GeneratedCode
    @Override
    public boolean equals(Object obj) {
      if (this == obj)
        return true;
      if (!super.equals(obj))
        return false;
      if (getClass() != obj.getClass())
        return false;
      EWMAC other = (EWMAC) obj;
      if (longHorizonEwma == null) {
        if (other.longHorizonEwma != null)
          return false;
```

```
158.      } else if (!longHorizonEwma.equals(other.longHorizonEwma))
159.        return false;
160.      if (shortHorizonEwma == null) {
161.        if (other.shortHorizonEwma != null)
162.          return false;
163.      } else if (!shortHorizonEwma.equals(other.shortHorizonEwma))
164.        return false;
165.      return true;
166.    }
167.
168.    /**
169.     * Outputs the fields of this EWMAC as a {@code String}.
170.     */
171.    @GeneratedCode
172.    @Override
173.    public String toString() {
174.      StringBuilder builder = new StringBuilder();
175.      builder.append("EWMAC [longHorizonEwma=");
176.      builder.append(longHorizonEwma);
177.      builder.append(", shortHorizonEwma=");
178.      builder.append(shortHorizonEwma);
179.      builder.append("]");
180.      return builder.toString();
181.    }
182.
183.    /**
184.     * ========================================================================
185.     * GETTERS AND SETTERS
186.     * ========================================================================
187.     */
188.
189.    /**
190.     * Get the long horizon {@link EWMA} for an {@link EWMAC}
191.     *
192.     * @return long horizon {@link EWMA} for this {@link EWMAC}
193.     */
194.    public EWMA getLongHorizonEwma() {
195.      return longHorizonEwma;
196.    }
197.
198.    /**
199.     * Set the long horizon {@link EWMA} for an {@link EWMAC}
200.     *
201.     * @param longHorizonEwma {@link EWMA} long horizon the be set for this
202.     *                        {@link EWMAC}
203.     */
204.    private void setLongHorizonEwma(EWMA longHorizonEwma) {
205.      this.longHorizonEwma = longHorizonEwma;
206.    }
207.
208.    /**
209.     * Get the short horizon {@link EWMA} for an {@link EWMAC}
210.     *
211.     * @return short horizon {@link EWMA} for this {@link EWMAC}
```

```
212.      */
213.     public EWMA getShortHorizonEwma() {
214.       return shortHorizonEwma;
215.     }
216.
217.     /**
218.      * Set the short horizon {@link EWMA} for an {@link EWMAC}
219.      *
220.      * @param shortHorizonEwma short horizon {@link EWMA} to be set for this
221.      *                         {@link EWMAC}
222.      */
223.     private void setShortHorizonEwma(EWMA shortHorizonEwma) {
224.       this.shortHorizonEwma = shortHorizonEwma;
225.     }
226.   }
```

Komponente EWMA

Listing 16: Komponente EWMA

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.util.Arrays;
4.
5.  import org.apache.commons.lang3.ArrayUtils;
6.
7.  import de.rumford.tradingsystem.helper.GeneratedCode;
8.  import de.rumford.tradingsystem.helper.Validator;
9.  import de.rumford.tradingsystem.helper.ValueDateTupel;
10.
11. /**
12.  * The EWMA class represents the mathematical concept of an exponentially
13.  * weighted moving average. In an EWMA, the "older" a given base value is
14.  * (in proportion to the "current" base value) the less it influences the
15.  * current EWMA value. This influence deteriorates exponentially by the
16.  * given horizon to the power of 2, thus the name.
17.  *
18.  * @author Max Rumford
19.  *
20.  */
21. public class EWMA {
22.
23.   /* The horizon this EWMA shall cover. */
24.   private int horizon;
25.   /*
26.    * The decay factor for recent values calculated from the given horizon.
27.    */
28.   private double decay;
29.   /* The values this EWMA shall be based upon. */
30.   private ValueDateTupel[] baseValues;
31.   /* The calculated EWMA values. */
32.   private ValueDateTupel[] ewmaValues;
33.
34.   /**
35.    * Constructor for the {@link EWMA} class
36.    *
37.    * @param baseValues {@code ValueDateTupel[]} The values this EWMA shall
38.    *                   is to be based on.
39.    * @param horizon    {@code int} horizon this EWMA is to be over
40.    */
41.   public EWMA(ValueDateTupel[] baseValues, int horizon) {
42.     validateBaseValues(baseValues);
43.     validateHorizon(horizon);
44.
45.     this.setBaseValues(baseValues);
46.     this.setHorizon(horizon);
47.     this.setDecay(this.calculateDecay(this.getHorizon()));
48.     this.setEwmaValues(this.calculateEwmaValues(this.getBaseValues()));
49.   }
```

```
50.
51.     /**
52.      * Calculate the decay value based on the given horizon.
53.      *
54.      * @param horizon {@code int} Horizon of this EWMA.
55.      * @return {@code double} the decay used to calculate the importance of
56.      *         the previous EWMA.
57.      */
58.     private double calculateDecay(int horizon) {
59.       return 2d / (horizon + 1d);
60.     }
61.
62.     /**
63.      * Calculate the EWMA-value for given previous value and base value
64.      *
65.      * @param previousEWMA {@code double} EWMA of the previous time period
66.      * @param baseValue    {@code double} base value of the current time
67.      *                     period
68.      * @return {@code double} EWMA for the current time period
69.      */
70.     public double calculateEWMA(double previousEWMA, double baseValue) {
71.       /* E_t = A * P_t + [E_t-1 * ( 1 - A ) ] */
72.       return this.getDecay() * baseValue
73.           + (previousEWMA * (1d - this.getDecay()));
74.     }
75.
76.     /**
77.      * Calculate the EWMA values based on the given base values.
78.      *
79.      * @param baseValues {@code ValueDateTupel[]} The base values of the
80.      *                   given asset.
81.      * @return {@code ValueDateTupel[]} An array of calculated EWMA values.
82.      */
83.     private ValueDateTupel[] calculateEwmaValues(
84.         ValueDateTupel[] baseValues) {
85.       ValueDateTupel[] newEwmaValues = ValueDateTupel.createEmptyArray();
86.       double previousEwma = 0;
87.       /* Calculate all EWMA-Values */
88.       for (ValueDateTupel baseValue : baseValues) {
89.         double newValue = 0;
90.         if (Double.isNaN(baseValue.getValue())) {
91.           newValue = Double.NaN;
92.           previousEwma = 0;
93.         } else {
94.           /* Calculate the new values */
95.           newValue = this.calculateEWMA(previousEwma, baseValue.getValue());
96.           previousEwma = newValue;
97.         }
98.         /* Add the new value to the array of EWMA values */
99.         newEwmaValues = ArrayUtils.add(newEwmaValues,
100.            new ValueDateTupel(baseValue.getDate(), newValue));
101.      }
102.      return newEwmaValues;
103.    }
```

```
104.
105.     /**
106.      * Validates the given base values.
107.      *
108.      * @param baseValues {@code ValueDateTupel[]} the base values the EWMA is
109.      *                   to be calculated on. Must pass
110.      *                   {@link Validator#validateValues(ValueDateTupel[])}
111.      *                   and
112.      *                   {@link Validator#validateDates(ValueDateTupel[])}.
113.      * @throws IllegalArgumentException if the above specifications are not
114.      *                                  met.
115.      */
116.     private static void validateBaseValues(ValueDateTupel[] baseValues) {
117.       try {
118.         Validator.validateValues(baseValues);
119.         Validator.validateDates(baseValues);
120.       } catch (IllegalArgumentException e) {
121.         throw new IllegalArgumentException(
122.             "The given values do not meet the specifications.", e);
123.       }
124.     }
125.
126.     /**
127.      * Validates the given horizon.
128.      *
129.      * @param horizon {@code int} the horizon to be validated.
130.      * @throws IllegalArgumentException if the given horizon is < 2.
131.      */
132.     private static void validateHorizon(int horizon) {
133.       if (horizon < 2)
134.         throw new IllegalArgumentException("The horizon must not be < 2");
135.     }
136.
137.     /**
138.      * ========================================================================
139.      * OVERRIDES
140.      * ========================================================================
141.      */
142.
143.     /**
144.      * A hash code for this EWMA.
145.      */
146.     @GeneratedCode
147.     @Override
148.     public int hashCode() {
149.       final int prime = 31;
150.       int result = 1;
151.       result = prime * result + horizon;
152.       return result;
153.     }
154.
155.     /**
156.      * Checks if this EWMA is equal to another EWMA.
157.      */
```

```
158.     @GeneratedCode
159.     @Override
160.     public boolean equals(Object obj) {
161.       if (this == obj)
162.         return true;
163.       if (obj == null)
164.         return false;
165.       if (getClass() != obj.getClass())
166.         return false;
167.       EWMA other = (EWMA) obj;
168.       if (horizon != other.horizon)
169.         return false;
170.       return true;
171.     }
172.
173.     /**
174.      * Outputs the fields of this EWMA as a {@code String}.
175.      */
176.     @GeneratedCode
177.     @Override
178.     public String toString() {
179.       StringBuilder builder = new StringBuilder();
180.       builder.append("EWMA [horizon=");
181.       builder.append(horizon);
182.       builder.append(", decay=");
183.       builder.append(decay);
184.       builder.append(", baseValues=");
185.       builder.append(Arrays.toString(baseValues));
186.       builder.append(", ewmaValues=");
187.       builder.append(Arrays.toString(ewmaValues));
188.       builder.append("]");
189.       return builder.toString();
190.     }
191.
192.     /**
193.      * ========================================================================
194.      * GETTERS AND SETTERS
195.      * ========================================================================
196.      */
197.
198.     /**
199.      * Get the horizon of this EWMA.
200.      *
201.      * @return {@code int} horizon of the EWMA
202.      */
203.     public int getHorizon() {
204.       return horizon;
205.     }
206.
207.     /**
208.      * Set the horizon of this EWMA.
209.      *
210.      * @param horizon {@code int} horizon to be set
211.      */
```

```
212.    public void setHorizon(int horizon) {
213.      this.horizon = horizon;
214.    }
215.
216.    /**
217.     * Get the decay of this EWMA.
218.     *
219.     * @return {@code double} decay of the EWMA
220.     */
221.    public double getDecay() {
222.      return this.decay;
223.    }
224.
225.    /**
226.     * Set the decay of this EWMA.
227.     *
228.     * @param horizon {@code int} horizon on which the decay is derived from
229.     */
230.    private void setDecay(double decay) {
231.      this.decay = decay;
232.    }
233.
234.    /**
235.     * Get the base values of this EWMA.
236.     *
237.     * @return baseValues EWMA
238.     */
239.    public ValueDateTupel[] getBaseValues() {
240.      return baseValues;
241.    }
242.
243.    /**
244.     * Set the base values of this EWMA.
245.     *
246.     * @param baseValues the baseValues to set
247.     */
248.    private void setBaseValues(ValueDateTupel[] baseValues) {
249.      this.baseValues = baseValues;
250.    }
251.
252.    /**
253.     * Get the EWMA values of this EWMA.
254.     *
255.     * @return ewmaValues EWMA
256.     */
257.    public ValueDateTupel[] getEwmaValues() {
258.      return ewmaValues;
259.    }
260.
261.    /**
262.     * Set the EWMA values of this EWMA.
263.     *
264.     * @param ewmaValues the ewmaValues to set
265.     */
```

```
266.     private void setEwmaValues(ValueDateTupel[] ewmaValues) {
267.       this.ewmaValues = ewmaValues;
268.     }
269.   }
```

Komponente BaseValue

Listing 17: Komponente BaseValue

```
1.    package de.rumford.tradingsystem;
2.
3.    import java.util.Arrays;
4.
5.    import org.apache.commons.lang3.ArrayUtils;
6.
7.    import de.rumford.tradingsystem.helper.GeneratedCode;
8.    import de.rumford.tradingsystem.helper.Util;
9.    import de.rumford.tradingsystem.helper.Validator;
10.   import de.rumford.tradingsystem.helper.ValueDateTupel;
11.
12.   /**
13.    * The BaseValue is a substantial part for every trading system and
14.    * provides the values to be decided upon by the rules. It encapsulates the
15.    * underlying value, e.g. a stocks tracker, and represents its values as an
16.    * array of {@link ValueDateTupel}. This are guaranteed to be in ascending
17.    * order and free of duplicates.
18.    * <p>
19.    * Each BaseValue has two final static values that cannot be changed and
20.    * are deemed to preserve comparability of base values. The first such
21.    * value is the lookback window. It indicates the amount of values relevant
22.    * (enough) for standard deviation calculation. A lookback window of 25, as
23.    * implemented, deems all values "older" than 25 time intervals (as stated
24.    * by the given ValueDateTupels) irrelevant for standard deviation
25.    * calculation. As a matter of fact, these older values are still
26.    * considered in standard deviation calculation, but the factor they are
27.    * being multiplied with (due to the recursive calculation of the standard
28.    * deviation value) is being considered as too small to actually make a
29.    * noticable difference.
30.    * <p>
31.    * The second predefined value is the short index initial value. If no
32.    * array representing an adequate short index is given into the
33.    * constructor, an array of such values is calculated by subtracting the
34.    * returns (percentage wise) between two time intervals from the previous
35.    * short index value. The short index initial value simply marks the value
36.    * to be set for the first time interval. Its value does not play any role
37.    * in further calculations, as proportions will remain unaltered, no matter
38.    * the actual initial value.
39.    *
40.    * @author Max Rumford
41.    *
42.    */
43.   public class BaseValue {
44.
45.     /* Factor used in the lookback window for standard deviation */
46.     private static final int LOOKBACK_WINDOW = 25;
47.     /* Starting value for the short index values if no values are provided */
48.     private static final double SHORT_INDEX_INITIAL_VALUE = 1000d;
49.
```

```
50.     /* Name to identify an instance. Has no effect. */
51.     private String name;
52.     /* The values upon which the calculations shall take place. */
53.     private ValueDateTupel[] values;
54.     /*
55.      * An array of values representing the short index values to the given
56.      * values.
57.      */
58.     private ValueDateTupel[] shortIndexValues;
59.     /* An array of values representing the standard deviation values. */
60.     private ValueDateTupel[] standardDeviationValues;
61.
62.     /**
63.      * Creates a new {@link BaseValue} instance using the passed
64.      * {@code String} for identification and stores the passed array of
65.      * {@link ValueDateTupel} as values. Short index values are calculated
66.      * based on the given values as specified in
67.      * {@link BaseValue#calculateShortIndexValues(ValueDateTupel[])}.
68.      *
69.      * @param name   {@code String} Name used to identify the represented
70.      *               base value. Is not used for calculation of any kind.
71.      *               Must be of length greater than {@code 0}.
72.      * @param values {@code ValueDateTupel[]} Values of the represented base
73.      *               value. Must not be null. Must be of length greater than
74.      *               {@code 0}. Must be in an ascending order. Must not
75.      *               contain nulls. Must not contain values of Double.NaN.
76.      * @throws IllegalArgumentException if the input values are not within
77.      *                                  specification
78.      */
79.     public BaseValue(String name, ValueDateTupel[] values) {
80.       validateInput(name, values);
81.
82.       this.setName(name);
83.       this.setValues(values);
84.
85.       this.setShortIndexValues(calculateShortIndexValues(values));
86.       this.setStandardDeviationValues(
87.           calculateStandardDeviationValues(values));
88.     }
89.
90.     /**
91.      * Creates a new {@link BaseValue} instance using the passed
92.      * {@code String} for identification and stores the passed array of
93.      * {@link ValueDateTupel} as values and the second passed array of
94.      * {@link ValueDateTupel} as shortIndexValues.
95.      *
96.      * @param name              {@code String} Name used to identify the
97.      *                          represented base value. Fulfills no purpose
98.      *                          and is not used for calculation of any kind.
99.      *                          Must be of length greater than {@code 0}.
100.     * @param values            {@code ValueDateTupel[]} Values of the
101.     *                          represented base value. Must not be null. Must
102.     *                          be of length greater than {@code 0}. Must be
103.     *                          in an ascending order. Must not contain nulls.
```

```
104.      *                           Must not contain values of Double.NaN.
105.      * @param shortIndexValues {@code ValueDateTupel[]} Short index values of
106.      *                           the represented base value. Must not be null.
107.      *                           Must be of length greater than {@code 0}. Must
108.      *                           be in an ascending order. Must not contain
109.      *                           nulls. Must not contain values of Double.NaN.
110.      * @throws IllegalArgumentException if the input values are not within
111.      *                                  specification
112.      */
113.     public BaseValue(String name, ValueDateTupel[] values,
114.         ValueDateTupel[] shortIndexValues) {
115.       this(name, values);
116.
117.       try {
118.         Validator.validateValues(shortIndexValues);
119.         Validator.validateDates(shortIndexValues);
120.       } catch (Exception e) {
121.         throw new IllegalArgumentException(
122.             "Given short index values do not meet the specifications.", e);
123.       }
124.
125.       ValueDateTupel[][] valuesAndShortIndexValues = { values,
126.           shortIndexValues };
127.
128.       ValueDateTupel[][] alignedValuesAndShortIndexValues = ValueDateTupel
129.           .alignDates(valuesAndShortIndexValues);
130.       this.setValues(alignedValuesAndShortIndexValues[0]);
131.       this.setShortIndexValues(alignedValuesAndShortIndexValues[1]);
132.     }
133.
134.     /**
135.      * Calculates the short index values corresponding with a list of given
136.      * values. The initial value is set to be {@code 1000}. The short index
137.      * decreases by the same percentage the base value increases. If the base
138.      * value increases by {@code 10%}, the short index decreases by
139.      * {@code 10%} and vice versa.
140.      *
141.      * <p>
142.      * The short index value is calculated as follows:
143.      * {@code v_s,t = v_s,t-1 - v_s,t-1 *
144.      * return_t,t-1}. Variables:
145.      * </p>
146.      *
147.      * <ul>
148.      * <li>{@code v_s,t} = short value on time interval {@code t}</li>
149.      * <li>{@code return_t,t-1} = returns in the base value between two
150.      * consecutive time intervals</li>
151.      * </ul>
152.      *
153.      * <p>
154.      * If return of the base value exceeds 50% the return used to calculate
155.      * the short index value is floored to 50%.
156.      *
157.      * @param values {@codeValueDateTupel[]} values to base the short index
```

```
158.      *               values on
159.      * @return {@code ValueDateTupel[]} array of short index values
160.      * @throws IllegalArgumentException if the passed values array contains
161.      *                                  no elements
162.      */
163.     private static ValueDateTupel[] calculateShortIndexValues(
164.         ValueDateTupel[] values) {
165.       /**
166.        * Declare the return value. There are always as many short index
167.        * values as there are base values.
168.        */
169.       ValueDateTupel[] calculatedShortIndexValues = ValueDateTupel
170.           .createEmptyArray(values.length);
171.       calculatedShortIndexValues[0] = new ValueDateTupel(values[0].getDate(),
172.           SHORT_INDEX_INITIAL_VALUE);
173.
174.       ValueDateTupel formerValue;
175.       ValueDateTupel latterValue;
176.
177.       /**
178.        * Loop over the provided values array and calculate the corresponding
179.        * short index value for every time interval t > 0.
180.        */
181.       for (int i = 1; i < values.length; i++) {
182.         formerValue = values[i - 1];
183.         latterValue = values[i];
184.
185.         double returnPercentagePoints = Util
186.             .calculateReturn(formerValue.getValue(), latterValue.getValue());
187.
188.         /**
189.          * If the base value generates more than 50% in returns (and thus
190.          * decreasing the short index value by more than 50%) the return
191.          * percentage is set to 50%.
192.          */
193.         if (returnPercentagePoints > 0.5)
194.           returnPercentagePoints = 0.5;
195.
196.         double shortIndexValue = calculatedShortIndexValues[i - 1].getValue()
197.             - calculatedShortIndexValues[i - 1].getValue()
198.                 * returnPercentagePoints;
199.
200.         calculatedShortIndexValues[i] = new ValueDateTupel(
201.             latterValue.getDate(), shortIndexValue);
202.       }
203.
204.       return calculatedShortIndexValues;
205.     }
206.
207.     /**
208.      * Calculate the standard deviation values for the given base values. The
209.      * first value is always Double.NaN.
210.      * <p>
211.      * {@code sd = baseValue * sqrt[ EWMA( return^2 ) ]}
```

```
212.      *
213.      * @param baseValues {@code ValueDateTupel[]} the given base values.
214.      * @return {@code ValueDateTupel[]} the calculated standard deviation
215.      *         values.
216.      */
217.     private static ValueDateTupel[] calculateStandardDeviationValues(
218.         ValueDateTupel[] baseValues) {
219.
220.       /* Initiate the squared returns. The first value is always Double.NaN */
221.       ValueDateTupel[] squaredReturns = {};
222.
223.       /* Calculate the squared returns */
224.       for (int i = 0; i < baseValues.length - 1; i++) {
225.         double returns;
226.         returns = Util.calculateReturn(baseValues[i].getValue(),
227.             baseValues[i + 1].getValue());
228.         squaredReturns = ArrayUtils.add(squaredReturns, new ValueDateTupel(
229.             baseValues[i + 1].getDate(), Math.pow(returns, 2)));
230.       }
231.
232.       /* Instantiate the EWMA used for the standard deviation. */
233.       EWMA ewmaOfStandardDeviation = new EWMA(squaredReturns,
234.           LOOKBACK_WINDOW);
235.
236.       /*
237.        * The first value is always Double.NaN, as the first value cannot have
238.        * standard deviation from itself.
239.        */
240.       ValueDateTupel[] standardDeviationValues = {};
241.
242.       /* Fill in the calculated values. */
243.       for (int i = 0; i < squaredReturns.length; i++) {
244.         double squaredEwmaOfVolatility = ewmaOfStandardDeviation
245.             .getEwmaValues()[i].getValue();
246.         double ewmaOfVolatility = Math.sqrt(squaredEwmaOfVolatility);
247.         /*
248.          * The base values array has one more value than the
249.          * standardDeviationValues will have, as there cannot be a standard
250.          * deviation value for the first time interval. The first base value
251.          * will not have a standard deviation value. Therefore to e.g.
252.          * calculate the _first_ sd value, the _second_ base value has to be
253.          * used.
254.          */
255.         double standardDeviation = ewmaOfVolatility
256.             * baseValues[i + 1].getValue();
257.         standardDeviationValues = ArrayUtils.add(standardDeviationValues,
258.             new ValueDateTupel(baseValues[i + 1].getDate(),
259.                 standardDeviation));
260.       }
261.
262.       /* Return the standard deviations. */
263.       return standardDeviationValues;
264.     }
265.
```

```
266.    /**
267.     * Validates the given parameters. Used by the Constructors to validate
268.     * the constructor parameters.
269.     *
270.     * @param name   {@code String} Name to be set for a {@link BaseValue}.
271.     *               Must not be null. Must not have a length of {@code 0}.
272.     * @param values {@code ValueDateTupel[]} Values to be set for a
273.     *               {@link BaseValue}. Must pass
274.     *               {@link Validator#validateValues(ValueDateTupel[])} and
275.     *               {@link Validator#validateDates(ValueDateTupel[])}.
276.     * @throws IllegalArgumentException if one of the above specifications is
277.     *                                  not met.
278.     */
279.    private static void validateInput(String name, ValueDateTupel[] values) {
280.      /* Check if name is null */
281.      if (name == null)
282.        throw new IllegalArgumentException(
283.            "The given name must not be null");
284.
285.      /* Check if name is not empty */
286.      if (name.length() == 0)
287.        throw new IllegalArgumentException(
288.            "Name must not be an empty String");
289.
290.      Validator.validateValues(values);
291.      Validator.validateDates(values);
292.    }
293.
294.    /**
295.     * ========================================================================
296.     * OVERRIDES
297.     * ========================================================================
298.     */
299.
300.    /* A hash code for this base value */
301.    @GeneratedCode
302.    @Override
303.    public int hashCode() {
304.      final int prime = 31;
305.      int result = 1;
306.      result = prime * result + ((name == null) ? 0 : name.hashCode());
307.      result = prime * result + Arrays.hashCode(shortIndexValues);
308.      result = prime * result + Arrays.hashCode(values);
309.      return result;
310.    }
311.
312.    /* Checks if this base value is equal to another base value. */
313.    @GeneratedCode
314.    @Override
315.    public boolean equals(Object obj) {
316.      if (this == obj)
317.        return true;
318.      if (obj == null)
319.        return false;
```

```
320.      if (getClass() != obj.getClass())
321.        return false;
322.      BaseValue other = (BaseValue) obj;
323.      if (name == null) {
324.        if (other.name != null)
325.          return false;
326.      } else if (!name.equals(other.name))
327.        return false;
328.      if (!Arrays.equals(shortIndexValues, other.shortIndexValues))
329.        return false;
330.      if (!Arrays.equals(values, other.values))
331.        return false;
332.      return true;
333.    }
334.
335.    /* Outputs the fields of this base value as a {@code String}. */
336.    @GeneratedCode
337.    @Override
338.    public String toString() {
339.      StringBuilder builder = new StringBuilder();
340.      builder.append("BaseValue [name=");
341.      builder.append(name);
342.      builder.append(", values=");
343.      builder.append(Arrays.toString(values));
344.      builder.append(", shortIndexValues=");
345.      builder.append(Arrays.toString(shortIndexValues));
346.      builder.append(", standardDeviationValues=");
347.      builder.append(Arrays.toString(standardDeviationValues));
348.      builder.append("]");
349.      return builder.toString();
350.    }
351.
352.    /**
353.     * ========================================================================
354.     * GETTERS AND SETTERS
355.     * ========================================================================
356.     */
357.    /**
358.     * Get the name of this {@link BaseValue}
359.     *
360.     * @return name {@code String} of this {@link BaseValue}
361.     */
362.    public String getName() {
363.      return name;
364.    }
365.
366.    /**
367.     * Set the name of this {@link BaseValue}
368.     *
369.     * @param name {@code String} the name to be set in this
370.     *             {@link BaseValue}
371.     */
372.    private void setName(String name) {
373.      this.name = name;
```

```
374.     }
375.
376.     /**
377.      * Get the values of this {@link BaseValue}
378.      *
379.      * @return values {@code ValueDateTupel[]} BaseValue
380.      */
381.     public ValueDateTupel[] getValues() {
382.       return values;
383.     }
384.
385.     /**
386.      * Set the values of this {@link BaseValue}
387.      *
388.      * @param values {@code ValueDateTupel[]} the values to be set
389.      */
390.     private void setValues(ValueDateTupel[] values) {
391.       this.values = values;
392.     }
393.
394.     /**
395.      * Get the shortIndexValues of this {@link BaseValue}
396.      *
397.      * @return shortIndexValues {@code ValueDateTupel[]} shortIndexValues of
398.      *         this {@link BaseValue}
399.      */
400.     public ValueDateTupel[] getShortIndexValues() {
401.       return shortIndexValues;
402.     }
403.
404.     /**
405.      * Set the shortIndexValues of this {@link BaseValue}
406.      *
407.      * @param shortIndexValues {@code ValueDateTupel[]} the shortIndexValues
408.      *                         to be set
409.      */
410.     private void setShortIndexValues(ValueDateTupel[] shortIndexValues) {
411.       this.shortIndexValues = shortIndexValues;
412.     }
413.
414.     /**
415.      * Get the standard deviation values for this base value.
416.      *
417.      * @return standardDeviationValues BaseValue
418.      */
419.     public ValueDateTupel[] getStandardDeviationValues() {
420.       return standardDeviationValues;
421.     }
422.
423.     /**
424.      * Set the standard deviation values for this base value.
425.      *
426.      * @param standardDeviationValues the standardDeviationValues to set
427.      */
```

```
428.     private void setStandardDeviationValues(
429.         ValueDateTupel[] standardDeviationValues) {
430.       this.standardDeviationValues = standardDeviationValues;
431.     }
432.   }
```

Komponente SubSystem

Listing 18: Komponente SubSystem

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.time.LocalDateTime;
4.  import java.time.chrono.ChronoLocalDateTime;
5.  import java.util.Arrays;
6.
7.  import org.apache.commons.lang3.ArrayUtils;
8.
9.  import de.rumford.tradingsystem.helper.GeneratedCode;
10. import de.rumford.tradingsystem.helper.Util;
11. import de.rumford.tradingsystem.helper.Validator;
12. import de.rumford.tradingsystem.helper.ValueDateTupel;
13.
14. /**
15.  * The SubSystem is the most parental Structure in this library and
16.  * contains (directly or indirectly) all other (non-static) classes. It
17.  * combines rules and base values and is thus the only point to calculate
18.  * positions and perform backtesting of performance. By containing a
19.  * {@link DiversificationMultiplier} it copes with the diversity between
20.  * its rules and thus meets volatility target and scale.
21.  *
22.  * @author Max Rumford
23.  *
24.  */
25. public class SubSystem {
26.
27.   /*
28.    * The value to which the product prices shall be scaled to. Effect rises
29.    * if higher and capital goes lower.
30.    */
31.   private static final double PRICE_FACTOR_BASE_SCALE = 1;
32.
33.   /* An Exception message. */
34.   private static final String MESSAGE_ILLEGAL_TEST_WINDOW =
35.       "The given test window does not meet specifications.";
36.
37.   /* The base value to use for performance calculation. */
38.   private BaseValue baseValue;
39.   /* The rules to be governed in this subsystem. */
40.   private Rule[] rules;
41.   /* The diversification multiplier for this subsystem. */
42.   private DiversificationMultiplier diversificationMultiplier;
43.   /* The starting capital used for performance calculation. */
44.   private double capital;
45.   /* The combined forecasts of all rules. */
46.   private ValueDateTupel[] combinedForecasts;
47.   /* The value all forecasts shall be scaled to. */
48.   private double baseScale;
49.
```

```
50.     /**
51.      * Constructor for the SubSystem class.
52.      *
53.      * @param baseValue {@link BaseValue} The base value to be used for all
54.      *                  the given rules' calculations. See
55.      *                  {@link #validateInput( BaseValue, Rule[], double,
56.      *                  double)}
57.      *                  for limitations.
58.      * @param rules     {@code Rule[]} Array of {@link Rule} to be used for
59.      *                  forecast calculations in this SubSystem. See
60.      *                  {@link #validateInput( BaseValue, Rule[], double,
61.      *                  double)}
62.      *                  for limitations.
63.      * @param capital   {@code double} The capital to be managed by this
64.      *                  SubSystem. See
65.      *                  {@link #validateInput( BaseValue, Rule[], double,
66.      *                  double)}
67.      *                  for limitations.
68.      * @param baseScale {@code double} The base scale for this SubSystem's
69.      *                  forecasts. See
70.      *                  {@link #validateInput( BaseValue, Rule[], double,
71.      *                  double)}
72.      *                  for limitations.
73.      */
74.     public SubSystem(BaseValue baseValue, Rule[] rules, double capital,
75.         double baseScale) {
76.
77.       validateInput(baseValue, rules, capital, baseScale);
78.
79.       validateRules(rules);
80.       this.setRules(rules);
81.
82.       this.setBaseValue(baseValue);
83.       this.setCapital(capital);
84.       this.setBaseScale(baseScale);
85.       this.setDiversificationMultiplier(
86.           new DiversificationMultiplier(rules));
87.       this.setCombinedForecasts(this.calculateCombinedForecasts());
88.     }
89.
90.     /**
91.      * Performs a backtest for the given parameters. Utilizes
92.      * {@link #calculatePerformanceValues(BaseValue, LocalDateTime,
93.      * LocalDateTime, ValueDateTupel[], double, double)}
94.      * for actual performance calculation and returns performance value for
95.      * the last day.
96.      *
97.      * @see SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
98.      *      LocalDateTime, ValueDateTupel[], double, double)
99.      *
100.     * @param baseValue         {@link BaseValue} The base value to be tested
101.     *                          against.
102.     * @param startOfTestWindow {@link LocalDateTime} First time interval of
103.     *                          test window.
```

```
104.      * @param endOfTestWindow   {@link LocalDateTime} Last time interval of
105.      *                          test window.
106.      * @param combinedForecasts {@code ValueDateTupel[]} The forecasts to be
107.      *                          used for performance calculation.
108.      * @param baseScale         {@code double} The value to which to scale
109.      *                          the forecasts to.
110.      * @param capital           {@code double} The starting capital.
111.      * @return {@code double} The performance value on the last day of the
112.      *         given test window.
113.      */
114.     public static double backtest(BaseValue baseValue,
115.         LocalDateTime startOfTestWindow, LocalDateTime endOfTestWindow,
116.         ValueDateTupel[] combinedForecasts, double baseScale,
117.         double capital) {
118.
119.       ValueDateTupel[] performanceValues = calculatePerformanceValues(
120.           baseValue, startOfTestWindow, endOfTestWindow, combinedForecasts,
121.           baseScale, capital);
122.
123.       return performanceValues[performanceValues.length - 1].getValue();
124.     }
125.
126.     /**
127.      * Calls
128.      * {@link #calculatePerformanceValues(BaseValue, LocalDateTime,
129.      * LocalDateTime, ValueDateTupel[], double, double)}
130.      * with instance properties.
131.      *
132.      * @param startOfTestWindow {@link LocalDateTime} First time interval of
133.      *                          test window.
134.      * @param endOfTestWindow   {@link LocalDateTime} Last time interval of
135.      *                          test window.
136.      * @return {@code double} by way of
137.      *         {@link #backtest(BaseValue, LocalDateTime, LocalDateTime,
138.      *         ValueDateTupel[], double, double)}.
139.      */
140.     public double backtest(LocalDateTime startOfTestWindow,
141.         LocalDateTime endOfTestWindow) {
142.       return SubSystem.backtest(this.getBaseValue(), startOfTestWindow,
143.           endOfTestWindow, this.getCombinedForecasts(), this.getBaseScale(),
144.           this.getCapital());
145.     }
146.
147.     /**
148.      * Calculates the performance values for the given time frame, based on
149.      * the given baseValue, forecasts, baseScale and capital.
150.      *
151.      * @param baseValue         {@link BaseValue} The base value upon which
152.      *                          the products' prices are to be based.
153.      * @param startOfTestWindow {@link LocalDateTime} First time interval for
154.      *                          testing.
155.      * @param endOfTestWindow   {@link LocalDateTime} Last time interval for
156.      *                          testing.
157.      * @param combinedForecasts {@code ValueDateTupel[]} Array of
```

```
158.      *                           {@link ValueDateTupel} containing the
159.      *                           forecasts for this performance calculation.
160.      * @param baseScale          {@code double} The scale the given forecasts
161.      *                           are based upon.
162.      * @param capital            {@code double} The starting capital.
163.      * @return {@code ValueDateTupel[]} An array of {@link ValueDateTupel}
164.      *         containing the value of all held assets + cash for each time
165.      *         interval between the given startOfTestWindow and
166.      *         endOfTestWindow.
167.      */
168.     public static ValueDateTupel[] calculatePerformanceValues(
169.         BaseValue baseValue, LocalDateTime startOfTestWindow,
170.         LocalDateTime endOfTestWindow, ValueDateTupel[] combinedForecasts,
171.         double baseScale, double capital) {
172.
173.       try {
174.         Validator.validateTimeWindow(startOfTestWindow, endOfTestWindow,
175.             baseValue.getValues());
176.       } catch (IllegalArgumentException e) {
177.         /*
178.          * If the message contains "values" the message references an error
179.          * in the given base values in combination with the given test
180.          * window.
181.          */
182.         if (e.getMessage().contains("values"))
183.           throw new IllegalArgumentException(
184.               "Given base value and test window do not fit.", e);
185.
186.         throw new IllegalArgumentException(MESSAGE_ILLEGAL_TEST_WINDOW, e);
187.       }
188.
189.       try {
190.         Validator.validateTimeWindow(startOfTestWindow, endOfTestWindow,
191.             combinedForecasts);
192.       } catch (IllegalArgumentException e) {
193.         /*
194.          * The general checks of the test window would have thrown Exceptions
195.          * in the previous try-catch, so here we only have to deal with
196.          * combinedForecasts specific Exceptions.
197.          */
198.         throw new IllegalArgumentException(
199.             "Given forecasts and test window do not fit.", e);
200.       }
201.
202.       /* Fetch all base values inside the test window */
203.       ValueDateTupel[] relevantBaseValues = ValueDateTupel.getElements(
204.           baseValue.getValues(), startOfTestWindow, endOfTestWindow);
205.
206.       /*
207.        * Get the product price factor to calculate long and short product
208.        * prices
209.        */
210.       double productPriceFactor = calculateProductPriceFactor(
211.           ValueDateTupel.getValues(relevantBaseValues));
```

```
212.
213.       /*
214.        * Calculate the product prices based on the base value for each
215.        * interval inside the testing timespan and the calculated
216.        * productPriceFactor
217.        */
218.       ValueDateTupel[] productPrices = calculateProductPrices(
219.           relevantBaseValues, productPriceFactor);
220.
221.       /*
222.        * Calculate the short product prices based on the base value for each
223.        * interval inside the testing timespan and the calculated
224.        * productPriceFactor
225.        */
226.       ValueDateTupel[] relevantShortIndexValues = ValueDateTupel.getElements(
227.           baseValue.getShortIndexValues(), startOfTestWindow,
228.           endOfTestWindow);
229.       ValueDateTupel[] shortProductPrices = calculateProductPrices(
230.           relevantShortIndexValues, productPriceFactor);
231.
232.       /* Fetch all forecasts relevant for this backtest. */
233.       ValueDateTupel[] relevantCombinedForecasts = ValueDateTupel
234.           .getElements(combinedForecasts, startOfTestWindow,
235.               endOfTestWindow);
236.
237.       ValueDateTupel[] performanceValues = {};
238.
239.       long longProductsCount = 0;
240.       long shortProductsCount = 0;
241.       for (int i = 0; i < relevantCombinedForecasts.length; i++) {
242.
243.         /*
244.          * Calculate the capital available for this time interval by
245.          * "selling" off all currently held positions at the this time
246.          * interval's prices.
247.          */
248.         capital += longProductsCount * productPrices[i].getValue();
249.         capital += shortProductsCount * shortProductPrices[i].getValue();
250.
251.         /* Reset the products count as they were sold off */
252.         shortProductsCount = 0;
253.         longProductsCount = 0;
254.
255.         /*
256.          * Add this capital as performance value, as the overall value of
257.          * cash + assets held will not change during buying.
258.          */
259.         ValueDateTupel performanceValue = new ValueDateTupel(
260.             relevantCombinedForecasts[i].getDate(), capital);
261.         performanceValues = ArrayUtils.add(performanceValues,
262.             performanceValue);
263.
264.         if (relevantCombinedForecasts[i].getValue() > 0) {
265.           /* Long position */
```

```
266.            longProductsCount = calculateProductsCount(capital,
267.                productPrices[i].getValue(),
268.                relevantCombinedForecasts[i].getValue(), baseScale);
269.
270.            /*
271.             * "Buy" the calculated count of products and thus reduce the cash
272.             * capital
273.             */
274.            capital -= longProductsCount * productPrices[i].getValue();
275.
276.          } else if (relevantCombinedForecasts[i].getValue() < 0) {
277.            /* short position */
278.            shortProductsCount = calculateProductsCount(capital,
279.                shortProductPrices[i].getValue(),
280.                relevantCombinedForecasts[i].getValue(), baseScale);
281.
282.            /*
283.             * "Buy" the calculated count of products and thus reduce the cash
284.             * capital
285.             */
286.            capital -= shortProductsCount * shortProductPrices[i].getValue();
287.          } else {
288.            /*
289.             * If forecast was 0 nothing would be bought so no default-else
290.             * branch is needed.
291.             */
292.          }
293.
294.        }
295.        return performanceValues;
296.      }
297.
298.      /**
299.       * Calls
300.       * {@link #calculatePerformanceValues(BaseValue, LocalDateTime,
301.       * LocalDateTime, ValueDateTupel[], double, double)}
302.       * with instance properties.
303.       *
304.       * @param startOfTestWindow {@link LocalDateTime} First time interval for
305.       *                          testing.
306.       * @param endOfTestWindow   {@link LocalDateTime} Last time interval for
307.       *                          testing.
308.       * @return {@code ValueDateTupel[]} by way of
309.       *         {@link #calculatePerformanceValues(BaseValue, LocalDateTime,
310.       *         LocalDateTime, ValueDateTupel[], double, double)}.
311.       */
312.      public ValueDateTupel[] calculatePerformanceValues(
313.          LocalDateTime startOfTestWindow, LocalDateTime endOfTestWindow) {
314.        return SubSystem.calculatePerformanceValues(this.getBaseValue(),
315.            startOfTestWindow, endOfTestWindow, this.getCombinedForecasts(),
316.            this.getBaseScale(), this.getCapital());
317.      }
318.
319.      /**
```

```
320.      * Calculates the combined forecasts for all rules of this Sub System.
321.      *
322.      * @return {@code ValueDateTupel[]} The combined forecasts for all rules,
323.      *         multiplied by {@link DiversificationMultiplier#getValue()} of
324.      *         this Sub System.
325.      */
326.     private ValueDateTupel[] calculateCombinedForecasts() {
327.       Rule[] instanceRules = this.getRules();
328.       /*
329.        * Calculate the weight by which all rules' forecasts shall be
330.        * multiplied by
331.        */
332.       double rulesWeight = 1d / instanceRules.length;
333.
334.       ValueDateTupel[] calculatedCombinedForecasts = {};
335.
336.       /* Step through the given rules */
337.       for (int rulesIndex = 0;
338.           rulesIndex < instanceRules.length;
339.           rulesIndex++) {
340.
341.         /* For each rule: Step through the forecasts */
342.         ValueDateTupel[] forecasts = instanceRules[rulesIndex]
343.             .getForecasts();
344.         for (int fcIndex = 0; fcIndex < forecasts.length; fcIndex++) {
345.
346.           if (rulesIndex == 0) {
347.             /*
348.              * Combined forecasts must be filled with values on first
349.              * go-through
350.              */
351.             ValueDateTupel vdtToAdd = new ValueDateTupel(
352.                 forecasts[fcIndex].getDate(),
353.                 forecasts[fcIndex].getValue() * rulesWeight);
354.             calculatedCombinedForecasts = ArrayUtils
355.                 .add(calculatedCombinedForecasts, vdtToAdd);
356.           } else {
357.             /*
358.              * If this is not the first go-through add the weighted forecasts
359.              * of the current rule
360.              */
361.             ValueDateTupel vdtToAdd = new ValueDateTupel(
362.                 calculatedCombinedForecasts[fcIndex].getDate(),
363.                 calculatedCombinedForecasts[fcIndex].getValue()
364.                     + forecasts[fcIndex].getValue() * rulesWeight);
365.             calculatedCombinedForecasts[fcIndex] = vdtToAdd;
366.           }
367.         }
368.       }
369.
370.       /*
371.        * Apply Diversification Multiplier to all forecast values. Cut off
372.        * Forecast values at 2 x base scale or -2 x base scale respectively
373.        */
```

```
374.      final double instanceBaseScale = this.getBaseScale();
375.      final double diversificationMultiplierValue = this
376.          .getDiversificationMultiplier().getValue();
377.      final double MAX_VALUE = instanceBaseScale * 2;
378.      final double MIN_VALUE = 0 - MAX_VALUE;
379.
380.      for (int fcIndex = 0;
381.          fcIndex < calculatedCombinedForecasts.length;
382.          fcIndex++) {
383.        double fcWithDM = calculatedCombinedForecasts[fcIndex].getValue()
384.            * diversificationMultiplierValue;
385.        if (fcWithDM > MAX_VALUE)
386.          fcWithDM = MAX_VALUE;
387.        if (fcWithDM < MIN_VALUE)
388.          fcWithDM = MIN_VALUE;
389.
390.        calculatedCombinedForecasts[fcIndex].setValue(fcWithDM);
391.      }
392.
393.      return calculatedCombinedForecasts;
394.    }
395.
396.    /**
397.     * Call {@link SubSystem#calculateProductPriceFactor(double[], double)}
398.     * passing PRICE_FACTOR_BASE_SCALE as param.
399.     *
400.     * @see SubSystem#calculateProductPriceFactor(double[], double)
401.     * @param values {@code double[]} An Array of values the factor is to be
402.     *               calculated for
403.     * @return {@code double} The calculated factor.
404.     */
405.    private static double calculateProductPriceFactor(double[] values) {
406.      return SubSystem.calculateProductPriceFactor(values,
407.          PRICE_FACTOR_BASE_SCALE);
408.    }
409.
410.    /**
411.     * Calculate the factor by which all of the given values must be
412.     * multiplied so their products have an average of priceFactorBaseScale.
413.     * <br>
414.     * The factor is calculated by inverting the average of the given values
415.     * divided by the given priceFactorBaseScale.
416.     *
417.     * @param values               {@code double[]} An Array of values the
418.     *                             factor is to be calculated for
419.     * @param priceFactorBaseScale {@code double} The base scale to use.
420.     * @return {@code double} The calculated factor.
421.     */
422.    private static double calculateProductPriceFactor(double[] values,
423.        double priceFactorBaseScale) {
424.      double averageCourseValue = Util.calculateAverage(values);
425.
426.      return 1 / (averageCourseValue / priceFactorBaseScale);
427.    }
```

```
428.
429.     /**
430.      * Calculate product prices based on the given array of values and a
431.      * given product price factor.
432.      *
433.      * @param baseValues         {@code ValueDateTupel[]} The values the
434.      *                           prices are to be based on.
435.      * @param productPriceFactor {@code double} The factor used to calculate
436.      *                           the product prices.
437.      * @return {@code ValueDateTupel[]} An array of prices using the dates of
438.      *         the given baseValues.
439.      */
440.     private static ValueDateTupel[] calculateProductPrices(
441.         ValueDateTupel[] baseValues, double productPriceFactor) {
442.       ValueDateTupel[] productPrices = {};
443.       for (ValueDateTupel baseValue : baseValues)
444.         productPrices = ValueDateTupel.addOneAt(productPrices,
445.             new ValueDateTupel(baseValue.getDate(),
446.                 baseValue.getValue() * productPriceFactor),
447.             productPrices.length);
448.
449.       return productPrices;
450.     }
451.
452.     /**
453.      * Calculates the products to buy during a trading period according to
454.      * the given price and given forecast.
455.      *
456.      * @param capital   {@code double} The capital available for trading.
457.      * @param price     {@code double} The price at which a product can be
458.      *                  bought.
459.      * @param forecast  {@code double} The forecast for the current trading
460.      *                  period.
461.      * @param baseScale {@code double} The base scale by which the given
462.      *                  forecast is scaled.
463.      * @return {@code int} The number of products to buy.
464.      */
465.     private static long calculateProductsCount(double capital, double price,
466.         double forecast, double baseScale) {
467.       /* Number of products if forecast had MAX_VALUE */
468.       double maxProductsCount = capital / price;
469.
470.       /* Number of products if forecast was 1 */
471.       double fcOneProductsCounts = maxProductsCount / (baseScale * 2);
472.
473.       /*
474.        * Invert current forecast if it's negative to always generate a
475.        * positive number of products
476.        */
477.       if (forecast < 0)
478.         forecast *= -1;
479.
480.       /*
481.        * Number of products for actual forecast. Accept rounding
```

```
482.        * inaccuracies.
483.        */
484.       return (long) (fcOneProductsCounts * forecast);
485.     }
486.
487.     /**
488.      * Validate the given input parameters.
489.      *
490.      * @param baseValue {@link BaseValue} The baseValue to validate. Must
491.      *                  pass {@link Validator#validateBaseValue(BaseValue)}.
492.      * @param rules     {@code Rule[]} The rules to validate.
493.      *                  <ul>
494.      *                  <li>Must pass
495.      *                  {@link Validator#validateRules(Rule[])}</li>
496.      *                  <li>Each rule's base value must be same as the given
497.      *                  base value.</li>
498.      *                  </ul>
499.      * @param capital   {@code double} The capital to validate. Must pass
500.      *                  {@link Validator#validatePositiveDouble(double)}.
501.      * @param baseScale {@code double} The base scale to validate. Must pass
502.      *                  {@link Validator#validatePositiveDouble(double)}.
503.      * @throws IllegalArgumentException if any of the above criteria is not
504.      *                                  met.
505.      */
506.     private static void validateInput(BaseValue baseValue, Rule[] rules,
507.         double capital, double baseScale) {
508.       Validator.validateBaseValue(baseValue);
509.
510.       Validator.validateRules(rules);
511.
512.       Validator.validateRulesVsBaseValue(rules, baseValue);
513.
514.       try {
515.         Validator.validatePositiveDouble(capital);
516.       } catch (IllegalArgumentException e) {
517.         throw new IllegalArgumentException(
518.             "Given capital does not meet specifications.", e);
519.       }
520.
521.       try {
522.         Validator.validatePositiveDouble(baseScale);
523.       } catch (IllegalArgumentException e) {
524.         throw new IllegalArgumentException(
525.             "Given base scale does not meet specifications.", e);
526.       }
527.
528.       Validator.validateRulesVsBaseScale(rules, baseScale);
529.     }
530.
531.     /**
532.      * Validate the given rules. <br>
533.      * Rules have to fulfill the following criteria:
534.      * <ul>
535.      * <li>Be unique by {@link Util#areRulesUnique(Rule[])}</li>
```

```
536.      * <li>Equal in startOfReferenceWindow and endOfReferenceWinow by
537.      * {@link LocalDateTime#isEqual(ChronoLocalDateTime)}</li>
538.      * </ul>
539.      *
540.      * @param rules {@code Rule[]} Rules that are to be checked.
541.      * @throws IllegalArgumentException if the given rules are not unique.
542.      */
543.     private static void validateRules(Rule[] rules) {
544.       if (!Util.areRulesUnique(rules))
545.         throw new IllegalArgumentException("The given rules are not unique."
546.             + " Only unique rules can be used.");
547.
548.       /* All rules need to have the same reference window */
549.       for (int i = 1; i < rules.length; i++) {
550.         if (!rules[i].getStartOfReferenceWindow()
551.             .isEqual(rules[i - 1].getStartOfReferenceWindow())
552.             || !rules[i].getEndOfReferenceWindow()
553.                 .isEqual(rules[i - 1].getEndOfReferenceWindow())) {
554.           throw new IllegalArgumentException(
555.               "All rules need to have the same reference window but rules at"
556.                   + " position " + (i - 1) + " and " + i + " differ.");
557.         }
558.       }
559.
560.     }
561.
562.     /**
563.      * ========================================================================
564.      * OVERRIDES
565.      * ========================================================================
566.      */
567.
568.     /**
569.      * A hash code for this SubSystem.
570.      */
571.     @GeneratedCode
572.     @Override
573.     public int hashCode() {
574.       final int prime = 31;
575.       int result = 1;
576.       long temp;
577.       temp = Double.doubleToLongBits(baseScale);
578.       result = prime * result + (int) (temp ^ (temp >>> 32));
579.       result = prime * result
580.           + ((baseValue == null) ? 0 : baseValue.hashCode());
581.       temp = Double.doubleToLongBits(capital);
582.       result = prime * result + (int) (temp ^ (temp >>> 32));
583.       result = prime * result + Arrays.hashCode(combinedForecasts);
584.       result = prime * result + ((diversificationMultiplier == null) ? 0
585.           : diversificationMultiplier.hashCode());
586.       result = prime * result + Arrays.hashCode(rules);
587.       return result;
588.     }
589.
```

```
    /**
     * Checks if this SubSystem is equal to another SubSystem.
     */
    @GeneratedCode
    @Override
    public boolean equals(Object obj) {
      if (this == obj)
        return true;
      if (obj == null)
        return false;
      if (getClass() != obj.getClass())
        return false;
      SubSystem other = (SubSystem) obj;
      if (Double.doubleToLongBits(baseScale) != Double
          .doubleToLongBits(other.baseScale))
        return false;
      if (baseValue == null) {
        if (other.baseValue != null)
          return false;
      } else if (!baseValue.equals(other.baseValue))
        return false;
      if (Double.doubleToLongBits(capital) != Double
          .doubleToLongBits(other.capital))
        return false;
      if (!Arrays.equals(combinedForecasts, other.combinedForecasts))
        return false;
      if (diversificationMultiplier == null) {
        if (other.diversificationMultiplier != null)
          return false;
      } else if (!diversificationMultiplier
          .equals(other.diversificationMultiplier))
        return false;
      if (!Arrays.equals(rules, other.rules))
        return false;
      return true;
    }

    /**
     * Outputs the fields of this SubSystem as a {@code String}.
     */
    @GeneratedCode
    @Override
    public String toString() {
      StringBuilder builder = new StringBuilder();
      builder.append("SubSystem [baseValue=");
      builder.append(baseValue);
      builder.append(", rules=");
      builder.append(Arrays.toString(rules));
      builder.append(", diversificationMultiplier=");
      builder.append(diversificationMultiplier);
      builder.append(", capital=");
      builder.append(capital);
      builder.append(", combinedForecasts=");
      builder.append(Arrays.toString(combinedForecasts));
```

```
644.       builder.append(", baseScale=");
645.       builder.append(baseScale);
646.       builder.append("]");
647.       return builder.toString();
648.     }
649.
650.     /**
651.      * ========================================================================
652.      * GETTERS AND SETTERS
653.      * ========================================================================
654.      */
655.
656.     /**
657.      * Get the {@link BaseValue} for this subsystem.
658.      *
659.      * @return baseValue SubSystem
660.      */
661.     public BaseValue getBaseValue() {
662.       return baseValue;
663.     }
664.
665.     /**
666.      * Set the {@link BaseValue} for this subsystem.
667.      *
668.      * @param baseValue the baseValue to set
669.      */
670.     private void setBaseValue(BaseValue baseValue) {
671.       this.baseValue = baseValue;
672.     }
673.
674.     /**
675.      * Get the rules for this subsystem.
676.      *
677.      * @return rules SubSystem
678.      */
679.     public Rule[] getRules() {
680.       return rules;
681.     }
682.
683.     /**
684.      * Set the rules for this subsystem.
685.      *
686.      * @param rules the rules to set
687.      */
688.     public void setRules(Rule[] rules) {
689.       this.rules = rules;
690.     }
691.
692.     /**
693.      * Get the {@link DiversificationMultiplier} for this subsystem.
694.      *
695.      * @return diversificationMultiplier SubSystem
696.      */
697.     public DiversificationMultiplier getDiversificationMultiplier() {
```

```
698.       return diversificationMultiplier;
699.     }
700.
701.     /**
702.      * Set the {@link DiversificationMultiplier} for this subsystem.
703.      *
704.      * @param diversificationMultiplier the diversificationMultiplier to set
705.      */
706.     private void setDiversificationMultiplier(
707.         DiversificationMultiplier diversificationMultiplier) {
708.       this.diversificationMultiplier = diversificationMultiplier;
709.     }
710.
711.     /**
712.      * Get the capital for this subsystem.
713.      *
714.      * @return capital SubSystem
715.      */
716.     public double getCapital() {
717.       return capital;
718.     }
719.
720.     /**
721.      * Set the capital for this subsystem.
722.      *
723.      * @param capital the capital to set
724.      */
725.     private void setCapital(double capital) {
726.       this.capital = capital;
727.     }
728.
729.     /**
730.      * Get the combined forecasts for this subsystem.
731.      *
732.      * @return combinedForecasts SubSystem
733.      */
734.     public ValueDateTupel[] getCombinedForecasts() {
735.       return combinedForecasts;
736.     }
737.
738.     /**
739.      * Set the combined forecasts for this subsystem.
740.      *
741.      * @param combinedForecasts the combinedForecasts to set
742.      */
743.     public void setCombinedForecasts(ValueDateTupel[] combinedForecasts) {
744.       this.combinedForecasts = combinedForecasts;
745.     }
746.
747.     /**
748.      * Get the base scale for this subsystem.
749.      *
750.      * @return baseScale SubSystem
751.      */
```

```
752.     public double getBaseScale() {
753.       return baseScale;
754.     }
755.
756.     /**
757.      * Set the base scale for this subsystem.
758.      *
759.      * @param baseScale the baseScale to set
760.      */
761.     public void setBaseScale(double baseScale) {
762.       this.baseScale = baseScale;
763.     }
764.   }
```

Komponente DiversificationMultiplier

Listing 19: Komponente DiversificationMultiplier

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.util.Arrays;
4.
5.  import org.apache.commons.lang3.ArrayUtils;
6.  import org.apache.commons.math3.linear.BlockRealMatrix;
7.  import org.apache.commons.math3.stat.correlation.PearsonsCorrelation;
8.
9.  import de.rumford.tradingsystem.helper.GeneratedCode;
10. import de.rumford.tradingsystem.helper.Validator;
11.
12. /**
13.  * The DiversificationMultiplier negates the fact that using multiple rules
14.  * upon a single base value flattens the forecast curve.
15.  * <p>
16.  * "The only free real estate in capital investment is diversification". By
17.  * diversifying between assets the risk of an investment shall be reduced.
18.  * <p>
19.  * By diversifying forecast calculation (e.g. taking more rules into
20.  * account when predicting a base value's future development) a similar
21.  * effect takes place: Their respective predictions tend to reduce the risk
22.  * taken when combined, so their combined forecasts are less volatile than
23.  * each rule's forecasts is (Robert Carver, Systematic Trading (2015), pp.
24.  * 129 f.).
25.  * <p>
26.  * Therefore a factor is needed to ensure the desired volatility target is
27.  * met and to also ensure that the combined forecasts move around the
28.  * desired base scale (as defined in the {@link SubSystem} owning this
29.  * DiversificationMultiplier).
30.  *
31.  * @author Max Rumford
32.  *
33.  */
34. public class DiversificationMultiplier {
35.
36.   /* The value of this diversification multiplier. */
37.   private double value = 0d;
38.   /* The weights of the given rules. */
39.   private double[] weights;
40.   /* The relevant forecasts of the given rules. */
41.   private double[][] relevantForecasts;
42.   /* The correaltions of the given rules. */
43.   private double[][] correlations;
44.
45.   /**
46.    * Constructor for the class DiversificationMultiplier (DM). A DM is
47.    * always only valid for a given set of rules (which should be, but don't
48.    * have to be, unique). There rules must be known upon instantiation of a
49.    * new DM. Based on the given rules the DM value is calculated, using the
```

```
50.       * relative weights of all given rules (relative based on their placing
51.       * inside the rules tree).
52.       * <p>
53.       * There can be as many rules in the given array as desired, though the
54.       * desired ratio between rules (and their respective weights, which
55.       * shrink with each new layer) and their significance to the whole system
56.       * should be accounted for and is at the user's discretion.
57.       *
58.       * @param rules {@code Rule[]} An array of {@link Rule}s to be accounted
59.       *              for in this DM. Must pass
60.       *              {@link Validator#validateRules(Rule[])}.
61.       */
62.      public DiversificationMultiplier(Rule[] rules) {
63.        validateInput(rules);
64.
65.        WeightsAndForecasts weightsAndForecasts =
66.            getWeightsAndForecastsFromRules(rules);
67.        this.setWeights(weightsAndForecasts.weights);
68.        this.setRelevantForecasts(weightsAndForecasts.forecasts);
69.
70.        this.setCorrelations(
71.            getCorrelationsFromForecasts(this.getRelevantForecasts()));
72.
73.        this.setValue(this.calculateDiversificiationMultiplierValue());
74.      }
75.
76.      /**
77.       * Private class for extraction of weights and forecasts from the given
78.       * rules.
79.       */
80.      private class WeightsAndForecasts {
81.        public double[] weights;
82.        public double[][] forecasts;
83.
84.        public WeightsAndForecasts(double[] weights,
85.            double[][] relevantForecasts) {
86.          this.weights = weights;
87.          this.forecasts = relevantForecasts;
88.        }
89.
90.      }
91.
92.      /**
93.       * Calculate the diversification multiplier for the weights and
94.       * correlations set with this class. Represents this formula with c =
95.       * matrix of correlations, w = list of weights, i,j = indices:
96.       *
97.       * 1 / sqrt[ SUM( c_i,j * w_i * w_j ) ]
98.       *
99.       * @return {@code double} diversification multiplier for set weights and
100.      *         correlations
101.      * @throws IllegalArgumentException if correlations or weights do not
102.      *                                  meet criteria: non-empty, same number
103.      *                                  of values, correlations: same amount
```

```
     *                                    of rows and columns
     */
    private double calculateDiversificiationMultiplierValue() {
      double[][] instanceCorrelations = this.getCorrelations();
      double[] instanceWeights = this.getWeights();

      /* local field to hold sum of correlations multiplied with weights */
      double sumOfCorrelationsWeights = 0f;

      /*
       * Get the sum of all correlations multiplier with both corresponding
       * weights...
       */
      for (int row = 0; row < instanceCorrelations.length; row++) {
        for (int col = 0; col < instanceCorrelations.length; col++) {
          /*
           * ... by multiplying the correlation with both corresponding
           * weights
           */
          sumOfCorrelationsWeights += instanceCorrelations[row][col]
              * instanceWeights[row] * instanceWeights[col];
        }
      }

      /*
       * sumOfCorrelationsWeights is always > 0 as there is always at least
       * one weight > 0 and at least on correlation > 0 (self correlation)
       */
      return 1 / Math.sqrt(sumOfCorrelationsWeights);
    }

    /**
     * Calculate the correlations from the given array of forecast arrays.
     *
     * @param forecasts {@code double[][]} The forecasts to be used for
     *                  calculation.
     * @return {@code double[][]} A matrix of correlations, as by
     *         {@link PearsonsCorrelation#getCorrelationMatrix()}.
     */
    private static double[][] getCorrelationsFromForecasts(
        double[][] forecasts) {
      /*
       * If there is only one row of data return a 1x1 self correlation
       * matrix
       */
      if (forecasts.length == 1) {
        return new double[][] { { 1 } };
      }

      /* Load the given values into rows of a matrix */
      BlockRealMatrix matrix = new BlockRealMatrix(forecasts);
      /* Transpose the values into columns to get the correct correlations */
      matrix = matrix.transpose();

```

```
158.       /* Get the correlations of the passed value arrays */
159.       PearsonsCorrelation pearsonsCorrelations = new PearsonsCorrelation(
160.           matrix);
161.       return pearsonsCorrelations.getCorrelationMatrix().getData();
162.     }
163.
164.     /**
165.      * Recursively get the weights and forecasts from the given array of
166.      * Rules.
167.      *
168.      * @param rules {@code Rule[]} The array of rules to be searched.
169.      * @return {@link WeightsAndForecasts} The extracted weights and
170.      *         forecasts from the given array of Rules.
171.      */
172.     private WeightsAndForecasts getWeightsAndForecastsFromRules(
173.         Rule[] rules) {
174.       double[] weightsFromRules = {};
175.       double[][] relevantForecastsFromRules = {};
176.
177.       /* Iterate over the given rules */
178.       for (Rule rule : rules) {
179.
180.         /* If a rule has variations get their weights and forecasts */
181.         if (rule.hasVariations()) {
182.           WeightsAndForecasts wafToAdd = getWeightsAndForecastsFromRules(
183.               rule.getVariations());
184.           for (double weight : wafToAdd.weights)
185.             weightsFromRules = ArrayUtils.add(weightsFromRules,
186.                 weight * rule.getWeight());
187.
188.           for (double[] forecasts : wafToAdd.forecasts)
189.             relevantForecastsFromRules = ArrayUtils
190.                 .add(relevantForecastsFromRules, forecasts);
191.
192.         } else {
193.           double weight = rule.getWeight();
194.           /*
195.            * If a top level rule has no variations its weight has not been
196.            * set. Manually set its weight to be 1/numberOfTopeLevelRules
197.            */
198.           if (weight == 0)
199.             weight = 1d / rules.length;
200.           weightsFromRules = ArrayUtils.add(weightsFromRules, weight);
201.
202.           relevantForecastsFromRules = ArrayUtils.add(
203.               relevantForecastsFromRules,
204.               rule.extractRelevantForecastValues());
205.         }
206.       }
207.       return new WeightsAndForecasts(weightsFromRules,
208.           relevantForecastsFromRules);
209.     }
210.
211.     /**
```

```
     * Validates the given input values.
     *
     * @param rules {@code Rule[]} Must pass
     *              {@link Validator#validateRules(Rule[])}.
     */
    private static void validateInput(Rule[] rules) {
      Validator.validateRules(rules);
    }

    /**
     * ========================================================================
     * OVERRIDES
     * ========================================================================
     */

    /* A hash code for this diversification multiplier. */
    @GeneratedCode
    @Override
    public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + Arrays.deepHashCode(correlations);
      result = prime * result + Arrays.deepHashCode(relevantForecasts);
      long temp;
      temp = Double.doubleToLongBits(value);
      result = prime * result + (int) (temp ^ (temp >>> 32));
      result = prime * result + Arrays.hashCode(weights);
      return result;
    }

    /**
     * Checks if this diversification multiplier is equal to another
     * diversification multiplier.
     */
    @GeneratedCode
    @Override
    public boolean equals(Object obj) {
      if (this == obj)
        return true;
      if (obj == null)
        return false;
      if (getClass() != obj.getClass())
        return false;
      DiversificationMultiplier other = (DiversificationMultiplier) obj;
      if (!Arrays.deepEquals(correlations, other.correlations))
        return false;
      if (!Arrays.deepEquals(relevantForecasts, other.relevantForecasts))
        return false;
      if (Double.doubleToLongBits(value) != Double
          .doubleToLongBits(other.value))
        return false;
      if (!Arrays.equals(weights, other.weights))
        return false;
      return true;
```

```
266.     }
267.
268.     /**
269.      * Outputs the fields of this diversification multiplier as a
270.      * {@code String}.
271.      */
272.     @GeneratedCode
273.     @Override
274.     public String toString() {
275.       StringBuilder builder = new StringBuilder();
276.       builder.append("DiversificationMultiplier [value=");
277.       builder.append(value);
278.       builder.append(", weights=");
279.       builder.append(Arrays.toString(weights));
280.       builder.append(", relevantForecasts=");
281.       builder.append(Arrays.toString(relevantForecasts));
282.       builder.append(", correlations=");
283.       builder.append(Arrays.toString(correlations));
284.       builder.append("]");
285.       return builder.toString();
286.     }
287.
288.     /**
289.      * ========================================================================
290.      * GETTERS AND SETTERS
291.      * ========================================================================
292.      */
293.
294.     /**
295.      * Get value of this {@link DiversificationMultiplier}
296.      *
297.      * @return {@code double} value of this {@link DiversificationMultiplier}
298.      */
299.     public double getValue() {
300.       return value;
301.     }
302.
303.     /**
304.      * Set the value in this {@link DiversificationMultiplier}
305.      *
306.      * @param value the value to set
307.      */
308.     private void setValue(double value) {
309.       this.value = value;
310.     }
311.
312.     /**
313.      * Get the weights considered in this {@link DiversificationMultiplier}
314.      *
315.      * @return {@code double[]} Weights in this
316.      *         {@link DiversificationMultiplier}
317.      */
318.     public double[] getWeights() {
319.       return weights;
```

```
320.     }
321.
322.     /**
323.      * Set the Weights in this {@link DiversificationMultiplier}
324.      *
325.      * @param {@code double[]} the weights to set
326.      */
327.     private void setWeights(double[] weights) {
328.       this.weights = weights;
329.     }
330.
331.     /**
332.      * Get the relevant forecasts in this {@link DiversificationMultiplier}
333.      *
334.      * @return relevantForecasts DiversificationMultiplier
335.      */
336.     public double[][] getRelevantForecasts() {
337.       return relevantForecasts;
338.     }
339.
340.     /**
341.      * Set the relevant forecasts in this {@link DiversificationMultiplier}
342.      *
343.      * @param relevantForecasts the relevantForecasts to set
344.      */
345.     private void setRelevantForecasts(double[][] relevantForecasts) {
346.       this.relevantForecasts = relevantForecasts;
347.     }
348.
349.     /**
350.      * Get the correlations in this {@link DiversificationMultiplier}
351.      *
352.      * @return {@code double[][]} The correlations in this
353.      *         {@link DiversificationMultiplier}
354.      */
355.     public double[][] getCorrelations() {
356.       return correlations;
357.     }
358.
359.     /**
360.      * Set the correlations to be used in this
361.      * {@link DiversificationMultiplier}
362.      *
363.      * @param {@code double[][]} the correlations to be used in this
364.      * {@link DiversificationMultiplier}
365.      */
366.     private void setCorrelations(double[][] correlations) {
367.       this.correlations = correlations;
368.     }
369.   }
```

Komponente ValueDateTupel

Listing 20: Komponente ValueDateTupel

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  import java.time.LocalDateTime;
4.  import java.time.chrono.ChronoLocalDateTime;
5.  import java.util.ArrayList;
6.  import java.util.Arrays;
7.  import java.util.List;
8.  import java.util.TreeSet;
9.
10. import org.apache.commons.lang3.ArrayUtils;
11.
12. import de.rumford.tradingsystem.BaseValue;
13.
14. /**
15.  * A ValueDateTupel represents a decimal value at a given point in time.
16.  * <p>
17.  * The ValueDateTupel is the most used helper class in this library. It
18.  * consists of a LocalDateTime value representing a point in time, and of a
19.  * double, representing any kind of decimal value associated with the
20.  * aforementioned point in time. By using LocalDateTime (an not just a
21.  * class representing Date values) intraday usage is possible.
22.  * <p>
23.  * The ValueDateTupel brings a lot of static method used throughout the
24.  * entire library. Most of these methods deal with arrays or larger
25.  * structures of ValueDateTupel, as the instance of ValueDateTupel by
26.  * itself does not have many limitations.
27.  *
28.  * @author Max Rumford
29.  *
30.  */
31. public class ValueDateTupel {
32.
33.   /* The value to be represented. */
34.   private double value;
35.   /* The datetime to be represented. */
36.   private LocalDateTime date;
37.
38.   static final String MESSAGE_ARRAY_MUST_NOT_BE_NULL =
39.       "Given array must not be null";
40.   static final String MESSAGE_VALUE_MUST_NOT_BE_NULL =
41.       "Given value must not be null";
42.
43.   /**
44.    * Creates a new {@link ValueDateTupel} instance using the given
45.    * LocaDateTime and double.
46.    *
47.    * @param date  {@link LocalDateTime} The dateTime to be set for this
48.    *              {@link ValueDateTupel}
49.    * @param value {@code double} The value to be set for this
```

```
50.       *              {@link ValueDateTupel}
51.       */
52.      public ValueDateTupel(LocalDateTime date, double value) {
53.        this.setDate(date);
54.        this.setValue(value);
55.      }
56.
57.      /**
58.       * Adds a given {@link ValueDateTupel} to the given array of
59.       * {@link ValueDateTupel} at the given position. Returns the given array
60.       * extended with the given single value.
61.       *
62.       * @param valueDateTupels {@code ValueDateTupel[]} Array to be extended.
63.       * @param vdtToBeAdded    {@link ValueDateTupel} Value to be added.
64.       * @param position        {@code int} Position the given value shall be
65.       *                        put into.
66.       * @return {@code ValueDateTupel[]} The extended array.
67.       * @throws IllegalArgumentException If the given array is null.
68.       * @throws IllegalArgumentException If the given value to be added is
69.       *                                  null.
70.       * @throws IllegalArgumentException If the position is negative.
71.       * @throws IllegalArgumentException If the given position is greater than
72.       *                                  the length of the given array.
73.       */
74.      public static ValueDateTupel[] addOneAt(ValueDateTupel[] valueDateTupels,
75.          ValueDateTupel vdtToBeAdded, int position) {
76.        if (valueDateTupels == null)
77.          throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
78.        if (vdtToBeAdded == null)
79.          throw new IllegalArgumentException(MESSAGE_VALUE_MUST_NOT_BE_NULL);
80.        if (position < 0)
81.          throw new IllegalArgumentException(
82.              "Cannot add a value at position < 0. Given position is "
83.                  + position);
84.        if (position > valueDateTupels.length)
85.          throw new IllegalArgumentException(
86.              "Cannot add a value at position > " + valueDateTupels.length
87.                  + ". Given position is " + position + ".");
88.
89.        ValueDateTupel[] extendedArray = ValueDateTupel
90.            .createEmptyArray(valueDateTupels.length + 1);
91.
92.        /* Add new ValueDateTupel at the beginning of the given array. */
93.        if (position == 0) {
94.          extendedArray[position] = vdtToBeAdded;
95.          System.arraycopy(valueDateTupels, 0, extendedArray, 1,
96.              valueDateTupels.length);
97.          return extendedArray;
98.        }
99.        /* Add new ValueDateTupel at the end of the given array. */
100.       if (position == valueDateTupels.length) {
101.         System.arraycopy(valueDateTupels, 0, extendedArray, 0,
102.             valueDateTupels.length);
103.         extendedArray[position] = vdtToBeAdded;
```

```
104.          return extendedArray;
105.        }
106.        /*
107.         * This code is only reached, when the new ValueDateTupel shall not be
108.         * added at end or at beginning.
109.         */
110.        /* Add all values prior to the new ValueDateTupel */
111.        System.arraycopy(valueDateTupels, 0, extendedArray, 0, position);
112.        /* Add new ValueDateTupel at the given position. */
113.        extendedArray[position] = vdtToBeAdded;
114.        /* Add all values subsequent to the new ValueDateTupel */
115.        System.arraycopy(valueDateTupels, position, extendedArray,
116.            position + 1, valueDateTupels.length - position);
117.        return extendedArray;
118.      }
119.
120.      /**
121.       * Add missing {@link LocalDateTime} values to all given
122.       * {@code ValueDateTupel[]}. The corresponding value will be set to
123.       * average the values of its direct predecessor and successor.
124.       * <p>
125.       * If there are multiple {@link LocalDateTime} missing in a row, all of
126.       * those will get the average value of the last position before and the
127.       * first position after all missing values.
128.       * <p>
129.       * If the missing {@link LocalDateTime} would be the first value in the
130.       * new array, its value will be set to match the previously first one.
131.       * <p>
132.       * If the missing {@link LocalDateTime} would be the last value in the
133.       * new array, its value will be set to match the previously last one.
134.       *
135.       * @param valueDateTupels {@code ValueDateTupel[][]} Array of arrays of
136.       *                        {@link ValueDateTupel} whose
137.       *                        {@link LocalDateTime} shall be aligned.
138.       * @return {@code ValueDateTupel[][]} Array of arrays of
139.       *         {@link ValueDateTupel} with now aligned {@link LocalDateTime}
140.       *         values.
141.       * @throws IllegalArgumentException If the given array of arrays is null.
142.       * @throws IllegalArgumentException If the given array of arrays contains
143.       *                                  null.
144.       * @throws IllegalArgumentException If any array of the given array of
145.       *                                  arrays contains null.
146.       * @throws IllegalArgumentException If the given array contains an array
147.       *                                  of {@link ValueDateTupel} not sorted
148.       *                                  in ascending order.
149.       * @throws IllegalArgumentException If the one of the given arrays
150.       *                                  contains only {@link Double#NaN}.
151.       */
152.      public static ValueDateTupel[][] alignDates(
153.          ValueDateTupel[][] valueDateTupels) {
154.        if (valueDateTupels == null)
155.          throw new IllegalArgumentException(
156.              "Given array of arrays must not be null");
157.
```

```
158.      /* TreeSet (unique and sorted) of all dates in all valueDateTupel[] */
159.      TreeSet<LocalDateTime> uniqueSortedDates = getUniqueDates(
160.          valueDateTupels);
161.
162.      /* Loop over all rows */
163.      for (int rowIndex = 0; rowIndex < valueDateTupels.length; rowIndex++) {
164.
165.        try {
166.          /* Validate if the row contains at least one suitable value. */
167.          Validator.validateRow(valueDateTupels[rowIndex]);
168.        } catch (IllegalArgumentException e) {
169.          throw new IllegalArgumentException(
170.              "Row at position " + rowIndex + " is not valid.", e);
171.        }
172.
173.        /*
174.         * If the row's length equals the length of uniqueSortedDates no
175.         * Value has to be added as it already contains all dateTimes.
176.         */
177.        if (valueDateTupels[rowIndex].length == uniqueSortedDates.size())
178.          continue;
179.
180.        /* Enhance current row by missing LocalDateTime values. */
181.        valueDateTupels[rowIndex] = enhanceRowByNaNs(
182.            valueDateTupels[rowIndex], uniqueSortedDates);
183.
184.        /* Replace all values of Double.NaN by real values. */
185.        valueDateTupels[rowIndex] = replaceNansByValues(
186.            valueDateTupels[rowIndex]);
187.
188.      }
189.      return valueDateTupels;
190.    }
191.
192.    /**
193.     * Check if the given {@link ValueDateTupel} can be found in the given
194.     * array of {@link ValueDateTupel}. Will only find exact matches.
195.     *
196.     * @param valueDateTupels {@code ValueDateTupel[]} Array to be searched
197.     *                        in.
198.     * @param vdtToBeFound    {@link ValueDateTupel} Value to be searched
199.     *                        for.
200.     * @return {@code boolean} True, if the given value can be found in the
201.     *         given array, false otherwise.
202.     */
203.    public static boolean contains(ValueDateTupel[] valueDateTupels,
204.        ValueDateTupel vdtToBeFound) {
205.      if (valueDateTupels == null)
206.        throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
207.      List<ValueDateTupel> list = new ArrayList<>(
208.          Arrays.asList(valueDateTupels));
209.      return list.contains(vdtToBeFound);
210.    }
211.
```

```
212.     /**
213.      * Check if the given array of {@link ValueDateTupel} contains the given
214.      * {@link LocalDateTime}.
215.      *
216.      * @param valueDateTupels {@code ValueDateTupel[]} Array to be searched
217.      *                        in.
218.      * @param dtToBeFound     {@link LocalDateTime} Value to be searched for.
219.      * @return {@code boolean} True, if the given value can be found inside
220.      *         the given array, false otherwise.
221.      * @throws IllegalArgumentException If the given array of
222.      *                                  {@link ValueDateTupel} is null.
223.      * @throws IllegalArgumentException If the given {@link LocalDateTime} is
224.      *                                  null.
225.      *
226.      */
227.     public static boolean containsDate(ValueDateTupel[] valueDateTupels,
228.         LocalDateTime dtToBeFound) {
229.       if (valueDateTupels == null)
230.         throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
231.       if (dtToBeFound == null)
232.         throw new IllegalArgumentException(MESSAGE_VALUE_MUST_NOT_BE_NULL);
233.       /* Load all values from the given array into an ArrayList. */
234.       List<LocalDateTime> list = new ArrayList<>(
235.           Arrays.asList(ValueDateTupel.getDates(valueDateTupels)));
236.       /* Utilize the generic contains method on ArrayList. */
237.       return list.contains(dtToBeFound);
238.     }
239.
240.     /**
241.      * Creates an empty array of {@link ValueDateTupel}.
242.      *
243.      * @return {@code ValueDateTupel[]} An Empty array of
244.      *         {@link ValueDateTupel}.
245.      */
246.     public static ValueDateTupel[] createEmptyArray() {
247.       return ValueDateTupel.createEmptyArray(0);
248.     }
249.
250.     /**
251.      * Creates an empty array of {@link ValueDateTupel} with the given
252.      * length.
253.      *
254.      * @param length {@code int} Length the new array should have.
255.      * @return {@code ValueDateTupel[]} An Empty array of
256.      *         {@link ValueDateTupel}.
257.      */
258.     public static ValueDateTupel[] createEmptyArray(int length) {
259.       return new ValueDateTupel[length];
260.     }
261.
262.     /**
263.      * Check if the {@link BaseValue} instance contains the given
264.      * {@link LocalDateTime} in its values. Returns the containing
265.      * {@link ValueDateTupel} if so, returns {@code null} otherwise.
```

```
266.      *
267.      * @param valueDateTupels {@code ValueDateTupel[]} The array of
268.      *                        {@link ValueDateTupel} to be searched through.
269.      * @param dtToBeFound     {@link LocalDateTime} Value to be found inside
270.      *                        the {@link BaseValue} values.
271.      * @return {@link ValueDateTupel} containing the given
272.      *         {@link LocalDateTime}. {@code null} if the given
273.      *         {@link LocalDateTime} cannot be found.
274.      */
275.     public static ValueDateTupel getElement(ValueDateTupel[] valueDateTupels,
276.         LocalDateTime dtToBeFound) {
277.       /* Check if given array is null */
278.       if (valueDateTupels == null)
279.         throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
280.
281.       if (dtToBeFound == null)
282.         throw new IllegalArgumentException(MESSAGE_VALUE_MUST_NOT_BE_NULL);
283.
284.       for (ValueDateTupel value : valueDateTupels) {
285.         if (value.getDate().equals(dtToBeFound))
286.           return value;
287.       }
288.       return null;
289.     }
290.
291.     /**
292.      * Get all elements between two given DateTimes (inclusive) from the
293.      * given array. If null is passed for either LocalDateTime, the
294.      * representing border will be set to the boundaries of the given array.
295.      *
296.      * @param valueDateTupels {@code ValueDateTupel[]} The base array.
297.      * @param dtFrom          {@link LocalDateTime} The first DateTime to be
298.      *                        included. If null, all values up until dtTo
299.      *                        will be given.
300.      * @param dtTo            {@link LocalDateTime} The last DateTime to be
301.      *                        included. If null, all values starting from
302.      *                        dtFrom will be given.
303.      * @return {@code ValueDateTupel[]} The found elements. null, if dtFrom
304.      *         or dtTo cannot be found in the given array.
305.      */
306.     public static ValueDateTupel[] getElements(
307.         ValueDateTupel[] valueDateTupels, LocalDateTime dtFrom,
308.         LocalDateTime dtTo) {
309.       int positionFrom;
310.       int positionTo;
311.       /*
312.        * Get the position indices of the given LocalDateTime values. If null
313.        * is passed, set the positions to be the respective boundaries of the
314.        * given array.
315.        */
316.       if (dtFrom == null) {
317.         positionFrom = 0;
318.       } else {
319.         positionFrom = ValueDateTupel.getPosition(valueDateTupels, dtFrom);
```

```
320.      }
321.      if (dtTo == null) {
322.        positionTo = valueDateTupels.length - 1;
323.      } else {
324.        positionTo = ValueDateTupel.getPosition(valueDateTupels, dtTo);
325.      }
326.
327.      /*
328.       * If the given LocalDateTime values cannot be found in the given array
329.       * return null.
330.       */
331.      if (positionFrom == Integer.MIN_VALUE
332.          || positionTo == Integer.MIN_VALUE)
333.        return null;
334.
335.      ValueDateTupel[] elements = {};
336.      /* Add all elements between the two found positions ... */
337.      for (int i = positionFrom; i <= positionTo; i++) {
338.        elements = ArrayUtils.add(elements, valueDateTupels[i]);
339.      }
340.      /* ... and return them. */
341.      return elements;
342.    }
343.
344.    /**
345.     * Get all {@link LocalDateTime} from an array of {@link ValueDateTupel}.
346.     *
347.     * @param valueDateTupels {@code ValueDateTupel[]} An array of
348.     *                        {@link ValueDateTupel}}.
349.     * @return {@code LocalDateTime[]} An array of {@link LocalDateTime} of
350.     *         the given {@code ValueDateTupel[]}. Returns an empty array if
351.     *         the given array is empty.
352.     * @throws IllegalArgumentException if the given array is null.
353.     */
354.    public static LocalDateTime[] getDates(
355.        ValueDateTupel[] valueDateTupels) {
356.      if (valueDateTupels == null)
357.        throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
358.
359.      LocalDateTime[] values = {};
360.      for (ValueDateTupel tupel : valueDateTupels) {
361.        values = ArrayUtils.add(values, tupel.getDate());
362.      }
363.      return values;
364.    }
365.
366.    /**
367.     * Finds the position of a given {@link LocalDateTime} in a given array
368.     * of {@link ValueDateTupel}.
369.     *
370.     * @param valueDateTupels {@code ValueDateTupel[]} The array to be
371.     *                        searched.
372.     * @param dtToBeFound     {@link LocalDateTime} The value to be found.
373.     * @return {@code int} The position the given LocalDateTime was found. If
```

```
374.      *          the given LocalDateTime cannot be found, Integer.MIN_VALUE is
375.      *          returned. If the given array's length is 0 Integer.MIN_VALUE
376.      *          is returned.
377.      */
378.     public static int getPosition(ValueDateTupel[] valueDateTupels,
379.         LocalDateTime dtToBeFound) {
380.       final int defaultReturnValue = Integer.MIN_VALUE;
381.
382.       /* Check if given array is null */
383.       if (valueDateTupels == null)
384.         throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
385.
386.       /*
387.        * If there are no values in the given array return the default return
388.        * value.
389.        */
390.       if (valueDateTupels.length == 0)
391.         return defaultReturnValue;
392.
393.       /* Check if given LocalDateTime is null */
394.       if (dtToBeFound == null)
395.         throw new IllegalArgumentException(MESSAGE_VALUE_MUST_NOT_BE_NULL);
396.
397.       /*
398.        * if the given LocalDateTime is in the given array, return its
399.        * position.
400.        */
401.       for (int i = 0; i < valueDateTupels.length; i++) {
402.         if (valueDateTupels[i].getDate().equals(dtToBeFound))
403.           return i;
404.       }
405.
406.       /* Otherwise, return Integer.MIN_VALUE */
407.       return defaultReturnValue;
408.     }
409.
410.     /**
411.      * Get all values from an array of {@link ValueDateTupel}.
412.      *
413.      * @param valueDateTupels {@code ValueDateTupel[]} An array of
414.      *                        {@link ValueDateTupel}}.
415.      * @return {@code double[]} An array of values of the given
416.      *         {@code ValueDateTupel[]}. Returns an empty array if the given
417.      *         array is empty.
418.      * @throws IllegalArgumentException if the given array is null.
419.      */
420.     public static double[] getValues(ValueDateTupel[] valueDateTupels) {
421.       if (valueDateTupels == null)
422.         throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
423.
424.       double[] values = {};
425.       for (ValueDateTupel tupel : valueDateTupels)
426.         values = ArrayUtils.add(values, tupel.getValue());
427.       return values;
```

```
428.     }
429.
430.     /**
431.      * Evaluate if the given array of {@link ValueDateTupel} is sorted in
432.      * ascending order, i.e., if the value at position 0 has the lowest
433.      * {@link LocalDateTime} value (implicit check) and all subsequent
434.      * {@link ValueDateTupel} each have a {@link LocalDateTime} after
435.      * ({@link LocalDateTime#isAfter(ChronoLocalDateTime)}) the previous one.
436.      * <p>
437.      * If two values have the same {@link LocalDateTime} false will be
438.      * returned.
439.      *
440.      * @param valueDateTupels {@code ValueDateTupel[]} array of
441.      *                        {@link ValueDateTupel} to be checked for
442.      *                        ascending order.
443.      * @return {@code boolean} False, if any date is not chronologically
444.      *         after its predecessor, true if otherwise.
445.      * @throws IllegalArgumentException If the given array is null.
446.      * @throws IllegalArgumentException If the given array contains null
447.      *                                  values.
448.      */
449.     public static boolean isSortedAscending(
450.         ValueDateTupel[] valueDateTupels) {
451.       if (valueDateTupels == null)
452.         throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
453.       if (ValueDateTupel.contains(valueDateTupels, null))
454.         throw new IllegalArgumentException(
455.             "The given array must not contain any nulls");
456.
457.       for (int i = 1; i < valueDateTupels.length; i++) {
458.         if (!valueDateTupels[i].getDate()
459.             .isAfter(valueDateTupels[i - 1].getDate())) {
460.           return false;
461.         }
462.       }
463.       return true;
464.     }
465.
466.     /**
467.      * Evaluate if the given array of {@link ValueDateTupel} is sorted in
468.      * descending order, i.e., if the value at position 0 has the highest
469.      * {@link LocalDateTime} value (implicit check) and all subsequent
470.      * {@link ValueDateTupel} each have a {@link LocalDateTime} before
471.      * ({@link LocalDateTime#isBefore(ChronoLocalDateTime)}) the previous
472.      * one.
473.      * <p>
474.      * If two values have the same {@link LocalDateTime} false will be
475.      * returned.
476.      *
477.      * @param valueDateTupels {@code ValueDateTupel[]} array of
478.      *                        {@link ValueDateTupel} to be checked for
479.      *                        descending order.
480.      * @return {@code boolean} False, if any date is not chronologically
481.      *         before its predecessor, true if otherwise.
```

```
482.      * @throws IllegalArgumentException If the given array is null.
483.      * @throws IllegalArgumentException If the given array contains null
484.      *                                  values.
485.      */
486.     public static boolean isSortedDescending(
487.         ValueDateTupel[] valueDateTupels) {
488.       if (valueDateTupels == null)
489.         throw new IllegalArgumentException(MESSAGE_ARRAY_MUST_NOT_BE_NULL);
490.       if (ValueDateTupel.contains(valueDateTupels, null))
491.         throw new IllegalArgumentException(
492.             "The given array must not contain any null LocalDateTime");
493.
494.       for (int i = 1; i < valueDateTupels.length; i++) {
495.         if (!valueDateTupels[i].getDate()
496.             .isBefore(valueDateTupels[i - 1].getDate())) {
497.           return false;
498.         }
499.       }
500.       return true;
501.     }
502.
503.     /**
504.      * Finds all {@link LocalDateTime} present in uniqueSortedDates but not
505.      * in valueDateTupels and add them to the latter with a value of
506.      * Double.NaN.
507.      *
508.      * @param valueDateTupels   {@code ValueDateTupel[]} The array of
509.      *                          {@link ValueDateTupel} to be enhanced.
510.      * @param uniqueSortedDates {@code TreeSet<LocalDateTime>} A
511.      *                          {@link TreeSet} of {@link LocalDateTime}
512.      *                          containing all unique LocalDateTimes.
513.      * @return {@code ValueDateTupel[]} valueDateTupels + all LocalDateTime
514.      *         additionally given by uniqueSortedDates. Array is sorted as by
515.      *         {@link #isSortedAscending(ValueDateTupel[])}.
516.      */
517.     private static ValueDateTupel[] enhanceRowByNaNs(
518.         ValueDateTupel[] valueDateTupels,
519.         TreeSet<LocalDateTime> uniqueSortedDates) {
520.
521.       /*
522.        * Load unique sorted dates into an ArrayList to have access to an
523.        * index.
524.        */
525.       List<LocalDateTime> uniqueSortedDatesList = new ArrayList<>(
526.           uniqueSortedDates);
527.
528.       /*
529.        * Loop over all unique dateTimes to assess if they are in the current
530.        * row. If not, missing dateTimes are added into the original arrays.
531.        * Their value is set to Double.NaN
532.        */
533.       for (int fieldIndex = 0; fieldIndex < uniqueSortedDates
534.           .size(); fieldIndex++) {
535.         ValueDateTupel valueDateTupelToBeAdded = new ValueDateTupel(
```

```
536.             uniqueSortedDatesList.get(fieldIndex), Double.NaN);
537.
538.         if (fieldIndex < valueDateTupels.length
539.             && uniqueSortedDatesList.get(fieldIndex)
540.                 .isEqual(valueDateTupels[fieldIndex].getDate())) {
541.
542.           /*
543.            * Nothing has to be done, as we're not at the end of the list and
544.            * the current LocalDateTime out of the list of unique values is
545.            * already in the given row.
546.            */
547.           continue;
548.         }
549.
550.         valueDateTupels = ValueDateTupel.addOneAt(valueDateTupels,
551.             valueDateTupelToBeAdded, fieldIndex);
552.       }
553.
554.       return valueDateTupels;
555.     }
556.
557.     /**
558.      * Fills up a gap of NaN-values in a given array of
559.      * {@link ValueDateTupel} with the average of the previously last and
560.      * first next available non-NaN-value.
561.      *
562.      * @param valueDateTupels   {@code ValueDateTupel} Array of
563.      *                          {@link ValueDateTupel} holding the values.
564.      * @param previousAvailable {@code int} Index of the last available
565.      *                          non-NaN-value before the gap to be filled.
566.      * @param nextAvailable     {@code int} Index of the first available
567.      *                          non-NaN-value after the gap to be filled.
568.      * @return {@code ValueDateTupel} Array of {@link ValueDateTupel} with
569.      *         the gap filled.
570.      */
571.     private static ValueDateTupel[] fillCenterValues(
572.         ValueDateTupel[] valueDateTupels, int previousAvailable,
573.         int nextAvailable) {
574.       /*
575.        * The value to be set to all missing values is the average of the last
576.        * non-NaN before the NaNs and the first non-NaN after the NaNs. This
577.        * is the case when the missing NaNs are not at the beginning or the
578.        * end of the given array.
579.        */
580.       double valueToBeSet = (valueDateTupels[previousAvailable].getValue()
581.           + valueDateTupels[nextAvailable].getValue()) / 2;
582.
583.       int localIndex = previousAvailable + 1;
584.       /* Fill all values up to the next NaN with the calculated value. */
585.       while (localIndex < nextAvailable) {
586.         valueDateTupels[localIndex].setValue(valueToBeSet);
587.         localIndex++;
588.       }
589.       return valueDateTupels;
```

```
590.    }
591.
592.    /**
593.     * Fills the values at the beginning of the array. If the first
594.     * valueDateTupel contains Double.NaN, its value will be set to match the
595.     * next non-NaN-value. If the following values are also Double.NaN,
596.     * iterate through the array until a value != Double.NaN is found.
597.     *
598.     * @param valueDateTupels {@code ValueDateTupel[]} An array of
599.     *                        {@link ValueDateTupel} to be filled.
600.     * @return {@code ValueDateTupel[]} The same array of
601.     *         {@link ValueDateTupel} but with starting values filled.
602.     * @throws IllegalArgumentException if the row only contains Double.NaN.
603.     */
604.    private static ValueDateTupel[] fillStartingValues(
605.        ValueDateTupel[] valueDateTupels) {
606.      int localFieldIndex = 1;
607.      /* Iterate through the array until a value != Double.NaN is found */
608.      while (Double.isNaN(valueDateTupels[localFieldIndex].getValue())) {
609.        localFieldIndex++;
610.      }
611.
612.      /*
613.       * If only one value has to be set execution can continue with the next
614.       * loop iteration
615.       */
616.      if (localFieldIndex == 1) {
617.        valueDateTupels[localFieldIndex - 1]
618.            .setValue(valueDateTupels[localFieldIndex].getValue());
619.      }
620.
621.      /*
622.       * If multiple values have to be set iterate over them an fill them
623.       * subsequently, starting from the last NaN before the first valid
624.       * value.
625.       */
626.      while (localFieldIndex >= 1) {
627.        valueDateTupels[localFieldIndex - 1]
628.            .setValue(valueDateTupels[localFieldIndex].getValue());
629.        localFieldIndex--;
630.      }
631.
632.      return valueDateTupels;
633.    }
634.
635.    /**
636.     * Get unique dates from an array of arrays of {@link ValueDateTupel}.
637.     *
638.     * @param valueDateTupels {@code ValueDateTupel[][]} The array of arrays
639.     *                        of {@link ValueDateTupel} the get all unique
640.     *                        dates from.
641.     * @return {@code TreeSet<LocalDateTime>} A TreeSet of all unique dates.
642.     */
643.    private static TreeSet<LocalDateTime> getUniqueDates(
```

```
644.          ValueDateTupel[][] valueDateTupels) {
645.        TreeSet<LocalDateTime> uniqueSortedDates = new TreeSet<>();
646.
647.        /* For each array in ValueDateTupels ... */
648.        for (int rowIndex = 0; rowIndex < valueDateTupels.length; rowIndex++) {
649.
650.          try {
651.            Validator.validateValues(valueDateTupels[rowIndex]);
652.            Validator.validateDates(valueDateTupels[rowIndex]);
653.          } catch (IllegalArgumentException e) {
654.            throw new IllegalArgumentException("The array at position "
655.                + rowIndex + " does not meet specifications.", e);
656.          }
657.
658.          /* ... add all values into uniqueSortedDates */
659.          uniqueSortedDates.addAll(Arrays
660.              .asList(ValueDateTupel.getDates(valueDateTupels[rowIndex])));
661.        }
662.        return uniqueSortedDates;
663.      }
664.
665.      /**
666.       * Replace Double.NaN in a given row by real values values.
667.       *
668.       * @param valueDateTupels {@code ValueDateTupel[]} An array of
669.       *                        {@link ValueDateTupel} to be enhanced.
670.       * @return {@code ValueDateTupel[]} The same array of
671.       *         {@link ValueDateTupel} but with real values instead of
672.       *         Double.NaN.
673.       */
674.      private static ValueDateTupel[] replaceNansByValues(
675.          ValueDateTupel[] valueDateTupels) {
676.        /*
677.         * Loop over all dateTimes for each row to assess if they are
678.         * Double.NaN.
679.         */
680.        for (int fieldIndex = 0;
681.            fieldIndex < valueDateTupels.length;
682.            fieldIndex++) {
683.          /*
684.           * If the ValueDateTupel contains a value other than Double.NaN
685.           * continue with the next iteration.
686.           */
687.          if (!Double.isNaN(valueDateTupels[fieldIndex].getValue()))
688.            continue;
689.
690.          /*
691.           * If the first valueDateTupel contains Double.NaN, its value will be
692.           * set to match the next non-NaN-value. If the following values are
693.           * also Double.NaN, iterate through the array until a value !=
694.           * Double.NaN is found.
695.           */
696.          if (fieldIndex == 0) {
697.            valueDateTupels = fillStartingValues(valueDateTupels);
```

```
698.          continue;
699.        }
700.
701.        /*
702.         * The missing value will be set to average the values of its direct
703.         * predecessor and successor.
704.         *
705.         * If there are multiple values missing in a row, all of those will
706.         * get the average value of the last position before and the first
707.         * position after all missing values.
708.         *
709.         * If all values until the last one are missing, all consecutive NaN
710.         * values will be set to the last position before all NaNs.
711.         */
712.        int limitIndex = fieldIndex;
713.
714.        double valueToBeSet = Double.NaN;
715.
716.        while (Double.isNaN(valueDateTupels[limitIndex].getValue())) {
717.          limitIndex++;
718.          /*
719.           * If there are no values in the remaining array set all values to
720.           * be the last non-NaN, which is at fieldIndex-1.
721.           */
722.          if (limitIndex == valueDateTupels.length) {
723.            limitIndex--;
724.            valueToBeSet = valueDateTupels[fieldIndex - 1].getValue();
725.            break;
726.          }
727.
728.        }
729.
730.        /*
731.         * If the value to be set has already been calculated then there is
732.         * NaNs left in the array, no more real values, until the very last
733.         * position. All remaining values can be set to this value then.
734.         * After that the loop can be left.
735.         */
736.        if (!Double.isNaN(valueToBeSet)) {
737.          int localIndex = fieldIndex;
738.          while (localIndex < valueDateTupels.length) {
739.            valueDateTupels[localIndex].setValue(valueToBeSet);
740.            localIndex++;
741.          }
742.          break;
743.        }
744.
745.        /* Fill in a "gap" of Double.NaN-values */
746.        valueDateTupels = fillCenterValues(valueDateTupels, fieldIndex - 1,
747.            limitIndex);
748.
749.      }
750.      return valueDateTupels;
751.    }
```

```
752.
753.     /**
754.      * ========================================================================
755.      * OVERRIDES
756.      * ========================================================================
757.      */
758.
759.     /**
760.      * A hash code for this ValueDateTupel.
761.      */
762.     @GeneratedCode
763.     @Override
764.     public int hashCode() {
765.       final int prime = 31;
766.       int result = 1;
767.       result = prime * result + ((date == null) ? 0 : date.hashCode());
768.       long temp;
769.       temp = Double.doubleToLongBits(value);
770.       result = prime * result + (int) (temp ^ (temp >>> 32));
771.       return result;
772.     }
773.
774.     /**
775.      * Checks if this ValueDateTupel is equal to another ValueDateTupel.
776.      */
777.     @GeneratedCode
778.     @Override
779.     public boolean equals(Object obj) {
780.       if (this == obj)
781.         return true;
782.       if (obj == null)
783.         return false;
784.       if (getClass() != obj.getClass())
785.         return false;
786.       ValueDateTupel other = (ValueDateTupel) obj;
787.       if (date == null) {
788.         if (other.date != null)
789.           return false;
790.       } else if (!date.equals(other.date))
791.         return false;
792.       if (Double.doubleToLongBits(value) != Double
793.           .doubleToLongBits(other.value))
794.         return false;
795.       return true;
796.     }
797.
798.     /**
799.      * Outputs the fields of this ValueDateTupel as a {@code String}.
800.      */
801.     @GeneratedCode
802.     @Override
803.     public String toString() {
804.       StringBuilder builder = new StringBuilder();
805.       builder.append("ValueDateTupel [value=");
```

```
806.       builder.append(value);
807.       builder.append(", date=");
808.       builder.append(date);
809.       builder.append("]");
810.       return builder.toString();
811.     }
812.
813.     /**
814.      * ========================================================================
815.      * GETTERS AND SETTERS
816.      * ========================================================================
817.      */
818.
819.     /**
820.      * Get the value of a {@link ValueDateTupel}
821.      *
822.      * @return {@code double} value of the {@link ValueDateTupel}
823.      */
824.     public double getValue() {
825.       return value;
826.     }
827.
828.     /**
829.      * Set the value of a {@link ValueDateTupel}
830.      *
831.      * @param value {@code double} value to be set
832.      */
833.     public void setValue(double value) {
834.       this.value = value;
835.     }
836.
837.     /**
838.      * Get the date of a {@link ValueDateTupel}
839.      *
840.      * @return {@link LocalDateTime} date for the {@link ValueDateTupel}
841.      */
842.     public LocalDateTime getDate() {
843.       return date;
844.     }
845.
846.     /**
847.      * Set the date for a {@link ValueDateTupel}
848.      *
849.      * @param date {@link LocalDateTime} date to be set for the
850.      *             {@link ValueDateTupel}
851.      */
852.     public void setDate(LocalDateTime date) {
853.       this.date = date;
854.     }
855.   }
```

Komponente Util

Listing 21: Komponente Util

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  import java.util.DoubleSummaryStatistics;
4.  import java.util.stream.DoubleStream;
5.
6.  import org.apache.commons.lang3.ArrayUtils;
7.  import org.apache.commons.math3.linear.BlockRealMatrix;
8.  import org.apache.commons.math3.linear.RealMatrix;
9.  import org.apache.commons.math3.stat.correlation.PearsonsCorrelation;
10.
11. import de.rumford.tradingsystem.Rule;
12.
13. /**
14.  * Utility class as used throughout the library containing solely of static
15.  * methods representing mainly mathematical calculations.
16.  *
17.  * @author Max Rumford
18.  *
19.  */
20. public final class Util {
21.
22.   /**
23.    * Constructor for the {@link Util} class<br/>
24.    * Only supports static methods, hence no instance shall be created,
25.    * hence a private constructor
26.    */
27.   private Util() {
28.   }
29.
30.   /**
31.    * Adjusts a given value for the given standard deviation
32.    *
33.    * @param value             {@code double} value to be adjusted
34.    * @param standardDeviation {@code double} standard deviation to be
35.    *                          adjusted for
36.    * @return {@code double} standard deviation adjusted value. Double.NaN,
37.    *         if the given standard deviation is zero.
38.    */
39.   public static double adjustForStandardDeviation(double value,
40.       double standardDeviation) {
41.     if (standardDeviation == 0)
42.       return Double.NaN;
43.     return value / standardDeviation;
44.   }
45.
46.   /**
47.    * Check if the given rules are unique by utilizing
48.    * {@link Rule#equals(Object)}
49.    *
```

```
     * @param rules {@code Rule} An array of rules to be check for
     *              uniqueness.
     * @return {@code boolean} True, if the rules are unique. False
     *         otherwise.
     */
    public static boolean areRulesUnique(Rule[] rules) {
      if (rules == null)
        throw new IllegalArgumentException(
            "The given rules must not be null");
      if (ArrayUtils.contains(rules, null))
        throw new IllegalArgumentException(
            "The given array must not contain nulls");
      if (rules.length == 0)
        throw new IllegalArgumentException(
            "The given array of rules must not be empty.");

      for (int i = 0; i < rules.length - 1; i++) {
        if (rules[i].equals(rules[i + 1])) {
          return false;
        }
      }
      return true;
    }

    /**
     * Calculates the average value of the given array of values.
     *
     * @param values {@code double[]} An array of values.
     * @return {@code double} The average value of the given values.
     * @throws IllegalArgumentException if the given array is null.
     */
    public static double calculateAverage(double[] values) {
      Validator.validateArrayOfDoubles(values);

      /* Calculate the average of absolute values */
      DoubleSummaryStatistics stats = new DoubleSummaryStatistics();
      /* Load absolute values into stats */
      for (double value : values)
        stats.accept(value);
      /* Get average of all values */
      return stats.getAverage();
    }

    /**
     * Calculate the correlations between the given array of value arrays.
     *
     * @param valuesMatrix {@code double[][]} An array of n arrays of values.
     * @return {@code double[]} The correlations between the given rows as by
     *         {@link PearsonsCorrelation#getCorrelationMatrix()}.
     * @throws IllegalArgumentException if the given array is null.
     * @throws IllegalArgumentException if the given array does not contain
     *                                  exactly 3 values.
     * @throws IllegalArgumentException if the given array contains null.
     * @throws IllegalArgumentException if the given array contains arrays
```

```
104.      *                                   containing null.
105.      * @throws IllegalArgumentException if the given array contains arrays
106.      *                                   containing Double.NaN.
107.      * @throws IllegalArgumentException if the given array contains arrays
108.      *                                   not of the same length.
109.      *
110.      */
111.     public static double[] calculateCorrelationOfRows(
112.         double[][] valuesMatrix) {
113.       /*
114.        * If one of the rows contains all identical values no correlation can
115.        * be calculated, as a division by zero will occur in correlations
116.        * calculation.
117.        */
118.       for (int i = 0; i < valuesMatrix.length; i++) {
119.         double[] noDuplicates = DoubleStream.of(valuesMatrix[i]).distinct()
120.             .toArray();
121.         if (noDuplicates.length == 1) {
122.           throw new IllegalArgumentException(
123.               "Correlations cannot be calculated caused by all identical "
124.                   + "values in row at position " + i + ".");
125.         }
126.       }
127.
128.       /* Load the given values into rows of a matrix */
129.       RealMatrix matrix = new BlockRealMatrix(valuesMatrix);
130.       /* Transpose the values into columns to get the correct correlations */
131.       matrix = matrix.transpose();
132.
133.       /* Get the correlations of the passed value arrays */
134.       PearsonsCorrelation pearsonsCorrelations = new PearsonsCorrelation(
135.           matrix);
136.       RealMatrix correlationMatrix = pearsonsCorrelations
137.           .getCorrelationMatrix();
138.
139.       double[] correlations = {};
140.       for (int rowIndex = 0; rowIndex < correlationMatrix
141.           .getRowDimension(); rowIndex++) {
142.         for (int columnIndex = 0; columnIndex < correlationMatrix
143.             .getColumnDimension(); columnIndex++) {
144.           if (columnIndex < rowIndex)
145.             correlations = ArrayUtils.add(correlations,
146.                 correlationMatrix.getEntry(rowIndex, columnIndex));
147.         }
148.       }
149.       return correlations;
150.     }
151.
152.     /**
153.      * Scales the forecast based on the given scalar
154.      *
155.      * @param unscaledForecast {@code double} unscaled forecast to be scaled
156.      * @param scalar           {@code double} scalar to scale the unscaled
157.      *                         forecast
```

```
158.      * @return {@code double} the scaled forecast
159.      */
160.     public static double calculateForecast(double unscaledForecast,
161.         double scalar) {
162.       return unscaledForecast * scalar;
163.     }
164.
165.     /**
166.      * Calculates the forecast scalar for the given array of values in the
167.      * scale of the given base scale.
168.      *
169.      * Formula: F = baseScale / [ sum( |fc| ) / n ]
170.      *
171.      * Base scale divided by the average of absolutes.
172.      *
173.      * @param values    {@code double[]} values to be scaled by the forecast
174.      *                  scalar
175.      * @param baseScale {@code double} base scale for scaling of the forecast
176.      *                  scalar
177.      * @return {@code double} forecast scalar to scale the given values to
178.      *         fit the given scalar base. Returns Double.NaN if the average
179.      *         of absolute values is 0
180.      * @throws IllegalArgumentException if the average of the absolutes of
181.      *                                  the given values is zero
182.      * @throws IllegalArgumentException if the given baseScale is zero
183.      */
184.     public static double calculateForecastScalar(double[] values,
185.         double baseScale) {
186.
187.       Validator.validateArrayOfDoubles(values);
188.       if (values.length == 0)
189.         throw new IllegalArgumentException(
190.             "Given array of values must not be empty");
191.
192.       try {
193.         Validator.validatePositiveDouble(baseScale);
194.       } catch (IllegalArgumentException e) {
195.         throw new IllegalArgumentException(
196.             "Given base scale does not meet specifications.", e);
197.       }
198.
199.       /* helper array */
200.       double[] absoluteValues = new double[values.length];
201.
202.       /* Calculate the absolute values for all values in the given array */
203.       for (int i = 0; i < values.length; i++)
204.         absoluteValues[i] = Math.abs(values[i]);
205.
206.       /* Get average of all values */
207.       double averageOfAbsolutes = Util.calculateAverage(absoluteValues);
208.
209.       if (averageOfAbsolutes == 0)
210.         return Double.NaN;
211.
```

```
212.       return baseScale / averageOfAbsolutes;
213.     }
214.
215.     /**
216.      * Calculates the difference between two values in percentage points of
217.      * change as seen from the former value
218.      *
219.      * @param formerValue {@code double} value the value of the difference is
220.      *                    based on
221.      * @param latterValue {@code double} "new" value which represents a
222.      *                    changed value in comparison to formerValue
223.      * @return {@code double} difference between formerValue and latterValue
224.      *         represented in percentage points. Double.NaN if the given
225.      *         formerValue is zero.
226.      */
227.     public static double calculateReturn(double formerValue,
228.         double latterValue) {
229.       if (formerValue == 0)
230.         return Double.NaN;
231.       return latterValue / formerValue - 1d;
232.     }
233.
234.     /**
235.      * Calculate the weights that should be given to the rows of values
236.      * making up the given correlations. Expects an array of length 3, where
237.      * position 0 holds the correlation of rows A and B, position 1 holds the
238.      * correlation for rows A and C, and position 2 holds the correlation for
239.      * rows B and C.
240.      *
241.      * @param correlations {@code double[]} Three values representing the
242.      *                     correlations between the rows A, B and C. The
243.      *                     expected array is constructed as follows: {
244.      *                     corr_AB, corr_AC, corr_BC }. See
245.      *                     {@link Validator#validateCorrelations(double[])}
246.      *                     for limitations.
247.      * @return {@code double[]} The calculated weights { w_A, w_B, w_C }.
248.      */
249.     public static double[] calculateWeightsForThreeCorrelations(
250.         double[] correlations) {
251.       Validator.validateCorrelations(correlations);
252.
253.       for (int i = 0; i < correlations.length; i++) {
254.         /*
255.          * Floor negative correlations at 0 (See Carver:
256.          * "Systematic Trading", p. 79)
257.          */
258.         if (correlations[i] < 0)
259.           correlations[i] = 0;
260.       }
261.
262.       double[] weights = {};
263.       /*
264.        * Catch three equal correlations. Three correlations of 1 each would
265.        * break further calculation.
```

```
266.        */
267.       if (correlations[0] == correlations[1]
268.           && correlations[0] == correlations[2]) {
269.         double correlationOfOneThird = 1d / 3d;
270.         weights = ArrayUtils.add(weights, correlationOfOneThird);
271.         weights = ArrayUtils.add(weights, correlationOfOneThird);
272.         weights = ArrayUtils.add(weights, correlationOfOneThird);
273.         return weights;
274.       }
275.
276.       /* Get the average correlation each row of values has */
277.       double averageCorrRowA = (correlations[0] + correlations[1]) / 2;
278.       double averageCorrRowB = (correlations[0] + correlations[2]) / 2;
279.       double averageCorrRowC = (correlations[1] + correlations[2]) / 2;
280.
281.       double[] averageCorrelations = { averageCorrRowA, averageCorrRowB,
282.           averageCorrRowC };
283.
284.       /*
285.        * Subtract each average correlation from 1 to get an inverse-ish value
286.        */
287.       for (int i = 0; i < averageCorrelations.length; i++)
288.         averageCorrelations[i] = 1 - averageCorrelations[i];
289.
290.       /* Calculate the sum of average correlations. */
291.       double sumOfAverageCorrelations = DoubleStream.of(averageCorrelations)
292.           .sum();
293.
294.       /*
295.        * Normalize the average correlations so they sum up to 1. These
296.        * normalized values are the weights.
297.        */
298.       for (int i = 0; i < averageCorrelations.length; i++)
299.         weights = ArrayUtils.add(weights,
300.             averageCorrelations[i] / sumOfAverageCorrelations);
301.
302.       return weights;
303.     }
304.
305.     /**
306.      * Returns the position literal for a given forecast.
307.      * <ul>
308.      * <li>"Long" for forecasts greater 0.</li>
309.      * <li>"Short" for forecasts less than 0.</li>
310.      * <li>"Hold" for forecasts of 0.</li>
311.      * </ul>
312.      *
313.      * @param forecast a forecast.
314.      * @return The String literal for the given forecast.
315.      */
316.     public static String getPositionFromForecast(double forecast) {
317.       if (forecast > 0)
318.         return "Long";
319.       if (forecast < 0)
```

```
320.         return "Short";
321.       return "Hold";
322.     }
323.   }
```

Komponente Validator

Listing 22: Komponente Validator

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  import java.time.LocalDateTime;
4.  import java.util.HashSet;
5.  import java.util.Set;
6.
7.  import de.rumford.tradingsystem.BaseValue;
8.  import de.rumford.tradingsystem.Rule;
9.
10. /**
11.  * Validator class as used throughout the library containing solely of
12.  * static methods representing validation of input values.
13.  *
14.  * @author Max Rumford
15.  *
16.  */
17. public class Validator {
18.
19.   /**
20.    * Constructor for the {@link Validator} class<br/>
21.    * Only supports static methods, hence no instance shall be created,
22.    * hence a private constructor
23.    */
24.   private Validator() {
25.   }
26.
27.   /**
28.    * Validates the given array of arrays.
29.    * <ul>
30.    * <li>Must not be null.</li>
31.    * </ul>
32.    *
33.    * @param values {@code double[]} The array to be validated.
34.    * @throws IllegalArgumentException if any of the above specifications
35.    *                                  are not met.
36.    */
37.   public static void validateArrayOfDoubles(double[] values) {
38.     if (values == null)
39.       throw new IllegalArgumentException("Given array must not be null");
40.   }
41.
42.   /**
43.    * Validates the given {@link BaseValue}.
44.    * <ul>
45.    * <li>Must not be null.</li>
46.    * </ul>
47.    *
48.    * @param baseValue {@link BaseValue} The base value to be validated.
49.    */
```

```
50.     public static void validateBaseValue(BaseValue baseValue) {
51.       if (baseValue == null)
52.         throw new IllegalArgumentException("Base value must not be null");
53.     }
54.
55.     /**
56.      * Validates the given correlations.
57.      *
58.      * @param correlations {@code double[]} Correlations to be validated.
59.      *                     Must not be null. Must have a length of 3. Must
60.      *                     only contain values {@code !Double.NaN} and
61.      *                     {@code -1 <= value <= 1}.
62.      * @throws IllegalArgumentException if the above specifications are not
63.      *                                  met.
64.      */
65.     public static void validateCorrelations(double[] correlations) {
66.       /* Check if the given array is null */
67.       if (correlations == null)
68.         throw new IllegalArgumentException(
69.             "Correlations array must not be null");
70.       /* Check if the given array contains exactly three elements. */
71.       if (correlations.length != 3)
72.         throw new IllegalArgumentException(
73.             "There must be exactly three correlation values in the given"
74.                 + " array");
75.
76.       /* Check all given values inside the array */
77.       for (int i = 0; i < correlations.length; i++) {
78.         if (Double.isNaN(correlations[i]))
79.           throw new IllegalArgumentException(
80.               "NaN-values are not allowed. Correlation at position " + i
81.                   + " is NaN.");
82.         if (correlations[i] > 1)
83.           throw new IllegalArgumentException(
84.               "Correlation at position " + i + " is greater than 1");
85.         if (correlations[i] < -1)
86.           throw new IllegalArgumentException(
87.               "Correlation at position " + i + " is less than -1");
88.       }
89.     }
90.
91.     /**
92.      * Validates if the given array of ValueDateTupel is sorted in an
93.      * ascending order.
94.      *
95.      * @param values {@code ValueDateTupel[]} The given array of values.
96.      * @throws IllegalArgumentException if the specifications above are not
97.      *                                  met.
98.      */
99.     public static void validateDates(ValueDateTupel[] values) {
100.      /*
101.       * The values cannot be used if they are not in ascending order.
102.       */
103.      if (!ValueDateTupel.isSortedAscending(values))
```

```
104.          throw new IllegalArgumentException(
105.              "Given values are not properly sorted or there are non-unique"
106.                  + " values.");
107.      }
108.
109.      /**
110.       * Validates the given double value. The value must meet the following
111.       * specifications:
112.       * <ul>
113.       * <li>Must not be Double.NaN.</li>
114.       * <li>Must be a positive decimal.</li>
115.       * </ul>
116.       *
117.       * @param value {@code double} The value to validate.
118.       * @throws IllegalArgumentException if the above specifications are not
119.       *                                  met.
120.       */
121.      public static void validatePositiveDouble(double value) {
122.        if (Double.isNaN(value))
123.          throw new IllegalArgumentException("Value must not be Double.NaN");
124.
125.        if (value <= 0)
126.          throw new IllegalArgumentException(
127.              "Value must be a positive decimal");
128.
129.      }
130.
131.      /**
132.       * Validates the given row of values. The row has to have at least 1
133.       * value not Double.NaN.
134.       *
135.       * @param valueDateTupels {@code ValueDateTupel[]} The array of
136.       *                        {@link ValueDateTupel} to be validated.
137.       * @throws IllegalArgumentException if the given row contains only
138.       *                                  Double.NaN.
139.       */
140.      public static void validateRow(ValueDateTupel[] valueDateTupels) {
141.        double[] values = ValueDateTupel.getValues(valueDateTupels);
142.        /* If the first value is NaN, check if the array only contains NaN. */
143.        if (Double.isNaN(values[0])) {
144.          Set<Double> uniqueValues = new HashSet<>();
145.          for (double value : values)
146.            uniqueValues.add(value);
147.
148.          /*
149.           * If the size of a set of all values is 1 then it contains only this
150.           * one value in all elements. If this value is Double.NaN, no values
151.           * were set but Double.NaN.
152.           */
153.          if (uniqueValues.size() == 1)
154.            throw new IllegalArgumentException(
155.                "Row contains only Double.NaN. Rows must contain at least one"
156.                    + " value != Double.NaN");
157.        }
```

```
158.     }
159.
160.     /**
161.      * Validates the given Array of {@link Rule}. Must meet the following
162.      * specifications:
163.      * <ul>
164.      * <li>Must not be null.</li>
165.      * <li>Must not be an empty array.</li>
166.      * </ul>
167.      *
168.      * @param rules {@code Rule[]} The Array of {@link Rule} to be validated.
169.      * @throws IllegalArgumentException if the above specifications are not
170.      *                                  met.
171.      */
172.     public static void validateRules(Rule[] rules) {
173.       if (rules == null)
174.         throw new IllegalArgumentException("Rules must not be null");
175.
176.       if (rules.length == 0)
177.         throw new IllegalArgumentException(
178.             "Rules must not be an empty array");
179.     }
180.
181.     /**
182.      * Validates if the rules in the given array of {@link Rule} all share
183.      * the given base scale.
184.      *
185.      * @param rules     The array of {@link Rule} to be examined.
186.      * @param baseScale The comparing base scale.
187.      * @throws IllegalArgumentException if the given rules do not share the
188.      *                                  given base scale.
189.      */
190.     public static void validateRulesVsBaseScale(Rule[] rules,
191.         double baseScale) {
192.       if (rules != null) {
193.         for (int i = 0; i < rules.length; i++)
194.           if (rules[i] != null && rules[i].getBaseScale() != baseScale)
195.             throw new IllegalArgumentException("The rule at index " + i
196.                 + " does not share the given base scale of " + baseScale
197.                 + ".");
198.       }
199.     }
200.
201.     /**
202.      * Validates the given array of {@link Rule} against the given
203.      * {@link BaseValue}.
204.      * <ul>
205.      * <li>All rules must have the given {@link BaseValue} as their own
206.      * {@link BaseValue}.</li>
207.      * </ul>
208.      *
209.      * @param rules     {@code Rule[]} The array of {@link Rule} to be
210.      *                  validated.
211.      * @param baseValue {@link BaseValue} The {@link BaseValue} to be
```

```
212.      *                   validated.
213.      * @throws IllegalArgumentException if any of the above specifications
214.      *                                  are not met.
215.      */
216.     public static void validateRulesVsBaseValue(Rule[] rules,
217.         BaseValue baseValue) {
218.       for (int i = 0; i < rules.length; i++)
219.         if (!rules[i].getBaseValue().equals(baseValue))
220.           throw new IllegalArgumentException(
221.               "The base value of all rules must be equal to given base value"
222.                   + " but the rule at position " + i + " does not comply.");
223.     }
224.
225.     /**
226.      * Validates the given time window values. Values are needed for checking
227.      * of containment. Must meet the following specifications:
228.      * <ul>
229.      * <li>startOfTimeWindow must not be null.</li>
230.      * <li>endOfTimeWindow must not be null.</li>
231.      * <li>endOfTimeWindow must be after startOfTimeWindow. See
232.      * {@link LocalDateTime#isAfter(java.time.chrono.ChronoLocalDateTime)}.
233.      * </li>
234.      * <li>values must contain startOfTimeWindow.</li>
235.      * <li>values must contain endOfTimeWindow.</li>
236.      * </ul>
237.      *
238.      * @param startOfTimeWindow {@link LocalDateTime} The start of Time
239.      *                          window to be checked.
240.      * @param endOfTimeWindow   {@link LocalDateTime} The end of Time window
241.      *                          to be checked.
242.      * @param values            {@code ValueDateTupel[]} The array of
243.      *                          {@link ValueDateTupel} to be checked.
244.      * @throws IllegalArgumentException if any of the above specifications
245.      *                                  are not met.
246.      */
247.     public static void validateTimeWindow(LocalDateTime startOfTimeWindow,
248.         LocalDateTime endOfTimeWindow, ValueDateTupel[] values) {
249.       /* Check if LocalDateTimes are null */
250.       if (startOfTimeWindow == null)
251.         throw new IllegalArgumentException(
252.             "Start of time window value must not be null");
253.       if (endOfTimeWindow == null)
254.         throw new IllegalArgumentException(
255.             "End of time window value must not be null");
256.       /* Check if time window is properly defined: end must be after start */
257.       if (!endOfTimeWindow.isAfter(startOfTimeWindow))
258.         throw new IllegalArgumentException(
259.             "End of time window value must be after start of time window"
260.                 + " value");
261.
262.       /*
263.        * The given startOfTimeWindow must be included in the given base
264.        * values.
265.        */
```

```
266.      if (!ValueDateTupel.containsDate(values, startOfTimeWindow))
267.        throw new IllegalArgumentException(
268.            "Given values do not include given start value for time window");
269.      /*
270.       * The given endOfTimeWindow must be included in the given base values.
271.       */
272.      if (!ValueDateTupel.containsDate(values, endOfTimeWindow))
273.        throw new IllegalArgumentException(
274.            "Given values do not include given end value for time window");
275.    }
276.
277.    /**
278.     * Validates the given array of ValueDateTupel. The given array must
279.     * fulfill the following specifications:
280.     * <ul>
281.     * <li>Must not be null</li>
282.     * <li>Must be of length greater than 0</li>
283.     * <li>Must not contain null</li>
284.     * <li>Must not contain NaNs as values</li>
285.     * </ul>
286.     *
287.     * @param values {@code ValueDateTupel[]} The values to be validated.
288.     * @throws IllegalArgumentException if the given array does not meet the
289.     *                                  above specifications.
290.     */
291.    public static void validateValues(ValueDateTupel[] values) {
292.      /* Check if passed values array is null */
293.      if (values == null)
294.        throw new IllegalArgumentException(
295.            "The given values array must not be null");
296.      /* Check if passed values array contains elements */
297.      if (values.length == 0)
298.        throw new IllegalArgumentException(
299.            "Values must not be an empty array");
300.
301.      for (ValueDateTupel value : values) {
302.        /* Validate if there are null values in the given values array. */
303.        if (value == null)
304.          throw new IllegalArgumentException(
305.              "Given values must not contain null.");
306.
307.        /* Validate if there are NaN values in the given values array. */
308.        if (Double.isNaN(value.getValue()))
309.          throw new IllegalArgumentException(
310.              "Given values must not contain NaN.");
311.      }
312.    }
313.
314.    /**
315.     * Validates the given variations array of {@link Rule} against the given
316.     * {@link LocalDateTime} values for startOfReferenceWindow and
317.     * endOfReferenceWindow and the given {@link BaseValue}. Must fulfill the
318.     * following specifications:
319.     * <ul>
```

```
     * <li>Must not be of length greater than 3.</li>
     * <li>Must not be of length 0.</li>
     * <li>Elements must not be null.</li>
     * <li>Elements' {@link Rule#getStartOfReferenceWindow()} must equal the
     * given startOfReferenceWindow.</li>
     * <li>Elements' {@link Rule#getEndOfReferenceWindow()} must equal the
     * given endOfReferenceWindow.</li>
     * <li>Must pass
     * {@link #validateRulesVsBaseValue(Rule[], BaseValue)}.</li>
     * </ul>
     *
     * @param variations              {@code Rule[]} An array of {@link Rule}
     *                                representing a rules' variations.
     * @param startOfReferenceWindow {@link LocalDateTime} The start of
     *                                reference window of the {@link Rule}
     *                                containing the given variations.
     * @param endOfReferenceWindow   {@link LocalDateTime} The end of
     *                                reference window of the {@link Rule}
     *                                containing the given variations.
     * @param baseValue               {@link BaseValue} The base value of the
     *                                {@link Rule} containing the given
     *                                variations.
     */
    public static void validateVariations(Rule[] variations,
        LocalDateTime startOfReferenceWindow,
        LocalDateTime endOfReferenceWindow, BaseValue baseValue) {
      /* Check if there are too many variations for this rule */
      if (variations.length > 3)
        throw new IllegalArgumentException(
            "A rule must not contain more than 3 variations.");

      /* Check if the given variations array is empty. */
      if (variations.length == 0)
        throw new IllegalArgumentException(
            "The given variations array must not be empty.");

      for (int i = 0; i < variations.length; i++) {
        /* Check if the given variations array contains nulls. */
        if (variations[i] == null)
          throw new IllegalArgumentException("The variation at position " + i
              + " in the given variations array is null.");

        /* Check if main rule and variations share reference window. */
        if (!variations[i].getStartOfReferenceWindow()
            .equals(startOfReferenceWindow)) {
          throw new IllegalArgumentException(
              "The given reference window does not match the variation's at"
                  + " position " + i
                  + ". The given start of reference window is different.");
        }
        if (!variations[i].getEndOfReferenceWindow()
            .equals(endOfReferenceWindow)) {
          throw new IllegalArgumentException(
              "The given reference window does not match the variation's at"
```

```
                    + " position " + i
                    + ". The given end of reference window is different.");
          }
        }

        try {
          Validator.validateRulesVsBaseValue(variations, baseValue);
        } catch (IllegalArgumentException e) {
          throw new IllegalArgumentException(
              "The given variations do not meet specifications.", e);
        }
      }
    }
```

Komponente DataSource

Listing 23: Komponente DataSource

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  import java.io.*;
4.  import java.time.LocalDateTime;
5.  import java.util.regex.Pattern;
6.
7.  import org.apache.commons.lang3.ArrayUtils;
8.
9.  /**
10.  * The DataSource provides course value data from a given data source.
11.  *
12.  * @author Max Rumford
13.  *
14.  */
15. public class DataSource {
16.
17.   /**
18.    * Don't let anyone instantiate this class.
19.    */
20.   private DataSource() {
21.   }
22.
23.   /**
24.    * Reads the data from a given CSV path. Assumes the following "columns":
25.    * Date, Time, value. Depending on the formatting of the CSV file the
26.    * corresponding {@link CsvFormat} has to be passed. The underlying
27.    * enumeration is not final and can be altered to suit the user's needs.
28.    * <p>
29.    * The CSV file to be parsed is expected not to have column headings. If
30.    * so, the values of the first row might not be parsed and an
31.    * IllegalArgumentException as explained below might be thrown. If the
32.    * row can be parsed it will most likely not contain any useful
33.    * information and might result in incorrect calculation results. The CSV
34.    * file should always be cleared of headings.
35.    *
36.    * @param sourcePath {@code String} The path to the CSV file to be read.
37.    * @param format     {@link CsvFormat} The format of the CSV file.
38.    * @return {@code ValueDateTupel[]} An array of {@link ValueDateTupel}
39.    *         representing the read data.
40.    * @throws FileNotFoundException    if the FileReader can not find a file
41.    *                                  for the given {@code sourcePath}.
42.    * @throws IOException              if the given {@code sourcePath}
43.    *                                  cannot be properly resolved to an
44.    *                                  actual file.
45.    * @throws IllegalArgumentException if the given path is invalid.
46.    * @throws IllegalArgumentException if any of the rows in the read CSV
47.    *                                  file does not contain exactly 3
48.    *                                  columns.
49.    */
```

```
50.     public static ValueDateTupel[] getDataFromCsv(String sourcePath,
51.         CsvFormat format) throws IOException {
52.       File file;
53.       try {
54.         file = new File(sourcePath);
55.       } catch (Exception e) {
56.         throw new IllegalArgumentException(
57.             "The given path cannot be processed");
58.       }
59.
60.       if (!file.exists())
61.         throw new IOException(
62.             "Given source path does not point to an existing destination");
63.       if (!file.isFile())
64.         throw new IOException("Given source path does not point to a file");
65.       if (!file.canRead())
66.         throw new IOException("Given file path cannot be read");
67.
68.       try (BufferedReader br = new BufferedReader(new FileReader(file))) {
69.         String line;
70.
71.         ValueDateTupel[] returnValues = ValueDateTupel.createEmptyArray();
72.         while ((line = br.readLine()) != null) {
73.           /* Extract the fields into separate Strings */
74.           String[] columns = line
75.               .split(Pattern.quote(format.getFieldSeparator()));
76.
77.           if (columns.length != 3) {
78.             throw new IllegalArgumentException(
79.                 "The passed CSV does not have an appropriate number of"
80.                     + " columns");
81.           }
82.
83.           /*
84.            * Parse the first and second field (date, time) into a
85.            * LocalDateTime instance
86.            */
87.           String[] dateAndTimeStrings = new String[2];
88.           System.arraycopy(columns, 0, dateAndTimeStrings, 0, 2);
89.           LocalDateTime localDateTime;
90.           double value;
91.           localDateTime = parseLocalDateTime(dateAndTimeStrings, format);
92.
93.           /* Pass the third field (course value) into a double */
94.           String[] valueStrings = new String[1];
95.           System.arraycopy(columns, 2, valueStrings, 0, 1);
96.           value = parseCourseValue(valueStrings, format);
97.
98.           ValueDateTupel newElement = new ValueDateTupel(localDateTime,
99.               value);
100.
101.          returnValues = ArrayUtils.add(returnValues, newElement);
102.        }
103.        return returnValues;
```

```
104.       }
105.     }
106.
107.     /**
108.      * Parse the given columns {date, time} into a {@link LocalDateTime}
109.      * instance. Expects an array of Strings of length 2.
110.      *
111.      * @param columns {@code String[]} The columns containing the String
112.      *                representing date and time.
113.      * @param format  {@link CsvFormat} The {@link CsvFormat} representing
114.      *                the given CSV file.
115.      * @return {@link LocalDateTime} representation of the passed {date,
116.      *         time} columns.
117.      * @throws IllegalArgumentException If there are not exactly two columns
118.      *                                  in the passed {@code String[]}.
119.      * @throws IllegalArgumentException If the date pattern could not be
120.      *                                  recognized in subroutine
121.      *                                  {@link #evaluateDatePattern(
122.      *                                  CsvFormat)}.
123.      * @throws IllegalArgumentException If the given date values cannot be
124.      *                                  parsed to Integers.
125.      * @throws IllegalArgumentException If the given time values cannot be
126.      *                                  parsed to Integers.
127.      * @throws IllegalArgumentException If the given date and time values
128.      *                                  cannot be parsed to
129.      *                                  {@link LocalDateTime}}.
130.      */
131.     private static LocalDateTime parseLocalDateTime(String[] columns,
132.         CsvFormat format) {
133.       /* Extract the relevant date values */
134.       String[] date = columns[0]
135.           .split(Pattern.quote(format.getDateSeparator()));
136.
137.       /* Evaluate the date pattern */
138.       int[] datePositions = evaluateDatePattern(format);
139.
140.       int dayOfMonth;
141.       int month;
142.       int year;
143.       try {
144.         dayOfMonth = Integer.parseInt(date[datePositions[0]]);
145.         month = Integer.parseInt(date[datePositions[1]]);
146.         year = Integer.parseInt(date[datePositions[2]]);
147.       } catch (NumberFormatException e) {
148.         throw new IllegalArgumentException(
149.             "The date values of the read CSV file cannot be parsed into"
150.                 + " numbers. Failing value >" + columns[0] + "<");
151.       }
152.       /*
153.        * Catch Exception so BufferedReader can be closed (in calling method)
154.        * on unknown Exceptions to avoid memory leakage.
155.        */
156.       catch (Exception e) {
157.         throw e;
```

```
158.       }
159.
160.       /* Extract the relevant time values */
161.       String[] time = columns[1].split(format.getTimeSeparator());
162.       int hour;
163.       int minute;
164.       int second;
165.       try {
166.         hour = Integer.parseInt(time[0]);
167.         minute = Integer.parseInt(time[1]);
168.         second = Integer.parseInt(time[2]);
169.       } catch (NumberFormatException e) {
170.         throw new IllegalArgumentException(
171.             "The time values of the read CSV file cannot be parsed into"
172.                 + " numbers. Failing value >" + columns[1] + "<");
173.       }
174.
175.       LocalDateTime localDateTime;
176.       try {
177.         localDateTime = LocalDateTime.of(year, month, dayOfMonth, hour,
178.             minute, second);
179.       } catch (Exception e) {
180.         throw new IllegalArgumentException(
181.             "The date or time values of the read CSV file cannot be parsed"
182.                 + " into a LocalDateTime instance. Failing values >"
183.                 + columns[0] + "< and >" + columns[1] + "<.");
184.       }
185.       return localDateTime;
186.     }
187.
188.     /**
189.      * Evaluate the date pattern from the given {@link CsvFormat}. If
190.      * {@link CsvFormat} is extended this method has to be overridden, else
191.      * an {@link IllegalArgumentException} will be thrown due to an unknown
192.      * format.
193.      *
194.      * @param format {@link CsvFormat} Format of the CSV file to be parsed.
195.      * @return {@code int[]} containing the position of {day, month, year}
196.      *         values inside the date field of the given CSV.
197.      * @throws IllegalArgumentException if the date given date pattern is not
198.      *                                  recognized.
199.      */
200.     public static int[] evaluateDatePattern(CsvFormat format) {
201.       int monthPosition;
202.       int dayPosition;
203.       int yearPosition;
204.       if (format.getMonthDayOrder() == DateOrder.DAY_MONTH_YEAR) {
205.         dayPosition = 0;
206.         monthPosition = 1;
207.         yearPosition = 2;
208.       } else if (format.getMonthDayOrder() == DateOrder.MONTH_DAY_YEAR) {
209.         dayPosition = 1;
210.         monthPosition = 0;
211.         yearPosition = 2;
```

```
212.      } else {
213.        dayPosition = 2;
214.        monthPosition = 1;
215.        yearPosition = 0;
216.      }
217.
218.      int[] returnArray = { dayPosition, monthPosition, yearPosition };
219.
220.      return returnArray;
221.    }
222.
223.    /**
224.     * Parse the course value String into a {@code double} representing its
225.     * values.
226.     *
227.     * @param columns {@code String[]} The String representation of the CSV
228.     *                column containing the value.
229.     * @param format  {@link CsvFormat} Format the CSV file is in.
230.     * @return {@code double} The parsed value.
231.     * @throws IllegalArgumentException if the passed String cannot be
232.     *                                  properly parsed
233.     */
234.    private static double parseCourseValue(String[] columns,
235.        CsvFormat format) {
236.      String valueString = columns[0];
237.      /* Eliminate thousands separator from String */
238.      if (valueString.contains(format.getThousandsSeparator())) {
239.        valueString = valueString
240.            .replaceAll(Pattern.quote(format.getThousandsSeparator()), "");
241.      }
242.      /* Replace non-US decimal points with US decimal points */
243.      if (!format.getDecimalPoint().equals(CsvFormat.US.getDecimalPoint())) {
244.        valueString = valueString.replace(format.getDecimalPoint(),
245.            CsvFormat.US.getDecimalPoint());
246.      }
247.
248.      double value;
249.      /*
250.       * At this point all hindering sings have been eradicated from the
251.       * String
252.       */
253.      try {
254.        value = Double.parseDouble(valueString);
255.      } catch (NumberFormatException e) {
256.        throw new IllegalArgumentException(
257.            "The course value >" + columns[0] + "< cannot be parsed");
258.      } catch (Exception e) {
259.        throw e;
260.      }
261.
262.      return value;
263.    }
264.  }
```

Komponente CsvFormat

Listing 24: Komponente CsvFormat

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  /**
4.   * This enum provides the most common ways a CSV can be structured,
5.   * including the symbols for field, date, time, and thousands separator, as
6.   * well as the decimal point and the order of values (year, month, day)
7.   * inside the date value.
8.   *
9.   * @author Max Rumford
10.  *
11.  */
12. public enum CsvFormat {
13.   EU(";", ".", ":", ",", ".", DateOrder.DAY_MONTH_YEAR),
14.   EU_YEAR_MONTH_DAY(";", ".", ":", ",", ".", DateOrder.YEAR_MONTH_DAY),
15.   US(",", "/", ":", ".", ",", DateOrder.MONTH_DAY_YEAR),
16.   US_YEAR_MONTH_DAY(",", "/", ":", ".", ",", DateOrder.YEAR_MONTH_DAY);
17.
18.   private final String fieldSeparator;
19.   private final String dateSeparator;
20.   private final String timeSeparator;
21.   private final String decimalPoint;
22.   private final String thousandsSeparator;
23.   private final DateOrder dateOrder;
24.
25.   CsvFormat(String fieldSeparator, String dateSeparator,
26.       String timeSeparator, String decimalPoint, String thousandSeparator,
27.       DateOrder dateOrder) {
28.     this.fieldSeparator = fieldSeparator;
29.     this.dateSeparator = dateSeparator;
30.     this.timeSeparator = timeSeparator;
31.     this.decimalPoint = decimalPoint;
32.     this.thousandsSeparator = thousandSeparator;
33.     this.dateOrder = dateOrder;
34.   }
35.
36.   /**
37.    * ========================================================================
38.    * GETTERS AND SETTERS
39.    * ========================================================================
40.    */
41.   /**
42.    * @return fieldSeparator CsvFormat
43.    */
44.   public String getFieldSeparator() {
45.     return fieldSeparator;
46.   }
47.
48.   /**
49.    * @return dateSeparator CsvFormat
```

```
50.       */
51.      public String getDateSeparator() {
52.        return dateSeparator;
53.      }
54.
55.      /**
56.       * @return timeSeparator CsvFormat
57.       */
58.      public String getTimeSeparator() {
59.        return timeSeparator;
60.      }
61.
62.      /**
63.       * @return decimalPoint CsvFormat
64.       */
65.      public String getDecimalPoint() {
66.        return decimalPoint;
67.      }
68.
69.      /**
70.       * @return thousandSeparator CsvFormat
71.       */
72.      public String getThousandsSeparator() {
73.        return thousandsSeparator;
74.      }
75.
76.      /**
77.       * @return dateOrder CsvFormat
78.       */
79.      public DateOrder getMonthDayOrder() {
80.        return dateOrder;
81.      }
82.    }
```

Komponente DateOrder

Listing 25: Komponente DateOrder

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  /**
4.   * This enum represents three variations of how date fields can be ordered.
5.   *
6.   * @author Max Rumford
7.   *
8.   */
9.  public enum DateOrder {
10.   YEAR_MONTH_DAY, DAY_MONTH_YEAR, MONTH_DAY_YEAR;
11. }
```

Komponente ExampleClient

Listing 26: Komponente ExampleClient

```
1.  package de.rumford.tradingsystem;
2.
3.  import java.io.File;
4.  import java.io.IOException;
5.  import java.nio.file.Path;
6.  import java.text.NumberFormat;
7.  import java.time.Duration;
8.  import java.time.LocalDateTime;
9.  import java.util.Locale;
10.
11. import org.apache.log4j.Logger;
12.
13. import de.rumford.tradingsystem.helper.*;
14.
15. /**
16.  * The ExampleClient is an example of how to use this library.
17.  *
18.  * @author Max Rumford
19.  *
20.  */
21. @JacocoIgnoreGenerated
22. public class ExampleClient {
23.   static final double CAPITAL = 100000;
24.
25.   static String workingDir = Path.of("src", "test", "resources")
26.       .toString();
27.
28.   static final String DAX = "DAX";
29.   static String daxFileName = "DAX.csv";
30.   static String daxShortFileName = "DAX_short.csv";
31.   static String daxVolatilityFileName = "DAX_VDAX.csv";
32.
33.   static final String STOXX = "EURO STOXX 50";
34.   static String stoxxFileName = "STOXX.csv";
35.   static String stoxxShortFileName = "STOXX-Short.csv";
36.   static String stoxxVolatilityFileName = "STOXX-VSTOXX.csv";
37.
38.   static final String SP500 = "S&P 500";
39.   static String sp500FileName = "S&P.csv";
40.   static String sp500VolatilityFileName = "S&P_VIX.csv";
41.
42.   static final LocalDateTime START_OF_REFERENCE_WINDOW = LocalDateTime
43.       .of(2014, 1, 2, 22, 0);
44.   static final LocalDateTime END_OF_REFERENCE_WINDOW = LocalDateTime
45.       .of(2018, 12, 28, 22, 0);
46.   static final double BASE_SCALE = 10;
47.
48.   static final int LOOKBACK_WINDOW_8 = 8;
49.
```

```
50.     static final LocalDateTime START_OF_TEST_WINDOW = LocalDateTime.of(2019,
51.         1, 2, 22, 0);
52.     static final LocalDateTime END_OF_TEST_WINDOW = LocalDateTime.of(2019,
53.         12, 30, 22, 0);
54.
55.     private static final Logger logger = Logger
56.         .getLogger(ExampleClient.class);
57.
58.     private ExampleClient() {
59.     }
60.
61.     /**
62.      * Main method depicting how to all components of this system.
63.      *
64.      * @param args Arguments. Empty.
65.      * @throws IOException if the given filenames cannot be found.
66.      */
67.     public static void main(String[] args) throws IOException {
68.       String basePath = null;
69.       if (args.length != 0) {
70.         basePath = args[0];
71.       }
72.       setPathNames(basePath);
73.
74.       LocalDateTime startingTime = LocalDateTime.now();
75.
76.       if (args.length > 1) {
77.         switch (args[1]) {
78.         case "1":
79.           exampleForOneBaseValue();
80.           break;
81.         case "3":
82.           exampleForThreeBaseValues();
83.           break;
84.         default:
85.           logger.info(String.format(
86.               "No fitting routing found for argument %s. Running default.",
87.               args[1]));
88.           exampleForThreeBaseValues();
89.           break;
90.         }
91.       } else {
92.         logger.info("Running default.");
93.         exampleForThreeBaseValues();
94.       }
95.
96.       logDuration(startingTime);
97.     }
98.
99.     private static void setPathNames(String basePath) throws IOException {
100.      if (basePath != null) {
101.        File f = Path.of(basePath).toFile();
102.        if (!f.exists()) {
103.          logger.warn(String.format(
```

```
104.                "Given path %s does not exist. Using default path %s.",
105.                f.getCanonicalPath(),
106.                Path.of(workingDir).toFile().getCanonicalPath()));
107.          } else {
108.            workingDir = basePath;
109.          }
110.        }
111.        daxFileName = getPathAsStringByFileName(daxFileName);
112.        daxShortFileName = getPathAsStringByFileName(daxShortFileName);
113.        daxVolatilityFileName = getPathAsStringByFileName(
114.            daxVolatilityFileName);
115.
116.        stoxxFileName = getPathAsStringByFileName(stoxxFileName);
117.        stoxxShortFileName = getPathAsStringByFileName(stoxxShortFileName);
118.        stoxxVolatilityFileName = getPathAsStringByFileName(
119.            stoxxVolatilityFileName);
120.
121.        sp500FileName = getPathAsStringByFileName(sp500FileName);
122.        sp500VolatilityFileName = getPathAsStringByFileName(
123.            sp500VolatilityFileName);
124.      }
125.
126.      private static String getPathAsStringByFileName(String fileName) {
127.        return Path.of(workingDir, fileName).toString();
128.      }
129.
130.      /**
131.       * Performs all necessary constructions for one base values.
132.       *
133.       * @throws IOException if any file handling goes wrong.
134.       */
135.      private static void exampleForOneBaseValue() throws IOException {
136.
137.        forOneBaseValueWithShort(STOXX, stoxxFileName, stoxxShortFileName,
138.            stoxxVolatilityFileName, CAPITAL);
139.
140.      }
141.
142.      /**
143.       * Performs all necessary constructions for three base values.
144.       *
145.       * @throws IOException if any file handling goes wrong.
146.       */
147.      private static void exampleForThreeBaseValues() throws IOException {
148.        double daxPerf = forOneBaseValueWithShort(DAX, daxFileName,
149.            daxShortFileName, daxVolatilityFileName, CAPITAL * 0.2002);
150.
151.        double stoxxPerf = forOneBaseValueWithShort(STOXX, stoxxFileName,
152.            stoxxShortFileName, stoxxVolatilityFileName, CAPITAL * 0.407);
153.
154.        double spPerf = forOneBaseValue(SP500, sp500FileName,
155.            sp500VolatilityFileName, CAPITAL * 0.3928);
156.
157.        formatPerformance(CAPITAL, daxPerf + stoxxPerf + spPerf);
```

```
158.     }
159.
160.     /**
161.      * Perform all calculations for one base value. No Short index values
162.      * given.
163.      *
164.      * @param baseValueName      Name of the base value.
165.      * @param longFileName       File name with long index values.
166.      * @param volatilityFileName File name with volatility index values.
167.      * @param capital            Capital to be spent on this base value.
168.      * @return The backtested performance value
169.      * @throws IOException if the filenames cause issues in file handling.
170.      */
171.     private static double forOneBaseValue(String baseValueName,
172.         String longFileName, String volatilityFileName, double capital)
173.         throws IOException {
174.
175.       ValueDateTupel[] baseValues = DataSource.getDataFromCsv(longFileName,
176.           CsvFormat.EU);
177.       ValueDateTupel[] volatilityIndexValues = DataSource
178.           .getDataFromCsv(volatilityFileName, CsvFormat.EU);
179.
180.       ValueDateTupel[][] aligned = ValueDateTupel.alignDates(
181.           new ValueDateTupel[][] { baseValues, volatilityIndexValues });
182.
183.       baseValues = aligned[0];
184.       volatilityIndexValues = aligned[1];
185.
186.       BaseValue baseValue = new BaseValue(baseValueName, baseValues);
187.       // logger.info("BaseValue " + baseValue.getName() + " created.");
188.
189.       return instantiateAndBacktest(capital, volatilityIndexValues,
190.           baseValue);
191.     }
192.
193.     /**
194.      * Perform all calculations for one base value. Short index values given.
195.      *
196.      * @param baseValueName      Name of the base value.
197.      * @param longFileName       File name with long index values.
198.      * @param shortFileName      File name with short index values.
199.      * @param volatilityFileName File name with volatility index values.
200.      * @param capital            Capital to be spent on this base value.
201.      * @return The backtested performance value
202.      * @throws IOException if the filenames cause issues in file handling.
203.      */
204.     private static double forOneBaseValueWithShort(String baseValueName,
205.         String longFileName, String shortFileName, String volatilityFileName,
206.         double capital) throws IOException {
207.
208.       ValueDateTupel[] baseValues = DataSource.getDataFromCsv(longFileName,
209.           CsvFormat.EU);
210.       ValueDateTupel[] shortIndexValues = DataSource
211.           .getDataFromCsv(shortFileName, CsvFormat.EU);
```

```
212.      ValueDateTupel[] volatilityIndexValues = DataSource
213.          .getDataFromCsv(volatilityFileName, CsvFormat.EU);
214.
215.      ValueDateTupel[][] aligned = ValueDateTupel
216.          .alignDates(new ValueDateTupel[][] { baseValues, shortIndexValues,
217.              volatilityIndexValues });
218.
219.      baseValues = aligned[0];
220.      shortIndexValues = aligned[1];
221.      volatilityIndexValues = aligned[2];
222.
223.      BaseValue baseValue = new BaseValue(baseValueName, baseValues,
224.          shortIndexValues);
225.      // logger.info("BaseValue " + baseValue.getName() + " created.");
226.
227.      return instantiateAndBacktest(capital, volatilityIndexValues,
228.          baseValue);
229.    }
230.
231.    /**
232.     * Sets up the rules and runs a backtest for the set testing window.
233.     *
234.     * @param capital               The capital to be invested upon the given
235.     *                              base value.
236.     * @param volatilityIndexValues The volatility index values to be used in
237.     *                              the volatility difference rule.
238.     * @param baseValue             The base value to be evaluated.
239.     * @return The result as of
240.     *         {@link SubSystem#backtest(LocalDateTime, LocalDateTime)}
241.     */
242.    private static double instantiateAndBacktest(double capital,
243.        ValueDateTupel[] volatilityIndexValues, BaseValue baseValue) {
244.
245.      /* Create the rules for the given base value. */
246.      Rule[] rules = createRules(volatilityIndexValues, baseValue);
247.
248.      /* Create the subsystem. */
249.      SubSystem subSystem = new SubSystem(baseValue, rules, capital,
250.          BASE_SCALE);
251.
252.      /*
253.       * Perform the backtest. Gets the available capital after the last
254.       * trading period.
255.       */
256.      double performanceValue = performBacktest(subSystem);
257.
258.      /* Formats and logs the realized returns. */
259.      formatPerformance(capital, performanceValue);
260.
261.      /*
262.       * Extracts the last forecast given in the subsystem. Also depicts the
263.       * position held after end of test window.
264.       */
265.      double lastForecast = subSystem
```

```
266.            .getCombinedForecasts()[subSystem.getCombinedForecasts().length
267.                - 1].getValue();
268.        logger.info(
269.            "Current position: " + Util.getPositionFromForecast(lastForecast)
270.                + ", " + lastForecast);
271.
272.        return performanceValue;
273.      }
274.
275.      /**
276.       * Performs the backtest for for the given subsystem.
277.       *
278.       * @param subSystem The subsystem to be tested.
279.       * @return The result as of
280.       *         {@link SubSystem#backtest(LocalDateTime, LocalDateTime)}
281.       */
282.      private static double performBacktest(SubSystem subSystem) {
283.        logger.info("Starting backtest for BaseValue "
284.            + subSystem.getBaseValue().getName() + " with testing window "
285.            + START_OF_TEST_WINDOW + " - " + END_OF_TEST_WINDOW);
286.        return subSystem.backtest(START_OF_TEST_WINDOW, END_OF_TEST_WINDOW);
287.      }
288.
289.      /**
290.       * Pretty print the given performance value in realtion to the given
291.       * starting capital.
292.       *
293.       * @param capital          The starting capital
294.       * @param performanceValue The performance achieved.
295.       */
296.      private static void formatPerformance(double capital,
297.          double performanceValue) {
298.        /* Calculate the returns percentage achieved. */
299.        double performancePercentage = Util.calculateReturn(capital,
300.            performanceValue) * 100;
301.
302.        /* Pretty print given capital and performance value. */
303.        NumberFormat moneyFormatter = NumberFormat
304.            .getCurrencyInstance(Locale.GERMANY);
305.        String capitalString = moneyFormatter.format(capital);
306.        String performanceValueString = moneyFormatter
307.            .format(performanceValue);
308.
309.        /* Pretty print the returns percentage */
310.        NumberFormat decimalFormatter = NumberFormat
311.            .getNumberInstance(Locale.GERMANY);
312.        String performancePercentageString = decimalFormatter
313.            .format(performancePercentage);
314.
315.        logger.info(
316.            "Done Testing. Value after backtest: " + performanceValueString);
317.        logger.info("With your starting capital of " + capitalString
318.            + " that's a net return of " + performancePercentageString + "%.");
319.      }
```

```
320.
321.    /**
322.     * Creates the rules for the given base value.
323.     *
324.     * @param volatilityIndexValues volatility index values to be used in the
325.     *                              {@link VolatilityDifference}.
326.     * @param baseValue             the base value to be used in the rules.
327.     * @return An array of top level rules.
328.     */
329.    private static Rule[] createRules(ValueDateTupel[] volatilityIndexValues,
330.        BaseValue baseValue) {
331.      VolatilityDifference volDif = createOneVolatilityDifference(
332.          volatilityIndexValues, baseValue);
333.
334.      EWMAC ewmacTop = createEwmacs(baseValue);
335.
336.      return new Rule[] { volDif, ewmacTop };
337.    }
338.
339.    /**
340.     * Creates all {@link EWMAC}s for this example.
341.     *
342.     * @param baseValue the {@link BaseValue} to be used in the
343.     *                  {@link EWMAC}s.
344.     * @return The top level {@link EWMAC}.
345.     */
346.    private static EWMAC createEwmacs(BaseValue baseValue) {
347.      EWMAC ewmacShort = createOneEwmac(baseValue, null, 8, 2);
348.
349.      EWMAC ewmacMiddle = createOneEwmac(baseValue, null, 16, 4);
350.
351.      EWMAC ewmacLong = createOneEwmac(baseValue, null, 32, 8);
352.
353.      EWMAC[] ewmacVariations = { //
354.          ewmacShort, //
355.          ewmacMiddle, //
356.          ewmacLong };
357.
358.      return createOneEwmac(baseValue, ewmacVariations, 0, 0);
359.    }
360.
361.    /**
362.     * Creates one {@link EWMAC} using the given parameters.
363.     *
364.     * @param baseValue    the {@link BaseValue} to be used in the
365.     *                     {@link EWMAC}.
366.     * @param variations   The array of {@link EWMAC} to be given to the
367.     *                     {@link EWMAC} as variations.
368.     * @param longHorizon  The value for instantiation of the long horizon
369.     *                     {@link EWMA}.
370.     * @param shortHorizon The value for instantiation of the short horizon
371.     *                     {@link EWMA}.
372.     * @return The created {@link EWMAC}.
373.     */
```

```
374.     private static EWMAC createOneEwmac(BaseValue baseValue,
375.         EWMAC[] variations, int longHorizon, int shortHorizon) {
376.       return new EWMAC(baseValue, variations, START_OF_REFERENCE_WINDOW,
377.           END_OF_REFERENCE_WINDOW, longHorizon, shortHorizon, BASE_SCALE);
378.     }
379.
380.     /**
381.      * Creates one {@link VolatilityDifference} for the given parameters.
382.      *
383.      * @param volatilityIndexValues The volatility index values to be used in
384.      *                              the new {@link VolatilityDifference}
385.      * @param baseValue             The {@link BaseValue} ot be used for the
386.      *                              new {@link VolatilityDifference}.
387.      * @return The new {@link VolatilityDifference}.
388.      */
389.     private static VolatilityDifference createOneVolatilityDifference(
390.         ValueDateTupel[] volatilityIndexValues, BaseValue baseValue) {
391.
392.       return new VolatilityDifference(baseValue, null,
393.           START_OF_REFERENCE_WINDOW, END_OF_REFERENCE_WINDOW,
394.           LOOKBACK_WINDOW_8, BASE_SCALE, volatilityIndexValues);
395.     }
396.
397.     /**
398.      * Log the time elapsed between the given starting time and
399.      * {@link LocalDateTime#now()}
400.      *
401.      * @param startingTime The {@link LocalDateTime} to be used for
402.      *                     comparison.
403.      */
404.     private static void logDuration(LocalDateTime startingTime) {
405.       Duration duration = Duration.between(startingTime,
406.           LocalDateTime.now());
407.       logger.info("Runtime: " + duration.toMillis() / 1000d + " seconds.");
408.     }
409.   }
```

A4 Testcode

Komponente RuleTest

Listing 27: Komponente RuleTest

```
1.    package de.rumford.tradingsystem;
2.
3.    import static org.junit.jupiter.api.Assertions.assertArrayEquals;
4.    import static org.junit.jupiter.api.Assertions.assertEquals;
5.    import static org.junit.jupiter.api.Assertions.assertThrows;
6.
7.    import java.time.LocalDate;
8.    import java.time.LocalDateTime;
9.    import java.time.LocalTime;
10.
11.   import org.junit.jupiter.api.BeforeAll;
12.   import org.junit.jupiter.api.BeforeEach;
13.   import org.junit.jupiter.api.Test;
14.
15.   import de.rumford.tradingsystem.helper.BaseValueFactory;
16.   import de.rumford.tradingsystem.helper.GeneratedCode;
17.   import de.rumford.tradingsystem.helper.ValueDateTupel;
18.
19.   /**
20.    * Test class for {@link Rule}.
21.    *
22.    * @author Max Rumford
23.    *
24.    */
25.   public class RuleTest {
26.
27.     public static class RealRule extends Rule {
28.       private double variator;
29.
30.       public RealRule(BaseValue baseValue, Rule[] variations,
31.           LocalDateTime startOfReferenceWindow,
32.           LocalDateTime endOfReferenceWindow, double baseScale,
33.           double variator) {
34.         super(baseValue, variations, startOfReferenceWindow,
35.             endOfReferenceWindow, baseScale);
36.         this.variator = variator;
37.       }
38.
39.       public static RealRule from(BaseValue baseValue, Rule[] variations,
40.           LocalDateTime startOfReferenceWindow,
41.           LocalDateTime endOfReferenceWindow, double baseScale,
42.           double variator) {
43.         return new RealRule(baseValue, variations, startOfReferenceWindow,
44.             endOfReferenceWindow, baseScale, variator);
45.       }
46.
47.       @Override
```

```
48.         double calculateRawForecast(LocalDateTime forecastDateTime) {
49.           return ValueDateTupel
50.               .getElement(this.getBaseValue().getValues(), forecastDateTime)
51.               .getValue() + this.variator * 100;
52.         }
53.
54.         @GeneratedCode
55.         @Override
56.         public int hashCode() {
57.           final int prime = 31;
58.           int result = super.hashCode();
59.           long temp;
60.           temp = Double.doubleToLongBits(variator);
61.           result = prime * result + (int) (temp ^ (temp >>> 32));
62.           return result;
63.         }
64.
65.         @GeneratedCode
66.         @Override
67.         public boolean equals(Object obj) {
68.
69.           if (this == obj)
70.             return true;
71.
72.           if (!super.equals(obj))
73.             return false;
74.
75.           if (getClass() != obj.getClass())
76.             return false;
77.           RealRule other = (RealRule) obj;
78.           if (Double.doubleToLongBits(variator) != Double
79.               .doubleToLongBits(other.variator))
80.             return false;
81.           return true;
82.         }
83.
84.         @GeneratedCode
85.         @Override
86.         public String toString() {
87.           return "RealRule [variator=" + variator + "]";
88.         }
89.
90.       }
91.
92.       static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
93.           "Incorrect Exception message";
94.
95.       RealRule realRule;
96.       static BaseValue baseValue;
97.       static double variator;
98.
99.       static final String BASE_VALUE_NAME = "Base value name";
100.      static final int BASE_SCALE = 10;
101.
```

```
102.    static LocalDateTime localDateTime2019Dec31220000;
103.    static LocalDateTime localDateTimeJan01220000;
104.    static LocalDateTime localDateTimeJan02220000;
105.    static LocalDateTime localDateTimeJan03220000;
106.    static LocalDateTime localDateTimeJan04220000;
107.    static LocalDateTime localDateTimeJan05220000;
108.    static LocalDateTime localDateTimeJan07220000;
109.    static LocalDateTime localDateTimeJan10220000;
110.    static LocalDateTime localDateTimeJan12220000;
111.    static LocalDateTime localDateTimeFeb05220000;
112.    static LocalDateTime localDateTime2020Dec31220000;
113.
114.    @BeforeAll
115.    static void setUpBeforeClass() {
116.      baseValue = BaseValueFactory.jan1Jan31calcShort(BASE_VALUE_NAME);
117.
118.      localDateTime2019Dec31220000 = LocalDateTime
119.          .of(LocalDate.of(2019, 12, 31), LocalTime.of(22, 0));
120.      localDateTimeJan01220000 = LocalDateTime.of(LocalDate.of(2020, 1, 1),
121.          LocalTime.of(22, 0));
122.      localDateTimeJan02220000 = LocalDateTime.of(LocalDate.of(2020, 1, 2),
123.          LocalTime.of(22, 0));
124.      localDateTimeJan03220000 = LocalDateTime.of(LocalDate.of(2020, 1, 3),
125.          LocalTime.of(22, 0));
126.      localDateTimeJan04220000 = LocalDateTime.of(LocalDate.of(2020, 1, 4),
127.          LocalTime.of(22, 0));
128.      localDateTimeJan05220000 = LocalDateTime.of(LocalDate.of(2020, 1, 5),
129.          LocalTime.of(22, 0));
130.      localDateTimeJan07220000 = LocalDateTime.of(LocalDate.of(2020, 1, 7),
131.          LocalTime.of(22, 0));
132.      localDateTimeJan10220000 = LocalDateTime.of(LocalDate.of(2020, 1, 10),
133.          LocalTime.of(22, 0));
134.      localDateTimeJan12220000 = LocalDateTime.of(LocalDate.of(2020, 1, 12),
135.          LocalTime.of(22, 0));
136.      localDateTimeFeb05220000 = LocalDateTime.of(LocalDate.of(2020, 2, 5),
137.          LocalTime.of(22, 0));
138.      localDateTime2020Dec31220000 = LocalDateTime
139.          .of(LocalDate.of(2020, 12, 31), LocalTime.of(22, 0));
140.    }
141.
142.    @BeforeEach
143.    void setUp() {
144.      variator = 1;
145.      realRule = RealRule.from(baseValue, null, localDateTimeJan10220000,
146.          localDateTimeJan12220000, BASE_SCALE, variator);
147.    }
148.
149.    /**
150.     * Test method for {@link Rule#calculateForecastScalar()}.
151.     */
152.    @Test
153.    void testCalculateForecastScalar() {
154.      double expectedValue = 2.793618728459556; // Excel: 2.79361872845956
155.
```

```
156.      double actualValue = realRule.getForecastScalar();
157.
158.      assertEquals(expectedValue, actualValue,
159.          "Forecast scalar is not correctly calculated");
160.    }
161.
162.    /**
163.     * Test method for {@link Rule#calculateForecastScalar()}.
164.     */
165.    @Test
166.    void testCalculateForecastScalar_FcScalarDiv0() {
167.      String expectedMessage = "Illegal values in calulated forecast values."
168.          + " Adjust reference window.";
169.
170.      double variator = -1d;
171.      realRule = RealRule.from(
172.          BaseValueFactory.jan1Jan5calcShort_sameValuesOn2To5(
173.              BASE_VALUE_NAME),
174.          null, localDateTimeJan02220000, localDateTimeJan05220000,
175.          BASE_SCALE, variator);
176.
177.      Exception thrown = assertThrows(IllegalArgumentException.class,
178.          () -> realRule.getForecastScalar(),
179.          "Illegal values in reference window are not properly handled");
180.
181.      assertEquals(expectedMessage, thrown.getMessage(),
182.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
183.    }
184.
185.    /**
186.     * Test method for {@link Rule#calculateScaledForecasts()}.
187.     */
188.    @Test
189.    void testCalculateForecasts() {
190.      double expectedValue = 12.66459427439483; // Excel: 12.6645942743948
191.
192.      double actualValue = ValueDateTupel
193.          .getElement(realRule.getForecasts(), localDateTimeJan10220000)
194.          .getValue();
195.
196.      assertEquals(expectedValue, actualValue,
197.          "Forecasts are not correctly calculated");
198.    }
199.
200.    /**
201.     * Test method for {@link Rule#calculateScaledForecasts()}.
202.     */
203.    @Test
204.    void testCalculateForecasts_unchangedOverTime() {
205.      double expectedValue = 12.66459427439483; // Excel: 12.6645942743948
206.
207.      ValueDateTupel
208.          .getElement(realRule.getForecasts(), localDateTimeJan10220000)
209.          .getValue();
```

```
210.      double actualValue = ValueDateTupel
211.          .getElement(realRule.getForecasts(), localDateTimeJan10220000)
212.          .getValue();
213.
214.      assertEquals(expectedValue, actualValue,
215.          "Forecasts are not correctly calculated");
216.    }
217.
218.    /**
219.     * Test method for {@link Rule#calculateScaledForecast(double)}.
220.     */
221.    @Test
222.    void testCalculateScaledForecast_FcNegative20() {
223.      double expectedValue = -20d;
224.      variator = -0.1d;
225.      realRule = RealRule.from(
226.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
227.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
228.          variator);
229.
230.      ValueDateTupel
231.          .getElement(realRule.getForecasts(), localDateTimeFeb05220000)
232.          .getValue();
233.      double actualValue = ValueDateTupel
234.          .getElement(realRule.getForecasts(), localDateTimeFeb05220000)
235.          .getValue();
236.
237.      assertEquals(expectedValue, actualValue,
238.          "Forecasts < -20 are not correctly calculated");
239.    }
240.
241.    /**
242.     * Test method for
243.     * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
244.     * LocalDateTime, double)}.
245.     */
246.    @Test
247.    void testCalculateVolatilityIndices_baseValue_null() {
248.      BaseValue nullBaseValue = null;
249.      String expectedMessage = "Base value must not be null";
250.
251.      Exception thrown = assertThrows(IllegalArgumentException.class,
252.          () -> RealRule.from(nullBaseValue, null, localDateTimeJan10220000,
253.              localDateTimeJan12220000, BASE_SCALE, variator),
254.          "Base value of null is not correctly handled");
255.
256.      assertEquals(expectedMessage, thrown.getMessage(),
257.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
258.    }
259.
260.    /**
261.     * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
262.     * LocalDateTime, double}.
263.     */
```

```
264.     @Test
265.     void testValidateInputs_startOfReferenceWindow_null() {
266.       String expectedMessage = "The given reference window does not meet "
267.           + "specifications.";
268.       String expectedCause = "Start of time window value must not be null";
269.
270.       Exception thrown = assertThrows(IllegalArgumentException.class,
271.           () -> RealRule.from(baseValue, null, null,
272.               localDateTimeJan12220000, BASE_SCALE, variator),
273.           "startOfReferenceWindow of null is not correctly handled");
274.
275.       assertEquals(expectedMessage, thrown.getMessage(),
276.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
277.       assertEquals(expectedCause, thrown.getCause().getMessage(),
278.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
279.     }
280.
281.     /**
282.      * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
283.      * LocalDateTime, double}.
284.      */
285.     @Test
286.     void testValidateInputs_endOfReferenceWindow_null() {
287.       String expectedMessage = "The given reference window does not meet "
288.           + "specifications.";
289.       String expectedCause = "End of time window value must not be null";
290.
291.       Exception thrown = assertThrows(IllegalArgumentException.class,
292.           () -> RealRule.from(baseValue, null, localDateTimeJan10220000,
293.               null, BASE_SCALE, variator),
294.           "endOfReferenceWindow of null is not correctly handled");
295.
296.       assertEquals(expectedMessage, thrown.getMessage(),
297.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
298.       assertEquals(expectedCause, thrown.getCause().getMessage(),
299.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
300.     }
301.
302.     /**
303.      * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
304.      * LocalDateTime, double}.
305.      */
306.     @Test
307.     void testValidateInputs_baseScale_0() {
308.       String expectedMessage = "The given base scale does not meet "
309.           + "specifications.";
310.       String expectedCause = "Value must be a positive decimal";
311.       double zeroBaseScale = 0;
312.
313.       Exception thrown = assertThrows(IllegalArgumentException.class,
314.           () -> RealRule.from(baseValue, null, localDateTimeJan10220000,
315.               localDateTimeJan12220000, zeroBaseScale, variator),
316.           "baseScale of zero is not correctly handled");
317.
```

```
318.      assertEquals(expectedMessage, thrown.getMessage(),
319.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
320.      assertEquals(expectedCause, thrown.getCause().getMessage(),
321.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
322.    }
323.
324.    /**
325.     * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
326.     * LocalDateTime, double}.
327.     */
328.    @Test
329.    void testValidateInputs_baseScale_sub0() {
330.      String expectedMessage = "The given base scale does not meet "
331.          + "specifications.";
332.      String expectedCause = "Value must be a positive decimal";
333.      double subZeroBaseScale = -1;
334.
335.      Exception thrown = assertThrows(IllegalArgumentException.class,
336.          () -> RealRule.from(baseValue, null, localDateTimeJan10220000,
337.              localDateTimeJan12220000, subZeroBaseScale, variator),
338.          "baseScale of less than zero is not correctly handled");
339.
340.      assertEquals(expectedMessage, thrown.getMessage(),
341.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
342.      assertEquals(expectedCause, thrown.getCause().getMessage(),
343.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
344.    }
345.
346.    /**
347.     * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
348.     * LocalDateTime, double}.
349.     */
350.    @Test
351.    void testValidateInputs_endOfRefWindow_before_startOfRefWindow() {
352.      String expectedMessage = "The given reference window does not meet "
353.          + "specifications.";
354.      String expectedCause = "End of time window value must be after start "
355.          + "of time window value";
356.
357.      Exception thrown = assertThrows(IllegalArgumentException.class,
358.          () -> RealRule.from(baseValue, null, localDateTimeJan12220000,
359.              localDateTimeJan10220000, BASE_SCALE, variator),
360.          "endOfReferenceWindow before startOfReferenceWindow is not "
361.          + "correctly handled");
362.
363.      assertEquals(expectedMessage, thrown.getMessage(),
364.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
365.      assertEquals(expectedCause, thrown.getCause().getMessage(),
366.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
367.    }
368.
369.    /**
370.     * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
371.     * LocalDateTime, double}.
```

```
372.      */
373.     @Test
374.     void testValidateInputs_illegalStartOfReferenceWindow() {
375.       String expectedMessage = "Given base value and reference window do "
376.           + "not fit.";
377.       String expectedCause = "Given values do not include given start value "
378.           + "for time window";
379.
380.       Exception thrown = assertThrows(IllegalArgumentException.class,
381.           () -> RealRule.from(baseValue, null, localDateTime2019Dec31220000,
382.               localDateTimeJan12220000, BASE_SCALE, variator),
383.           "Not included startOfReferenceWindow is not correctly handled");
384.
385.       assertEquals(expectedMessage, thrown.getMessage(),
386.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
387.       assertEquals(expectedCause, thrown.getCause().getMessage(),
388.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
389.     }
390.
391.     /**
392.      * Test method for {@link Rule#validateInputs(BaseValue, LocalDateTime,
393.      * LocalDateTime, double}.
394.      */
395.     @Test
396.     void testValidateInputs_illegalEndOfReferenceWindow() {
397.       String expectedMessage = "Given base value and reference window do not"
398.           + " fit.";
399.       String expectedCause = "Given values do not include given end value "
400.           + "for time window";
401.
402.       Exception thrown = assertThrows(IllegalArgumentException.class,
403.           () -> RealRule.from(baseValue, null, localDateTimeJan10220000,
404.               localDateTime2020Dec31220000, BASE_SCALE, variator),
405.           "Not included endOfReferenceWindow is not correctly handled");
406.
407.       assertEquals(expectedMessage, thrown.getMessage(),
408.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
409.       assertEquals(expectedCause, thrown.getCause().getMessage(),
410.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
411.     }
412.
413.     /**
414.      * Test method for
415.      * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
416.      * LocalDateTime, double)}.
417.      */
418.     @Test
419.     void testValidateInputs_moreThan3Variations() {
420.       String expectedMessage = "A rule must not contain more than 3 "
421.           + "variations.";
422.       RealRule var1 = RealRule.from(
423.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
424.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
425.           variator);
```

```
426.       RealRule var2 = RealRule.from(
427.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
428.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
429.           variator);
430.       RealRule var3 = RealRule.from(
431.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
432.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
433.           variator);
434.       RealRule var4 = RealRule.from(
435.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
436.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
437.           variator);
438.       RealRule[] variations = { var1, var2, var3, var4 };
439.
440.       Exception thrown = assertThrows(IllegalArgumentException.class,
441.           () -> RealRule.from(
442.               BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME),
443.               variations, localDateTimeJan10220000, localDateTimeJan12220000,
444.               BASE_SCALE, variator),
445.           "> 3 variations is not properly handled");
446.
447.       assertEquals(expectedMessage, thrown.getMessage(),
448.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
449.     }
450.
451.     /**
452.      * Test method for
453.      * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
454.      * LocalDateTime, double)}.
455.      */
456.     @Test
457.     void testValidateInputs_emptyVariationsArray() {
458.       String expectedMessage = "The given variations array must not be "
459.           + "empty.";
460.       RealRule[] variations = {};
461.
462.       Exception thrown = assertThrows(IllegalArgumentException.class,
463.           () -> RealRule.from(
464.               BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME),
465.               variations, localDateTimeJan10220000, localDateTimeJan12220000,
466.               BASE_SCALE, variator),
467.           "Empty variations array is not properly handled");
468.
469.       assertEquals(expectedMessage, thrown.getMessage(),
470.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
471.     }
472.
473.     /**
474.      * Test method for
475.      * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
476.      * LocalDateTime, double)}.
477.      */
478.     @Test
479.     void testValidateInputs_variationIsNull() {
```

```
480.      String expectedMessage = "The variation at position 2 in the given "
481.          + "variations array is null.";
482.      RealRule var1 = RealRule.from(
483.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
484.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
485.          variator);
486.      RealRule var2 = RealRule.from(
487.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
488.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
489.          variator);
490.      RealRule var3 = null;
491.      RealRule[] variations = { var1, var2, var3 };
492.
493.      Exception thrown = assertThrows(IllegalArgumentException.class,
494.          () -> RealRule.from(
495.              BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME),
496.              variations, localDateTimeJan10220000, localDateTimeJan12220000,
497.              BASE_SCALE, variator),
498.          "Variation = null is not properly handled");
499.
500.      assertEquals(expectedMessage, thrown.getMessage(),
501.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
502.    }
503.
504.    /**
505.     * Test method for
506.     * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
507.     * LocalDateTime, double)}.
508.     */
509.    @Test
510.    void testValidateInputs_variationsStartOfRefWindowDoesNotMatchRules() {
511.      String expectedMessage = "The given reference window does not match "
512.          + "the variation's at position 1. The given start of reference "
513.          + "window is different.";
514.      RealRule var1 = RealRule.from(
515.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
516.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
517.          variator);
518.      RealRule var2 = RealRule.from(
519.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
520.          localDateTimeJan07220000, localDateTimeJan12220000, BASE_SCALE,
521.          variator);
522.      RealRule var3 = RealRule.from(
523.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
524.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
525.          variator);
526.      RealRule[] variations = { var1, var2, var3 };
527.
528.      Exception thrown = assertThrows(IllegalArgumentException.class,
529.          () -> RealRule.from(
530.              BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME),
531.              variations, localDateTimeJan10220000, localDateTimeJan12220000,
532.              BASE_SCALE, variator),
533.          "Unmatched start of reference window is not properly handled");
```

```
534.
535.       assertEquals(expectedMessage, thrown.getMessage(),
536.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
537.     }
538.
539.     /**
540.      * Test method for
541.      * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
542.      * LocalDateTime, double)}.
543.      */
544.     @Test
545.     void testValidateInputs_variationsEndOfRefWindowDoesNotMatchRules() {
546.       String expectedMessage = "The given reference window does not match "
547.           + "the variation's at position 1. The given end of reference "
548.           + "window is different.";
549.       RealRule var1 = RealRule.from(
550.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
551.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
552.           variator);
553.       RealRule var2 = RealRule.from(
554.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
555.           localDateTimeJan10220000, localDateTimeFeb05220000, BASE_SCALE,
556.           variator);
557.       RealRule var3 = RealRule.from(
558.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
559.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
560.           variator);
561.       RealRule[] variations = { var1, var2, var3 };
562.
563.       Exception thrown = assertThrows(IllegalArgumentException.class,
564.           () -> RealRule.from(
565.               BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME),
566.               variations, localDateTimeJan10220000, localDateTimeJan12220000,
567.               BASE_SCALE, variator),
568.           "Unmatched end of reference window is not properly handled");
569.
570.       assertEquals(expectedMessage, thrown.getMessage(),
571.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
572.     }
573.
574.     /**
575.      * Test method for
576.      * {@link Rule#validateInputs(BaseValue, Rule[], LocalDateTime,
577.      * LocalDateTime, double)}.
578.      */
579.     @Test
580.     void testValidateInputs_variationsDoNotHaveRulesBaseValues() {
581.       String expectedMessage = "The given variations do not meet "
582.           + "specifications.";
583.       String expectedCause = "The base value of all rules must be equal to "
584.           + "given base value but the rule at position 0 does not comply.";
585.       RealRule var1 = RealRule.from(
586.           BaseValueFactory.jan1Jan31calcShort(BASE_VALUE_NAME), null,
587.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
```

```
588.          variator);
589.      RealRule[] variations = { var1 };
590.
591.      Exception thrown = assertThrows(IllegalArgumentException.class,
592.          () -> RealRule.from(
593.              BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME),
594.              variations, localDateTimeJan10220000, localDateTimeJan12220000,
595.              BASE_SCALE, variator),
596.          "Incorrect BaseValue in variations is not properly handled");
597.
598.      assertEquals(expectedMessage, thrown.getMessage(),
599.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
600.      assertEquals(expectedCause, thrown.getCause().getMessage(),
601.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
602.    }
603.
604.    /**
605.     * Test method for {@link Rule#weighVariations()}.
606.     */
607.    @Test
608.    void testWeighVariations_1Variation() {
609.      double expectedValue = 1;
610.      RealRule var1 = RealRule.from(
611.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
612.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
613.          variator);
614.      RealRule[] variations = { var1 };
615.
616.      RealRule realRule = RealRule.from(
617.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), variations,
618.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
619.          variator);
620.      double actualValue = realRule.getVariations()[0].getWeight();
621.
622.      assertEquals(expectedValue, actualValue,
623.          "Weight for 1 variation is not correctly calculated");
624.    }
625.
626.    /**
627.     * Test method for {@link Rule#weighVariations()}.
628.     */
629.    @Test
630.    void testWeighVariations_2Variations() {
631.      double[] expectedValue = { 0.5, 0.5 };
632.      RealRule var1 = RealRule.from(
633.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
634.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
635.          variator);
636.      RealRule var2 = RealRule.from(
637.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
638.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
639.          variator);
640.      RealRule[] variations = { var1, var2 };
641.
```

```
642.       RealRule realRule = RealRule.from(
643.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), variations,
644.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
645.           variator);
646.       double[] actualValue = { realRule.getVariations()[0].getWeight(),
647.           realRule.getVariations()[1].getWeight() };
648.
649.       assertArrayEquals(expectedValue, actualValue,
650.           "Weights for 2 variation are not correctly calculated");
651.     }
652.
653.     /**
654.      * Test method for {@link Rule#weighVariations()}.
655.      */
656.     @Test
657.     void testWeighVariations_3EqualVariations() {
658.       double[] expectedValue = { 1d / 3d, 1d / 3d, 1d / 3d };
659.       RealRule var1 = RealRule.from(
660.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
661.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
662.           variator);
663.       RealRule var2 = RealRule.from(
664.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
665.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
666.           variator);
667.       RealRule var3 = RealRule.from(
668.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
669.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
670.           variator);
671.       RealRule[] variations = { var1, var2, var3 };
672.
673.       RealRule realRule = RealRule.from(
674.           BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), variations,
675.           localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
676.           variator);
677.       double[] actualValue = { realRule.getVariations()[0].getWeight(),
678.           realRule.getVariations()[1].getWeight(),
679.           realRule.getVariations()[2].getWeight() };
680.
681.       assertArrayEquals(expectedValue, actualValue,
682.           "Weights for 3 equal variations are not correctly calculated");
683.     }
684.
685.     /**
686.      * Test method for {@link Rule#weighVariations()}.
687.      */
688.     @Test
689.     void testWeighVariations_3Variations() {
690.       double[] expectedValue = { //
691.           0.19285828960561063, // Excel: 0.192858289605609
692.           0.3259968088978676, // Excel: 0.325996808897865
693.           0.4811449014965218, // Excel: 0.481144901496526
694.       };
695.       double variator1 = 1;
```

```
696.      double variator2 = 3.19;
697.      double variator3 = -0.1;
698.      RealRule var1 = RealRule.from(
699.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
700.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
701.          variator1);
702.      RealRule var2 = RealRule.from(
703.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
704.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
705.          variator2);
706.      RealRule var3 = RealRule.from(
707.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
708.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
709.          variator3);
710.      RealRule[] variations = { var1, var2, var3 };
711.
712.      RealRule realRule = RealRule.from(
713.          BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), variations,
714.          localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
715.          variator);
716.      double[] actualValue = { realRule.getVariations()[0].getWeight(),
717.          realRule.getVariations()[1].getWeight(),
718.          realRule.getVariations()[2].getWeight() };
719.
720.      assertArrayEquals(expectedValue, actualValue,
721.          "Weights for 3 inequal variations are not correctly calculated");
722.    }
723.
724.    /**
725.     * Test method for {@link Rule#calculateForecastScalar()}.
726.     */
727.    @Test
728.    void testCalculateForecastScalar_refWindowOnFirstDayOfBaseValues() {
729.      String expectedMessage = "Reference window must not start on first "
730.          + "time interval of base value data.";
731.      double variator3 = 1;
732.
733.      Exception thrown = assertThrows(IllegalArgumentException.class,
734.          () -> RealRule.from(
735.              BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
736.              localDateTimeJan01220000, localDateTimeJan03220000, BASE_SCALE,
737.              variator3),
738.          "Illegal start of reference window is not properly handled.");
739.
740.      assertEquals(expectedMessage, thrown.getMessage(),
741.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
742.    }
743.
744.    /**
745.     * Test method for {@link Rule#calculateWeights(double[])}.
746.     */
747.    @Test
748.    void testCalculateWeights_negativeCorrelations() {
749.      double[] expectedValue = { //
```

```
750.            0.4999239558356957, // Excel: 0.499923955835696
751.            0.2500380220821522, // Excel: 0.250038022082152
752.            0.2500380220821522, // Excel: 0.250038022082152
753.        };
754.        double variator1 = -1;
755.        double variator2 = 0.5;
756.        double variator3 = 1;
757.        RealRule var1 = RealRule.from(
758.            BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
759.            localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
760.            variator1);
761.        RealRule var2 = RealRule.from(
762.            BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
763.            localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
764.            variator2);
765.        RealRule var3 = RealRule.from(
766.            BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), null,
767.            localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
768.            variator3);
769.        RealRule[] variations = { var1, var2, var3 };
770.
771.        RealRule realRule = RealRule.from(
772.            BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME), variations,
773.            localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
774.            variator);
775.        double[] actualValue = { realRule.getVariations()[0].getWeight(),
776.            realRule.getVariations()[1].getWeight(),
777.            realRule.getVariations()[2].getWeight() };
778.
779.        assertArrayEquals(expectedValue, actualValue,
780.            "Weights for variations with negative correlations are not "
781.            + "correctly calculated");
782.    }
783.
784.    /**
785.     * Test method for {@link Rule#extractRelevantForecastValues()}.
786.     */
787.    @Test
788.    void testGetRelevantForecastValues() {
789.      double[] expectedValues = { //
790.          12.66459427439483, // Excel: 12.6645942743948
791.          8.277321595406873, // Excel: 8.27732159540687
792.          9.058084130198298, // Excel: 9.0580841301983
793.      };
794.
795.      double[] actualValues = realRule.extractRelevantForecastValues();
796.
797.      assertArrayEquals(expectedValues, actualValues,
798.          "Relevant forecasts are not properly extracted");
799.    }
800.  }
```

Komponente VolatilityDifferenceTest

Listing 28: Komponente VolatilityDifferenceTest

```
1.    package de.rumford.tradingsystem;
2.
3.    import static org.junit.jupiter.api.Assertions.assertArrayEquals;
4.    import static org.junit.jupiter.api.Assertions.assertEquals;
5.    import static org.junit.jupiter.api.Assertions.assertThrows;
6.
7.    import java.time.LocalDate;
8.    import java.time.LocalDateTime;
9.    import java.time.LocalTime;
10.
11.   import org.apache.commons.lang3.ArrayUtils;
12.   import org.junit.jupiter.api.BeforeAll;
13.   import org.junit.jupiter.api.BeforeEach;
14.   import org.junit.jupiter.api.Test;
15.
16.   import de.rumford.tradingsystem.helper.BaseValueFactory;
17.   import de.rumford.tradingsystem.helper.ValueDateTupel;
18.
19.   /**
20.    * Test class for {@link VolatilityDifference}.
21.    *
22.    * @author Max Rumford
23.    *
24.    */
25.   class VolatilityDifferenceTest {
26.
27.     static final String BASE_VALUE_NAME = "Base Value";
28.     static final int BASE_SCALE = 10;
29.
30.     static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
31.         "Incorrect Exception message";
32.
33.     BaseValue baseValue;
34.     int lookbackWindow;
35.     static ValueDateTupel[] volatilityIndicesArray;
36.
37.     static LocalDateTime localDateTime2019Dec31220000;
38.     static LocalDateTime localDateTime2020Jan01220000;
39.     static LocalDateTime localDateTime2020Jan02220000;
40.     static LocalDateTime localDateTime2020Jan03220000;
41.     static LocalDateTime localDateTime2020Jan04220000;
42.     static LocalDateTime localDateTime2020Jan05220000;
43.     static LocalDateTime localDateTime2020Jan08220000;
44.     static LocalDateTime localDateTime2020Jan09220000;
45.     static LocalDateTime localDateTime2020Jan10220000;
46.     static LocalDateTime localDateTime2020Jan11220000;
47.     static LocalDateTime localDateTime2020Jan12220000;
48.     static LocalDateTime localDateTime2020Jan31220000;
49.
```

```
    VolatilityDifference volatilityDifference;
    VolatilityDifference volatilityDifference2;

    /**
     * @throws java.lang.Exception
     */
    @BeforeAll
    static void setUpBeforeClass() throws Exception {
      localDateTime2019Dec31220000 = LocalDateTime
          .of(LocalDate.of(2019, 12, 31), LocalTime.of(22, 0));
      localDateTime2020Jan01220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 1), LocalTime.of(22, 0));
      localDateTime2020Jan02220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 2), LocalTime.of(22, 0));
      localDateTime2020Jan03220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 3), LocalTime.of(22, 0));
      localDateTime2020Jan04220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 4), LocalTime.of(22, 0));
      localDateTime2020Jan05220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 5), LocalTime.of(22, 0));
      localDateTime2020Jan08220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 8), LocalTime.of(22, 0));
      localDateTime2020Jan09220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 9), LocalTime.of(22, 0));
      localDateTime2020Jan10220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 10), LocalTime.of(22, 0));
      localDateTime2020Jan11220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 11), LocalTime.of(22, 0));
      localDateTime2020Jan12220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 12), LocalTime.of(22, 0));
      localDateTime2020Jan31220000 = LocalDateTime
          .of(LocalDate.of(2020, 1, 31), LocalTime.of(22, 0));
    }

    /**
     * @throws java.lang.Exception
     */
    @BeforeEach
    void setUp() throws Exception {
      baseValue = BaseValueFactory.jan1Jan4calcShort(BASE_VALUE_NAME);

      lookbackWindow = 2;
    }

    /**
     * Test method for
     * {@link VolatilityDifference#VolatilityDifference(BaseValue,
     * VolatilityDifference[], LocalDateTime, LocalDateTime, int, double)}.
     */
    @Test
    void testVolatilityDifference() {
      volatilityDifference = new VolatilityDifference(baseValue, null,
          localDateTime2020Jan03220000, localDateTime2020Jan04220000,
          lookbackWindow, BASE_SCALE);
```

```
104.      volatilityDifference2 = new VolatilityDifference(baseValue, null,
105.          localDateTime2020Jan03220000, localDateTime2020Jan04220000,
106.          lookbackWindow, BASE_SCALE);
107.
108.      assertEquals(volatilityDifference, volatilityDifference2,
109.          "Two identical instances are not considered identical");
110.    }
111.
112.    /**
113.     * Test method for
114.     * {@link VolatilityDifference#VolatilityDifference(BaseValue,
115.     * VolatilityDifference[], LocalDateTime, LocalDateTime, int, double)}.
116.     */
117.    @Test
118.    void testVolatilityDifference_volatilityIndicesGiven() {
119.      ValueDateTupel volatilityIndex1 = new ValueDateTupel(
120.          localDateTime2020Jan01220000, Double.NaN);
121.      ValueDateTupel volatilityIndex2 = new ValueDateTupel(
122.          localDateTime2020Jan02220000, 100d);
123.      ValueDateTupel volatilityIndex3 = new ValueDateTupel(
124.          localDateTime2020Jan03220000, 5d);
125.      ValueDateTupel volatilityIndex4 = new ValueDateTupel(
126.          localDateTime2020Jan04220000, 10d);
127.      volatilityIndicesArray = ValueDateTupel.createEmptyArray();
128.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
129.          volatilityIndex1);
130.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
131.          volatilityIndex2);
132.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
133.          volatilityIndex3);
134.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
135.          volatilityIndex4);
136.
137.      VolatilityDifference volDif = new VolatilityDifference(baseValue, null,
138.          localDateTime2020Jan02220000, localDateTime2020Jan04220000,
139.          lookbackWindow, BASE_SCALE, volatilityIndicesArray);
140.      VolatilityDifference volDif2 = new VolatilityDifference(baseValue,
141.          null, localDateTime2020Jan02220000, localDateTime2020Jan04220000,
142.          lookbackWindow, BASE_SCALE, volatilityIndicesArray);
143.
144.      assertEquals(volDif, volDif2,
145.          "Two identical instances are not considered identical");
146.    }
147.
148.    /**
149.     * Test method for
150.     * {@link VolatilityDifference#calculateVolatilityIndices(BaseValue,
151.     * int)}.
152.     */
153.    @Test
154.    void testCalculateVolatilityIndices() {
155.      double expectedVolatilityValue2 = 0.5303300858899106; // Excel:
156.      // 0.530330085889911
157.      double expectedVolatilityValue3 = 0.6010407640085653; // Excel:
```

```
158.        // 0.601040764008565
159.        ValueDateTupel volatilityIndex1 = new ValueDateTupel(
160.            localDateTime2020Jan01220000, Double.NaN);
161.        ValueDateTupel volatilityIndex2 = new ValueDateTupel(
162.            localDateTime2020Jan02220000, Double.NaN);
163.        ValueDateTupel volatilityIndex3 = new ValueDateTupel(
164.            localDateTime2020Jan03220000, expectedVolatilityValue2);
165.        ValueDateTupel volatilityIndex4 = new ValueDateTupel(
166.            localDateTime2020Jan04220000, expectedVolatilityValue3);
167.        ValueDateTupel[] expectedValues = ValueDateTupel.createEmptyArray();
168.        expectedValues = ArrayUtils.add(expectedValues, volatilityIndex1);
169.        expectedValues = ArrayUtils.add(expectedValues, volatilityIndex2);
170.        expectedValues = ArrayUtils.add(expectedValues, volatilityIndex3);
171.        expectedValues = ArrayUtils.add(expectedValues, volatilityIndex4);
172.
173.        volatilityDifference = new VolatilityDifference(baseValue, null,
174.            localDateTime2020Jan03220000, localDateTime2020Jan04220000,
175.            lookbackWindow, BASE_SCALE);
176.        ValueDateTupel[] actualValues = volatilityDifference
177.            .getVolatilityIndices();
178.
179.        assertArrayEquals(expectedValues, actualValues,
180.            "Volatility index values are not properly calculated");
181.    }
182.
183.    /**
184.     * Test method for
185.     * {@link VolatilityDifference#calculateVolatilityIndices(BaseValue,
186.     * int)}.
187.     */
188.    @Test
189.    void testCalculateVolatilityIndices_lessBaseValuesThanLookbackWindow() {
190.      int lookbackWindowTooGreat = 10;
191.      String expectedMessage = "The amount of base values must not be "
192.          + "smaller than the lookback window. Number of base values: 4, "
193.          + "lookback window: 10.";
194.
195.      Exception thrown = assertThrows(IllegalArgumentException.class,
196.          () -> new VolatilityDifference(baseValue, null,
197.              localDateTime2020Jan02220000, localDateTime2020Jan04220000,
198.              lookbackWindowTooGreat, BASE_SCALE),
199.          "Too great of a lookback window is not correctly handled");
200.
201.      assertEquals(expectedMessage, thrown.getMessage(),
202.          "Too great of a lookback window is not correctly handled.");
203.    }
204.
205.    /**
206.     * Test method for
207.     * {@link VolatilityDifference#validateLookbackWindow(int)}.
208.     */
209.    @Test
210.    void testValidateLookbackWindow_lookbackWindow_1() {
211.      int lookbackWindowOne = 1;
```

```
212.      String expectedMessage = "Lookback window must be at least 2";
213.
214.      Exception thrown = assertThrows(IllegalArgumentException.class,
215.          () -> new VolatilityDifference(baseValue, null,
216.              localDateTime2020Jan03220000, localDateTime2020Jan04220000,
217.              lookbackWindowOne, BASE_SCALE),
218.          "Lookback window <= 1 is not correctly handled");
219.
220.      assertEquals(expectedMessage, thrown.getMessage(),
221.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
222.    }
223.
224.    /**
225.     * Test method for
226.     * {@link VolatilityDifference#validateVolatilityIndices(
227.     * ValueDateTupel[])}.
228.     */
229.    @Test
230.    void testValidateVolatilityIndices_volatilityIndicesArrayNull() {
231.      String expectedMessage = "Volatility indices must not be null.";
232.
233.      Exception thrown = assertThrows(IllegalArgumentException.class,
234.          () -> new VolatilityDifference(baseValue, null,
235.              localDateTime2020Jan02220000, localDateTime2020Jan04220000,
236.              lookbackWindow, BASE_SCALE, null),
237.          "Empty array of volatlilty indices is not correctly handled");
238.
239.      assertEquals(expectedMessage, thrown.getMessage(),
240.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
241.    }
242.
243.    /**
244.     * Test method for
245.     * {@link VolatilityDifference#validateVolatilityIndices(
246.     * ValueDateTupel[])}.
247.     */
248.    @Test
249.    void testValidateVolatilityIndices_emptyVolatilityIndicesArray() {
250.      String expectedMessage = "Volatility indices must not be an empty "
251.          + "array";
252.
253.      ValueDateTupel[] emptyVolatilityIndicesArray = ValueDateTupel
254.          .createEmptyArray();
255.
256.      Exception thrown = assertThrows(IllegalArgumentException.class,
257.          () -> new VolatilityDifference(baseValue, null,
258.              localDateTime2020Jan02220000, localDateTime2020Jan04220000,
259.              lookbackWindow, BASE_SCALE, emptyVolatilityIndicesArray),
260.          "Empty array of volatlilty indices is not correctly handled");
261.
262.      assertEquals(expectedMessage, thrown.getMessage(),
263.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
264.    }
265.
```

```
266.    /**
267.     * Test method for
268.     * {@link VolatilityDifference#validateVolatilityIndices(
269.     * ValueDateTupel[])}.
270.     */
271.    @Test
272.    void testValidateVolatilityIndices_unsortedVolatilityIndicesArray() {
273.      String expectedMessage = "Given volatility indices are not properly "
274.          + "sorted or there are duplicate LocalDateTime values";
275.
276.      ValueDateTupel volatilityIndex1 = new ValueDateTupel(
277.          localDateTime2020Jan01220000, Double.NaN);
278.      ValueDateTupel volatilityIndex2 = new ValueDateTupel(
279.          localDateTime2020Jan02220000, 100d);
280.      ValueDateTupel volatilityIndex3 = new ValueDateTupel(
281.          localDateTime2020Jan04220000, 5d);
282.      ValueDateTupel volatilityIndex4 = new ValueDateTupel(
283.          localDateTime2020Jan03220000, 10d);
284.      volatilityIndicesArray = ValueDateTupel.createEmptyArray();
285.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
286.          volatilityIndex1);
287.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
288.          volatilityIndex2);
289.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
290.          volatilityIndex3);
291.      volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
292.          volatilityIndex4);
293.
294.      Exception thrown = assertThrows(IllegalArgumentException.class,
295.          () -> new VolatilityDifference(baseValue, null,
296.              localDateTime2020Jan02220000, localDateTime2020Jan04220000,
297.              lookbackWindow, BASE_SCALE, volatilityIndicesArray),
298.          "Improperly sorted volatility indices are not correctly handled");
299.
300.      assertEquals(expectedMessage, thrown.getMessage(),
301.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
302.    }
303.
304.    /**
305.     * Test method for
306.     * {@link VolatilityDifference#validateVolatilityIndices(
307.     * ValueDateTupel[])}.
308.     */
309.    @Test
310.    void testValidateVolatilityIndices_duplicatesInVolatilityIndicesArray() {
311.      String expectedMessage = "Given volatility indices are not properly "
312.          + "sorted or there are duplicate LocalDateTime values";
313.
314.      ValueDateTupel volatilityIndex1 = new ValueDateTupel(
315.          localDateTime2020Jan01220000, Double.NaN);
316.      ValueDateTupel volatilityIndex2 = new ValueDateTupel(
317.          localDateTime2020Jan02220000, 100d);
318.      ValueDateTupel volatilityIndex3 = new ValueDateTupel(
319.          localDateTime2020Jan02220000, 5d);
```

```
320.        ValueDateTupel volatilityIndex4 = new ValueDateTupel(
321.            localDateTime2020Jan04220000, 10d);
322.        volatilityIndicesArray = ValueDateTupel.createEmptyArray();
323.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
324.            volatilityIndex1);
325.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
326.            volatilityIndex2);
327.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
328.            volatilityIndex3);
329.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
330.            volatilityIndex4);
331.
332.        Exception thrown = assertThrows(IllegalArgumentException.class,
333.            () -> new VolatilityDifference(baseValue, null,
334.                localDateTime2020Jan02220000, localDateTime2020Jan04220000,
335.                lookbackWindow, BASE_SCALE, volatilityIndicesArray),
336.            "Duplicate volatility indices are not correctly handled");
337.
338.        assertEquals(expectedMessage, thrown.getMessage(),
339.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
340.    }
341.
342.    /**
343.     * Test method for
344.     * {@link VolatilityDifference#validateVolatilityIndices(
345.     * ValueDateTupel[])}.
346.     */
347.    @Test
348.    void testValidateVolatilityIndices_startOfRefWindowNotInVolIndicesArr() {
349.      String expectedMessage = "Giving volatility indices do not meet "
350.          + "specificiation.";
351.      String expectedCause = "Given values do not include given start "
352.          + "value for time window";
353.
354.        ValueDateTupel volatilityIndex2 = new ValueDateTupel(
355.            localDateTime2020Jan03220000, Double.NaN);
356.        ValueDateTupel volatilityIndex3 = new ValueDateTupel(
357.            localDateTime2020Jan04220000, 5d);
358.        ValueDateTupel volatilityIndex4 = new ValueDateTupel(
359.            localDateTime2020Jan05220000, 10d);
360.        volatilityIndicesArray = ValueDateTupel.createEmptyArray();
361.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
362.            volatilityIndex2);
363.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
364.            volatilityIndex3);
365.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
366.            volatilityIndex4);
367.
368.        Exception thrown = assertThrows(IllegalArgumentException.class,
369.            () -> new VolatilityDifference(baseValue, null,
370.                localDateTime2020Jan02220000, localDateTime2020Jan04220000,
371.                lookbackWindow, BASE_SCALE, volatilityIndicesArray),
372.            "Invalid start of reference window value is not correctly "
373.            + "handled");
```

```
374.
375.       assertEquals(expectedMessage, thrown.getMessage(),
376.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
377.       assertEquals(expectedCause, thrown.getCause().getMessage(),
378.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
379.     }
380.
381.     /**
382.      * Test method for
383.      * {@link VolatilityDifference#validateVolatilityIndices(
384.      * ValueDateTupel[])}.
385.      */
386.     @Test
387.     void testValidateVolatilityIndices_endOfRefWindowNotInVolIndicesArray() {
388.       String expectedMessage = "Giving volatility indices do not meet "
389.           + "specificiation.";
390.       String expectedCause = "Given values do not include given end value "
391.           + "for time window";
392.
393.       ValueDateTupel volatilityIndex1 = new ValueDateTupel(
394.           localDateTime2020Jan01220000, Double.NaN);
395.       ValueDateTupel volatilityIndex2 = new ValueDateTupel(
396.           localDateTime2020Jan02220000, 100d);
397.       ValueDateTupel volatilityIndex3 = new ValueDateTupel(
398.           localDateTime2020Jan03220000, 5d);
399.       volatilityIndicesArray = ValueDateTupel.createEmptyArray();
400.       volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
401.           volatilityIndex1);
402.       volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
403.           volatilityIndex2);
404.       volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
405.           volatilityIndex3);
406.
407.       Exception thrown = assertThrows(IllegalArgumentException.class,
408.           () -> new VolatilityDifference(baseValue, null,
409.               localDateTime2020Jan02220000, localDateTime2020Jan04220000,
410.               lookbackWindow, BASE_SCALE, volatilityIndicesArray),
411.           "Invalid end of reference window value is not correctly handled");
412.
413.       assertEquals(expectedMessage, thrown.getMessage(),
414.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
415.       assertEquals(expectedCause, thrown.getCause().getMessage(),
416.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
417.     }
418.
419.     /**
420.      * Test method for
421.      * {@link VolatilityDifference#validateVolatilityIndices(
422.      * ValueDateTupel[])}.
423.      */
424.     @Test
425.     void testValidateVolatilityIndices_nansInRelevantAreaOfVolIndicesArr() {
426.       String expectedMessage = "There must not be NaN-Values in the given "
427.           + "volatility indices values in the area delimited by "
```

```
            + "startOfReferenceWindow and endOfReferenceWindow";

        ValueDateTupel volatilityIndex1 = new ValueDateTupel(
            localDateTime2020Jan01220000, 100d);
        ValueDateTupel volatilityIndex2 = new ValueDateTupel(
            localDateTime2020Jan02220000, 100d);
        ValueDateTupel volatilityIndex3 = new ValueDateTupel(
            localDateTime2020Jan03220000, Double.NaN);
        ValueDateTupel volatilityIndex4 = new ValueDateTupel(
            localDateTime2020Jan04220000, 10d);
        volatilityIndicesArray = ValueDateTupel.createEmptyArray();
        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
            volatilityIndex1);
        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
            volatilityIndex2);
        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
            volatilityIndex3);
        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
            volatilityIndex4);

        Exception thrown = assertThrows(IllegalArgumentException.class,
            () -> new VolatilityDifference(baseValue, null,
                localDateTime2020Jan02220000, localDateTime2020Jan04220000,
                lookbackWindow, BASE_SCALE, volatilityIndicesArray),
            "NaNs in relevant area of volatility indices are not correctly "
            + "handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for
     * {@link VolatilityDifference#validateVolatilityIndices(
     * ValueDateTupel[])}.
     */
    @Test
    void testValidateVolatilityIndices_baseValueVolIndicesNotAligned() {
      String expectedMessage = "Base value and volatility index values are "
            + "not properly aligned. Utilize "
            + "ValueDateTupel.alignDates(ValueDateTupel[][]) before creating a"
            + " new VolatilityDifference.";

        ValueDateTupel volatilityIndex1 = new ValueDateTupel(
            localDateTime2020Jan01220000, Double.NaN);
        ValueDateTupel volatilityIndex2 = new ValueDateTupel(
            localDateTime2020Jan02220000, 100d);
        ValueDateTupel volatilityIndex3 = new ValueDateTupel(
            localDateTime2020Jan03220000, 5d);
        ValueDateTupel volatilityIndex4 = new ValueDateTupel(
            localDateTime2020Jan04220000, 10d);
        ValueDateTupel volatilityIndex5 = new ValueDateTupel(
            localDateTime2020Jan05220000, 99d);
        volatilityIndicesArray = ValueDateTupel.createEmptyArray();
```

```
482.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
483.            volatilityIndex1);
484.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
485.            volatilityIndex2);
486.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
487.            volatilityIndex3);
488.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
489.            volatilityIndex4);
490.        volatilityIndicesArray = ArrayUtils.add(volatilityIndicesArray,
491.            volatilityIndex5);
492.
493.        Exception thrown = assertThrows(IllegalArgumentException.class,
494.            () -> new VolatilityDifference(baseValue, null,
495.                localDateTime2020Jan02220000, localDateTime2020Jan04220000,
496.                lookbackWindow, BASE_SCALE, volatilityIndicesArray),
497.            "Not aligned base value and volatility indices are not correctly "
498.            + "handled");
499.
500.        assertEquals(expectedMessage, thrown.getMessage(),
501.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
502.    }
503.
504.    /**
505.     * Test method for
506.     * {@link VolatilityDifference#calculateRawForecast(double)}.
507.     */
508.    @Test
509.    void testCalculateRawForecast() {
510.      baseValue = BaseValueFactory.jan1Jan31calcShort(BASE_VALUE_NAME);
511.      double expectedValue = -0.5604475969404489; /*
512.                                                   * Excel:
513.                                                   * -0.560447596940449
514.                                                   */
515.
516.        VolatilityDifference volDif = new VolatilityDifference(baseValue, null,
517.            localDateTime2020Jan08220000, localDateTime2020Jan10220000,
518.            lookbackWindow, BASE_SCALE);
519.        double actualValue = volDif
520.            .calculateRawForecast(localDateTime2020Jan11220000);
521.
522.        assertEquals(expectedValue, actualValue,
523.            "Raw Forecast is not correctly calculated");
524.    }
525.  }
```

Komponente EWMACTest

Listing 29: Komponente EWMACTest

```
1.    package de.rumford.tradingsystem;
2.
3.    import static org.junit.jupiter.api.Assertions.assertEquals;
4.    import static org.junit.jupiter.api.Assertions.assertThrows;
5.    import static org.junit.jupiter.api.Assertions.assertTrue;
6.
7.    import java.time.LocalDate;
8.    import java.time.LocalDateTime;
9.    import java.time.LocalTime;
10.
11.   import org.junit.jupiter.api.BeforeAll;
12.   import org.junit.jupiter.api.BeforeEach;
13.   import org.junit.jupiter.api.Test;
14.
15.   import de.rumford.tradingsystem.helper.BaseValueFactory;
16.
17.   /**
18.    * Test class for {@link EWMAC}.
19.    *
20.    * @author Max Rumford
21.    *
22.    */
23.   class EWMACTest {
24.
25.     static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
26.         "Incorrect Exception message";
27.
28.     EWMAC ewmac;
29.     int shortHorizon;
30.     int longHorizon;
31.     static BaseValue baseValue;
32.
33.     static final String BASE_VALUE_NAME = "Base value name";
34.     static final int BASE_SCALE = 10;
35.
36.     static LocalDateTime localDateTimeJan01220000;
37.     static LocalDateTime localDateTimeJan02220000;
38.     static LocalDateTime localDateTimeJan03220000;
39.     static LocalDateTime localDateTimeJan04220000;
40.     static LocalDateTime localDateTimeJan05220000;
41.     static LocalDateTime localDateTimeJan06220000;
42.     static LocalDateTime localDateTimeJan07220000;
43.     static LocalDateTime localDateTimeJan08220000;
44.     static LocalDateTime localDateTimeJan09220000;
45.     static LocalDateTime localDateTimeJan10220000;
46.     static LocalDateTime localDateTimeJan11220000;
47.     static LocalDateTime localDateTimeJan12220000;
48.     static LocalDateTime localDateTimeJan13220000;
49.     static LocalDateTime localDateTimeJan14220000;
```

```
50.     static LocalDateTime localDateTimeJan15220000;
51.     static LocalDateTime localDateTimeJan16220000;
52.     static LocalDateTime localDateTimeJan17220000;
53.     static LocalDateTime localDateTimeJan18220000;
54.     static LocalDateTime localDateTimeJan19220000;
55.     static LocalDateTime localDateTimeJan20220000;
56.     static LocalDateTime localDateTimeJan21220000;
57.     static LocalDateTime localDateTimeJan22220000;
58.     static LocalDateTime localDateTimeJan23220000;
59.     static LocalDateTime localDateTimeJan24220000;
60.     static LocalDateTime localDateTimeJan25220000;
61.     static LocalDateTime localDateTimeJan26220000;
62.     static LocalDateTime localDateTimeJan27220000;
63.     static LocalDateTime localDateTimeJan28220000;
64.     static LocalDateTime localDateTimeJan29220000;
65.     static LocalDateTime localDateTimeJan30220000;
66.     static LocalDateTime localDateTimeJan31220000;
67.
68.     @BeforeAll
69.     static void setUpBeforeClass() {
70.       baseValue = BaseValueFactory.jan1Jan31calcShort(BASE_VALUE_NAME);
71.
72.       localDateTimeJan01220000 = LocalDateTime.of(LocalDate.of(2020, 1, 1),
73.           LocalTime.of(22, 0));
74.       localDateTimeJan02220000 = LocalDateTime.of(LocalDate.of(2020, 1, 2),
75.           LocalTime.of(22, 0));
76.       localDateTimeJan03220000 = LocalDateTime.of(LocalDate.of(2020, 1, 3),
77.           LocalTime.of(22, 0));
78.       localDateTimeJan04220000 = LocalDateTime.of(LocalDate.of(2020, 1, 4),
79.           LocalTime.of(22, 0));
80.       localDateTimeJan05220000 = LocalDateTime.of(LocalDate.of(2020, 1, 5),
81.           LocalTime.of(22, 0));
82.       localDateTimeJan06220000 = LocalDateTime.of(LocalDate.of(2020, 1, 6),
83.           LocalTime.of(22, 0));
84.       localDateTimeJan07220000 = LocalDateTime.of(LocalDate.of(2020, 1, 7),
85.           LocalTime.of(22, 0));
86.       localDateTimeJan08220000 = LocalDateTime.of(LocalDate.of(2020, 1, 8),
87.           LocalTime.of(22, 0));
88.       localDateTimeJan09220000 = LocalDateTime.of(LocalDate.of(2020, 1, 9),
89.           LocalTime.of(22, 0));
90.       localDateTimeJan10220000 = LocalDateTime.of(LocalDate.of(2020, 1, 10),
91.           LocalTime.of(22, 0));
92.       localDateTimeJan11220000 = LocalDateTime.of(LocalDate.of(2020, 1, 11),
93.           LocalTime.of(22, 0));
94.       localDateTimeJan12220000 = LocalDateTime.of(LocalDate.of(2020, 1, 12),
95.           LocalTime.of(22, 0));
96.       localDateTimeJan13220000 = LocalDateTime.of(LocalDate.of(2020, 1, 13),
97.           LocalTime.of(22, 0));
98.       localDateTimeJan14220000 = LocalDateTime.of(LocalDate.of(2020, 1, 14),
99.           LocalTime.of(22, 0));
100.      localDateTimeJan15220000 = LocalDateTime.of(LocalDate.of(2020, 1, 15),
101.          LocalTime.of(22, 0));
102.      localDateTimeJan16220000 = LocalDateTime.of(LocalDate.of(2020, 1, 16),
103.          LocalTime.of(22, 0));
```

```
104.        localDateTimeJan17220000 = LocalDateTime.of(LocalDate.of(2020, 1, 17),
105.            LocalTime.of(22, 0));
106.        localDateTimeJan18220000 = LocalDateTime.of(LocalDate.of(2020, 1, 18),
107.            LocalTime.of(22, 0));
108.        localDateTimeJan19220000 = LocalDateTime.of(LocalDate.of(2020, 1, 19),
109.            LocalTime.of(22, 0));
110.        localDateTimeJan20220000 = LocalDateTime.of(LocalDate.of(2020, 1, 20),
111.            LocalTime.of(22, 0));
112.        localDateTimeJan21220000 = LocalDateTime.of(LocalDate.of(2020, 1, 21),
113.            LocalTime.of(22, 0));
114.        localDateTimeJan22220000 = LocalDateTime.of(LocalDate.of(2020, 1, 22),
115.            LocalTime.of(22, 0));
116.        localDateTimeJan23220000 = LocalDateTime.of(LocalDate.of(2020, 1, 23),
117.            LocalTime.of(22, 0));
118.        localDateTimeJan24220000 = LocalDateTime.of(LocalDate.of(2020, 1, 24),
119.            LocalTime.of(22, 0));
120.        localDateTimeJan25220000 = LocalDateTime.of(LocalDate.of(2020, 1, 25),
121.            LocalTime.of(22, 0));
122.        localDateTimeJan26220000 = LocalDateTime.of(LocalDate.of(2020, 1, 26),
123.            LocalTime.of(22, 0));
124.        localDateTimeJan27220000 = LocalDateTime.of(LocalDate.of(2020, 1, 27),
125.            LocalTime.of(22, 0));
126.        localDateTimeJan28220000 = LocalDateTime.of(LocalDate.of(2020, 1, 28),
127.            LocalTime.of(22, 0));
128.        localDateTimeJan29220000 = LocalDateTime.of(LocalDate.of(2020, 1, 29),
129.            LocalTime.of(22, 0));
130.        localDateTimeJan30220000 = LocalDateTime.of(LocalDate.of(2020, 1, 30),
131.            LocalTime.of(22, 0));
132.        localDateTimeJan31220000 = LocalDateTime.of(LocalDate.of(2020, 1, 31),
133.            LocalTime.of(22, 0));
134.    }
135.
136.    @BeforeEach
137.    void setUp() {
138.      shortHorizon = 2;
139.      longHorizon = 8;
140.      ewmac = new EWMAC(baseValue, null, localDateTimeJan08220000,
141.          localDateTimeJan10220000, longHorizon, shortHorizon, BASE_SCALE);
142.    }
143.
144.    /**
145.     * Test method for
146.     * {@link EWMAC#EWMAC(BaseValue, Rule[], LocalDateTime, LocalDateTime,
147.     * int, int, double)}.
148.     */
149.    @Test
150.    void testEWMAC() {
151.      EWMAC ewmac2 = new EWMAC(baseValue, null, localDateTimeJan08220000,
152.          localDateTimeJan10220000, longHorizon, shortHorizon, BASE_SCALE);
153.
154.      assertEquals(ewmac, ewmac2, "Two identical instances do not equal");
155.    }
156.
157.    /**
```

```
158.      * Test method for {@link EWMAC#validateHorizonValues(LocalDateTime)}.
159.      */
160.     @Test
161.     void testValidateHorizonValues_longSmallerThanShort_noVariations() {
162.       int shortHorizonValue = 8;
163.       int longHorizonValue = 4;
164.       String expectedMessage = "The long horizon must be greater than the "
165.           + "short horizon";
166.
167.       Exception thrown = assertThrows(IllegalArgumentException.class,
168.           () -> new EWMAC(baseValue, null, localDateTimeJan08220000,
169.               localDateTimeJan10220000, longHorizonValue, shortHorizonValue,
170.               BASE_SCALE),
171.           "Short horizon greater than long horizon is not properly handled");
172.
173.       assertEquals(expectedMessage, thrown.getMessage(),
174.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
175.     }
176.
177.     /**
178.      * Test method for {@link EWMAC#validateHorizonValues(LocalDateTime)}.
179.      */
180.     @Test
181.     void testValidateHorizonValues_longSmallerThanShort_withVariations() {
182.       EWMAC[] variations = {
183.           new EWMAC(baseValue, null, localDateTimeJan08220000,
184.               localDateTimeJan10220000, 8, 4, BASE_SCALE) };
185.       int shortHorizonValue = 8;
186.       int longHorizonValue = 4;
187.
188.       assertTrue(
189.           new EWMAC(baseValue, variations, localDateTimeJan08220000,
190.               localDateTimeJan10220000, longHorizonValue, shortHorizonValue,
191.               BASE_SCALE) instanceof EWMAC,
192.           "horizon values are not properly ignored when rule has "
193.           + "variations");
194.     }
195.
196.     /**
197.      * Test method for {@link EWMAC#validateHorizonValues(LocalDateTime)}.
198.      */
199.     @Test
200.     void testValidateHorizonValues_longEqualsShort() {
201.       int shortHorizonValue = 4;
202.       int longHorizonValue = 4;
203.       String expectedMessage = "The long horizon must be greater than the "
204.           + "short horizon";
205.
206.       Exception thrown = assertThrows(IllegalArgumentException.class,
207.           () -> new EWMAC(baseValue, null, localDateTimeJan08220000,
208.               localDateTimeJan10220000, longHorizonValue, shortHorizonValue,
209.               BASE_SCALE),
210.           "Short horizon equal to long horizon is not properly handled");
211.
```

```
212.       assertEquals(expectedMessage, thrown.getMessage(),
213.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
214.     }
215.
216.     /**
217.      * Test method for {@link EWMAC#validateHorizonValues(LocalDateTime)}.
218.      */
219.     @Test
220.     void testValidateHorizonValues_shortLessThan2_noVariations() {
221.       int shortHorizonValue = 1;
222.       int longHorizonValue = 4;
223.       String expectedMessage = "The short horizon must not be < 2";
224.
225.       Exception thrown = assertThrows(IllegalArgumentException.class,
226.           () -> new EWMAC(baseValue, null, localDateTimeJan08220000,
227.               localDateTimeJan10220000, longHorizonValue, shortHorizonValue,
228.               BASE_SCALE),
229.           "Short horizon < 2 is not properly handled");
230.
231.       assertEquals(expectedMessage, thrown.getMessage(),
232.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
233.     }
234.
235.     /**
236.      * Test method for {@link EWMAC#validateHorizonValues(LocalDateTime)}.
237.      */
238.     @Test
239.     void testValidateHorizonValues_shortLessThan2_withVariations() {
240.       EWMAC[] variations = {
241.           new EWMAC(baseValue, null, localDateTimeJan08220000,
242.               localDateTimeJan10220000, 8, 4, BASE_SCALE) };
243.       int shortHorizonValue = 1;
244.       int longHorizonValue = 4;
245.
246.       assertTrue(
247.           new EWMAC(baseValue, variations, localDateTimeJan08220000,
248.               localDateTimeJan10220000, longHorizonValue, shortHorizonValue,
249.               BASE_SCALE) instanceof EWMAC,
250.           "horizon values are not properly ignored when rule has "
251.           + "variations");
252.     }
253.
254.     /**
255.      * Test method for {@link EWMAC#calculateRawForecast(LocalDateTime)}.
256.      */
257.     @Test
258.     void testCalculateRawForecast_negativeRawForecast() {
259.       double expectedValue = -13.177732526197076; // Excel: -13.1777325261971
260.
261.       double actualValue = ewmac
262.           .calculateRawForecast(localDateTimeJan13220000);
263.
264.       assertEquals(expectedValue, actualValue,
265.           "Negative raw Forecast is not correctly calculated");
```

```
266.     }
267.
268.     /**
269.      * Test method for {@link EWMAC#calculateRawForecast(LocalDateTime)}.
270.      */
271.     @Test
272.     void testCalculateRawForecast_positiveRawForecast() {
273.       double expectedValue = 32.171834876807424; // Excel: 32.1718348768074
274.
275.       double actualValue = ewmac
276.           .calculateRawForecast(localDateTimeJan08220000);
277.
278.       assertEquals(expectedValue, actualValue,
279.           "Positive raw Forecast is not correctly calculated");
280.     }
281.   }
```

Komponente EWMATest

Listing 30: Komponente EWMATest

```
1.  package de.rumford.tradingsystem;
2.
3.  import static org.junit.jupiter.api.Assertions.assertEquals;
4.  import static org.junit.jupiter.api.Assertions.assertThrows;
5.  import static org.junit.jupiter.api.Assertions.assertTrue;
6.
7.  import org.junit.jupiter.api.BeforeEach;
8.  import org.junit.jupiter.api.Test;
9.
10. import de.rumford.tradingsystem.helper.BaseValueFactory;
11. import de.rumford.tradingsystem.helper.ValueDateTupel;
12.
13. /**
14.  * Test class for {@link EWMA}.
15.  *
16.  * @author Max Rumford
17.  *
18.  */
19. class EWMATest {
20.
21.   static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
22.       "Incorrect Exception message";
23.
24.   EWMA ewma2;
25.   EWMA ewma2_1;
26.   EWMA ewma4;
27.   EWMA ewma8;
28.
29.   BaseValue baseValue = BaseValueFactory
30.       .jan1Jan5calcShort("My base value");
31.
32.   @BeforeEach
33.   void setUp() throws Exception {
34.     ewma2 = new EWMA(baseValue.getValues(), 2);
35.     ewma2_1 = new EWMA(baseValue.getValues(), 2);
36.     ewma4 = new EWMA(baseValue.getValues(), 4);
37.     ewma8 = new EWMA(baseValue.getValues(), 8);
38.   }
39.
40.   /**
41.    * Test method for {@link EWMA#EWMA(ValueDateTupel[], int)}.
42.    */
43.   @Test
44.   void testEWMA_ewma_instanceof_EWMA() {
45.     assertTrue(ewma2 instanceof EWMA, "ewma2 is instanceof EWMA");
46.     assertEquals(ewma2, ewma2_1,
47.         "Two EWMAs with the same horizon are equal");
48.   }
49.
```

```
50.     /**
51.      * Test method for {@link EWMA#validateHorizon(int)}.
52.      */
53.     @Test
54.     void testValidateHorizon_Horizon1() {
55.       int horizonOf1 = 1;
56.       String expectedMessage = "The horizon must not be < 2";
57.
58.       Exception thrown = assertThrows(IllegalArgumentException.class,
59.           () -> new EWMA(baseValue.getValues(), horizonOf1),
60.           "Horizon less than 2 is not properly handled.");
61.
62.       assertEquals(expectedMessage, thrown.getMessage(),
63.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
64.     }
65.
66.     /**
67.      * Test method for {@link EWMA#validateBaseValues(ValueDateTupel[])}.
68.      */
69.     @Test
70.     void testValidateBaseValues_emptyBaseValuesArray() {
71.       String expectedMessage = "The given values do not meet the "
72.           + "specifications.";
73.       String expectedCause = "Values must not be an empty array";
74.
75.       ValueDateTupel[] emptyValuesArray = ValueDateTupel.createEmptyArray();
76.
77.       Exception thrown = assertThrows(IllegalArgumentException.class,
78.           () -> new EWMA(emptyValuesArray, 2),
79.           "Empty base values array is not properly handled.");
80.
81.       assertEquals(expectedMessage, thrown.getMessage(),
82.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
83.       assertEquals(expectedCause, thrown.getCause().getMessage(),
84.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
85.     }
86.
87.     /**
88.      * Test method for {@link EWMA#calculateEWMA(double, double)}.
89.      */
90.     @Test
91.     void testCalculateEWMA_Horizon2_baseValue786point75_Is_524point5() {
92.       double calculatedValue = ewma2.calculateEWMA(0d, 786.75d);
93.       double expectedValue = 524.5d;
94.       assertEquals(expectedValue, calculatedValue,
95.           "Calculated EWMA is expected EWMA");
96.     }
97.
98.     /**
99.      * Test method for {@link EWMA#calculateEWMA(double, double)}.
100.     */
101.    @Test
102.    void testCalculateEWMA_Horizon8_baseValue786point75_Is_174point833() {
103.      double calculatedValue = ewma2.calculateEWMA(0d, 100d);
```

```
104.       double expectedValue = (0d + 2d / 3d) * 100d;
105.       assertEquals(expectedValue, calculatedValue,
106.           "Calculated EWMA is expected EWMA");
107.     }
108.
109.     /**
110.      * Test method for {@link EWMA#calculateDecay(int)}.
111.      */
112.     @Test
113.     void testCalculateDecay_DecayForHorizon2_Is_point67() {
114.       double expectedValue = 2d / 3d;
115.       assertEquals(expectedValue, ewma2.getDecay(),
116.           "Calculated Decay is expected Decay");
117.     }
118.   }
```

Komponente BaseValueTest

Listing 31: Komponente BaseValueTest

```
1.  package de.rumford.tradingsystem;
2.
3.  import static org.junit.jupiter.api.Assertions.assertArrayEquals;
4.  import static org.junit.jupiter.api.Assertions.assertEquals;
5.  import static org.junit.jupiter.api.Assertions.assertThrows;
6.
7.  import java.time.LocalDate;
8.  import java.time.LocalDateTime;
9.  import java.time.LocalTime;
10.
11. import org.apache.commons.lang3.ArrayUtils;
12. import org.junit.jupiter.api.BeforeAll;
13. import org.junit.jupiter.api.BeforeEach;
14. import org.junit.jupiter.api.Test;
15.
16. import de.rumford.tradingsystem.helper.ValueDateTupel;
17.
18. /**
19.  * Test class for {@link BaseValue}.
20.  *
21.  * @author Max Rumford
22.  *
23.  */
24. class BaseValueTest {
25.
26.   static final String NAME_OF_TEST_BASE_VALUES = "Test Base Value";
27.   static final String EMPTY_STRING = "";
28.   static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
29.       "Incorrect Exception message";
30.
31.   final static int NUMBER_OF_VALUES = 4;
32.   static ValueDateTupel valuedatetupel1;
33.   static ValueDateTupel valuedatetupel2;
34.   static ValueDateTupel valuedatetupel3;
35.   static ValueDateTupel valuedatetupel4;
36.   static ValueDateTupel valuedatetupel5;
37.   static ValueDateTupel valuedatetupel6;
38.   static ValueDateTupel valuedatetupel7;
39.   static ValueDateTupel valuedatetupel8;
40.
41.   static ValueDateTupel[] values;
42.   static ValueDateTupel[] shortValues;
43.   static ValueDateTupel[] emptyValues;
44.
45.   BaseValue baseValue;
46.   BaseValue baseValue2;
47.
48.   static LocalDateTime localDateTimeJan01_22_00_00;
49.   static LocalDateTime localDateTimeJan02_22_00_00;
```

```
50.     static LocalDateTime localDateTimeJan03_22_00_00;
51.     static LocalDateTime localDateTimeJan04_22_00_00;
52.     static LocalDateTime localDateTimeJan05_22_00_00;
53.
54.     @BeforeAll
55.     static void setUpBeforeClass() throws Exception {
56.       localDateTimeJan01_22_00_00 = LocalDateTime
57.           .of(LocalDate.of(2020, 1, 1), LocalTime.of(22, 0));
58.       localDateTimeJan02_22_00_00 = LocalDateTime
59.           .of(LocalDate.of(2020, 1, 2), LocalTime.of(22, 0));
60.       localDateTimeJan03_22_00_00 = LocalDateTime
61.           .of(LocalDate.of(2020, 1, 3), LocalTime.of(22, 0));
62.       localDateTimeJan04_22_00_00 = LocalDateTime
63.           .of(LocalDate.of(2020, 1, 4), LocalTime.of(22, 0));
64.       localDateTimeJan05_22_00_00 = LocalDateTime
65.           .of(LocalDate.of(2020, 1, 5), LocalTime.of(22, 0));
66.
67.       valuedatetupel1 = new ValueDateTupel(localDateTimeJan01_22_00_00,
68.           200d);
69.       valuedatetupel2 = new ValueDateTupel(localDateTimeJan02_22_00_00,
70.           400d);
71.       valuedatetupel3 = new ValueDateTupel(localDateTimeJan03_22_00_00,
72.           500d);
73.       valuedatetupel4 = new ValueDateTupel(localDateTimeJan04_22_00_00,
74.           400d);
75.       valuedatetupel5 = new ValueDateTupel(localDateTimeJan01_22_00_00,
76.           1000d);
77.       valuedatetupel6 = new ValueDateTupel(localDateTimeJan02_22_00_00,
78.           500d);
79.       valuedatetupel7 = new ValueDateTupel(localDateTimeJan03_22_00_00,
80.           375d);
81.       valuedatetupel8 = new ValueDateTupel(localDateTimeJan04_22_00_00,
82.           450d);
83.
84.       shortValues = ValueDateTupel.createEmptyArray();
85.       shortValues = ArrayUtils.add(shortValues, valuedatetupel5);
86.       shortValues = ArrayUtils.add(shortValues, valuedatetupel6);
87.       shortValues = ArrayUtils.add(shortValues, valuedatetupel7);
88.       shortValues = ArrayUtils.add(shortValues, valuedatetupel8);
89.
90.       emptyValues = ValueDateTupel.createEmptyArray();
91.     }
92.
93.     @BeforeEach
94.     void setUp() throws Exception {
95.       values = ValueDateTupel.createEmptyArray();
96.       values = ArrayUtils.add(values, valuedatetupel1);
97.       values = ArrayUtils.add(values, valuedatetupel2);
98.       values = ArrayUtils.add(values, valuedatetupel3);
99.       values = ArrayUtils.add(values, valuedatetupel4);
100.
101.      shortValues = ValueDateTupel.createEmptyArray();
102.      shortValues = ArrayUtils.add(shortValues, valuedatetupel5);
103.      shortValues = ArrayUtils.add(shortValues, valuedatetupel6);
```

```
104.      shortValues = ArrayUtils.add(shortValues, valuedatetupel7);
105.      shortValues = ArrayUtils.add(shortValues, valuedatetupel8);
106.    }
107.
108.    /**
109.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
110.     */
111.    @Test
112.    void testBaseValue_name_values() {
113.      baseValue = new BaseValue(NAME_OF_TEST_BASE_VALUES, values);
114.      baseValue2 = new BaseValue(NAME_OF_TEST_BASE_VALUES, values);
115.
116.      assertEquals(baseValue, baseValue2,
117.          "Two instances with the same contents are not equal");
118.    }
119.
120.    /**
121.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
122.     */
123.    @Test
124.    void testBaseValue_nullName() {
125.      String nullName = null;
126.      String expectedMessage = "The given name must not be null";
127.
128.      Exception thrown = assertThrows(IllegalArgumentException.class,
129.          () -> new BaseValue(nullName, values),
130.          "Empty name not properly rejected");
131.      assertEquals(expectedMessage, thrown.getMessage(),
132.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
133.    }
134.
135.    /**
136.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
137.     */
138.    @Test
139.    void testBaseValue_nullValues() {
140.      ValueDateTupel[] nullValues = null;
141.      String expectedMessage = "The given values array must not be null";
142.
143.      Exception thrown = assertThrows(IllegalArgumentException.class,
144.          () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, nullValues),
145.          "Empty name not properly rejected");
146.      assertEquals(expectedMessage, thrown.getMessage(),
147.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
148.    }
149.
150.    /**
151.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
152.     */
153.    @Test
154.    void testBaseValue_emptyName() {
155.      String expectedMessage = "Name must not be an empty String";
156.
157.      Exception thrown = assertThrows(IllegalArgumentException.class,
```

```
158.          () -> new BaseValue(EMPTY_STRING, values),
159.          "Empty name not properly rejected");
160.      assertEquals(expectedMessage, thrown.getMessage(),
161.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
162.    }
163.
164.    /**
165.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
166.     */
167.    @Test
168.    void testBaseValue_emptyValues() {
169.      String expectedMessage = "Values must not be an empty array";
170.
171.      Exception thrown = assertThrows(IllegalArgumentException.class,
172.          () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, emptyValues),
173.          "Empty values not properly rejected");
174.      assertEquals(expectedMessage, thrown.getMessage(),
175.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
176.    }
177.
178.    /**
179.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
180.     */
181.    @Test
182.    void testBaseValue_duplicateDatesInValues() {
183.      String expectedMessage = "Given values are not properly sorted or "
184.          + "there are non-unique values.";
185.      values = ArrayUtils.add(values, valuedatetupel1);
186.
187.      Exception thrown = assertThrows(IllegalArgumentException.class,
188.          () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values),
189.          "Duplicate date/time values are not properly handled");
190.      assertEquals(expectedMessage, thrown.getMessage(),
191.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
192.    }
193.
194.    /**
195.     * Test method for {@link BaseValue#BaseValue(String, ValueDateTupel[])}.
196.     */
197.    @Test
198.    void testBaseValue_datesInIncorrectOrder() {
199.      String expectedMessage = "Given values are not properly sorted or "
200.          + "there are non-unique values.";
201.      values = ValueDateTupel.createEmptyArray();
202.      values = ArrayUtils.add(values, valuedatetupel1);
203.      values = ArrayUtils.add(values, valuedatetupel3);
204.      values = ArrayUtils.add(values, valuedatetupel2);
205.      values = ArrayUtils.add(values, valuedatetupel4);
206.
207.      Exception thrown = assertThrows(IllegalArgumentException.class,
208.          () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values),
209.          "Date/time values in incorrect order are not properly handled");
210.      assertEquals(expectedMessage, thrown.getMessage(),
211.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
```

```
212.     }
213.
214.     /**
215.      * Test method for
216.      * {@link BaseValue#BaseValue(String, ValueDateTupel[],
217.      * ValueDateTupel[])}.
218.      */
219.     @Test
220.     void testBaseValue_name_values_shortIndexValues() {
221.       baseValue = new BaseValue(NAME_OF_TEST_BASE_VALUES, values,
222.           shortValues);
223.       baseValue2 = new BaseValue(NAME_OF_TEST_BASE_VALUES, values,
224.           shortValues);
225.
226.       assertEquals(baseValue, baseValue2,
227.           "Two instances with the same contents are not equal");
228.     }
229.
230.     /**
231.      * Test method for
232.      * {@link BaseValue#BaseValue(String, ValueDateTupel[],
233.      * ValueDateTupel[])}.
234.      */
235.     @Test
236.     void testBaseValue_emptyName_values_shortIndexValues() {
237.       String expectedMessage = "Name must not be an empty String";
238.
239.       Exception thrown = assertThrows(IllegalArgumentException.class,
240.           () -> new BaseValue(EMPTY_STRING, values, shortValues),
241.           "Empty name not properly rejected");
242.       assertEquals(expectedMessage, thrown.getMessage(),
243.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
244.     }
245.
246.     /**
247.      * Test method for
248.      * {@link BaseValue#BaseValue(String, ValueDateTupel[],
249.      * ValueDateTupel[])}.
250.      */
251.     @Test
252.     void testBaseValue_name_emptyValues_shortIndexValues() {
253.       String expectedMessage = "Values must not be an empty array";
254.
255.       Exception thrown = assertThrows(IllegalArgumentException.class,
256.           () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, emptyValues,
257.               shortValues),
258.           "Empty values not properly rejected");
259.       assertEquals(expectedMessage, thrown.getMessage(),
260.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
261.     }
262.
263.     /**
264.      * Test method for
265.      * {@link BaseValue#BaseValue(String, ValueDateTupel[],
```

```
266.      * ValueDateTupel[])}.
267.      */
268.     @Test
269.     void testBaseValue_name_values_shortIndexValues_null() {
270.       String expectedMessage = "Given short index values do not meet the "
271.           + "specifications.";
272.       String expectedCauseMessage = "The given values array must not be "
273.           + "null";
274.
275.       Exception thrown = assertThrows(IllegalArgumentException.class,
276.           () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values, null),
277.           "Empty short index values not properly rejected");
278.       assertEquals(expectedMessage, thrown.getMessage(),
279.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
280.       assertEquals(expectedCauseMessage, thrown.getCause().getMessage(),
281.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
282.     }
283.
284.     /**
285.      * Test method for
286.      * {@link BaseValue#BaseValue(String, ValueDateTupel[],
287.      * ValueDateTupel[])}.
288.      */
289.     @Test
290.     void testBaseValue_name_values_emptyShortIndexValues() {
291.       String expectedMessage = "Given short index values do not meet the "
292.           + "specifications.";
293.       String expectedCauseMessage = "Values must not be an empty array";
294.
295.       Exception thrown = assertThrows(IllegalArgumentException.class,
296.           () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values, emptyValues),
297.           "Empty short index values not properly rejected");
298.       assertEquals(expectedMessage, thrown.getMessage(),
299.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
300.       assertEquals(expectedCauseMessage, thrown.getCause().getMessage(),
301.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
302.     }
303.
304.     /**
305.      * Test method for
306.      * {@link BaseValue#BaseValue(String, ValueDateTupel[],
307.      * ValueDateTupel[])}.
308.      */
309.     @Test
310.     void testBaseValue_name_values_duplicateShortIndexValues() {
311.       String expectedMessage = "Given short index values do not meet the "
312.           + "specifications.";
313.       String expectedCauseMessage = "Given values are not properly sorted or"
314.           + " there are non-unique values.";
315.       shortValues = ArrayUtils.add(shortValues, valuedatetupel1);
316.
317.       Exception thrown = assertThrows(IllegalArgumentException.class,
318.           () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values, shortValues),
319.           "Duplicate date/time values in short index values are not properly"
```

```
320.          + " handled");
321.      assertEquals(expectedMessage, thrown.getMessage(),
322.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
323.      assertEquals(expectedCauseMessage, thrown.getCause().getMessage(),
324.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
325.    }
326.
327.    /**
328.     * Test method for {@link BaseValue#validateInput(ValueDateTupel[])}.
329.     */
330.    @Test
331.    void testBaseValue_name_values_nanInShortIndexValues() {
332.      String expectedMessage = "Given short index values do not meet the "
333.          + "specifications.";
334.      String expectedCauseMessage = "Given values must not contain NaN.";
335.      shortValues = ArrayUtils.add(shortValues,
336.          new ValueDateTupel(localDateTimeJan05_22_00_00, Double.NaN));
337.
338.      Exception thrown = assertThrows(IllegalArgumentException.class,
339.          () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values, shortValues),
340.          "NaN values in short index values are not properly handled");
341.      assertEquals(expectedMessage, thrown.getMessage(),
342.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
343.      assertEquals(expectedCauseMessage, thrown.getCause().getMessage(),
344.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
345.    }
346.
347.    /**
348.     * Test method for {@link BaseValue#validateInput(ValueDateTupel[])}.
349.     */
350.    @Test
351.    void testBaseValue_name_values_nullInShortIndexValues() {
352.      String expectedMessage = "Given short index values do not meet the "
353.          + "specifications.";
354.      String expectedCauseMessage = "Given values must not contain null.";
355.      shortValues = ArrayUtils.add(shortValues, null);
356.
357.      Exception thrown = assertThrows(IllegalArgumentException.class,
358.          () -> new BaseValue(NAME_OF_TEST_BASE_VALUES, values, shortValues),
359.          "nulls in short index values are not properly handled");
360.      assertEquals(expectedMessage, thrown.getMessage(),
361.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
362.      assertEquals(expectedCauseMessage, thrown.getCause().getMessage(),
363.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
364.    }
365.
366.    /**
367.     * Test method for
368.     * {@link BaseValue#calculateShortIndexValues(ValueDateTupel[])}.
369.     */
370.    @Test
371.    void testCalculateShortIndexValues() {
372.      baseValue = new BaseValue(NAME_OF_TEST_BASE_VALUES, values);
373.
```

```
374.       ValueDateTupel[] actualValues = baseValue.getShortIndexValues();
375.
376.       assertArrayEquals(shortValues, actualValues,
377.           "The calculated short index values are not as expected");
378.     }
379.
380.     /**
381.      * Test method for
382.      * {@link BaseValue#calculateShortIndexValues(ValueDateTupel[])}.
383.      */
384.     @Test
385.     void testCalculateShortIndexValues_2valuesInBaseValue() {
386.       values = ValueDateTupel.createEmptyArray();
387.       values = ArrayUtils.add(values, valuedatetupel1);
388.       values = ArrayUtils.add(values, valuedatetupel2);
389.       shortValues = ValueDateTupel.createEmptyArray();
390.       shortValues = ArrayUtils.add(shortValues, valuedatetupel5);
391.       shortValues = ArrayUtils.add(shortValues, valuedatetupel6);
392.
393.       baseValue = new BaseValue(NAME_OF_TEST_BASE_VALUES, values);
394.       ValueDateTupel[] actualValues = baseValue.getShortIndexValues();
395.
396.       assertArrayEquals(shortValues, actualValues,
397.           "The calculated short index values are not as expected");
398.     }
399.   }
```

Komponente SubSystemTest

Listing 32: Komponente SubSystemTest

```
1.  package de.rumford.tradingsystem;
2.
3.  import static org.junit.jupiter.api.Assertions.assertEquals;
4.  import static org.junit.jupiter.api.Assertions.assertThrows;
5.
6.  import java.time.LocalDateTime;
7.
8.  import org.apache.commons.lang3.ArrayUtils;
9.  import org.junit.jupiter.api.BeforeAll;
10. import org.junit.jupiter.api.BeforeEach;
11. import org.junit.jupiter.api.Test;
12.
13. import de.rumford.tradingsystem.RuleTest.RealRule;
14. import de.rumford.tradingsystem.helper.BaseValueFactory;
15. import de.rumford.tradingsystem.helper.ValueDateTupel;
16.
17. /**
18.  * Test class for {@link SubSystem}.
19.  *
20.  * @author Max Rumford
21.  *
22.  */
23. class SubSystemTest {
24.   static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
25.       "Incorrect Exception message";
26.
27.   static final String BASE_VALUE_NAME = "Base value name";
28.   static BaseValue baseValue;
29.   static final double VARIATOR = 1;
30.   static final double BASE_SCALE = 10;
31.
32.   static final double CAPITAL = 10000000;
33.
34.   static LocalDateTime localDateTime2019Dec31220000;
35.   static LocalDateTime localDateTimeJan02220000;
36.   static LocalDateTime localDateTimeJan09220000;
37.   static LocalDateTime localDateTimeJan10220000;
38.   static LocalDateTime localDateTimeJan11220000;
39.   static LocalDateTime localDateTimeJan12220000;
40.   static LocalDateTime localDateTimeFeb04220000;
41.   static LocalDateTime localDateTimeFeb05220000;
42.   static LocalDateTime localDateTimeDec31220000;
43.
44.   static Rule r1;
45.   static Rule r2;
46.   static Rule r3;
47.   static Rule r4;
48.   static Rule[] rules;
49.
```

```
50.     static SubSystem subSystem;
51.
52.     @BeforeAll
53.     static void setUpBeforeClass() throws Exception {
54.       baseValue = BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME);
55.       localDateTime2019Dec31220000 = LocalDateTime.of(2019, 12, 31, 22, 0);
56.       localDateTimeJan02220000 = LocalDateTime.of(2020, 01, 2, 22, 0);
57.       localDateTimeJan09220000 = LocalDateTime.of(2020, 01, 9, 22, 0);
58.       localDateTimeJan10220000 = LocalDateTime.of(2020, 01, 10, 22, 0);
59.       localDateTimeJan11220000 = LocalDateTime.of(2020, 01, 11, 22, 0);
60.       localDateTimeJan12220000 = LocalDateTime.of(2020, 01, 12, 22, 0);
61.       localDateTimeFeb04220000 = LocalDateTime.of(2020, 02, 4, 22, 0);
62.       localDateTimeFeb05220000 = LocalDateTime.of(2020, 02, 5, 22, 0);
63.       localDateTimeDec31220000 = LocalDateTime.of(2020, 12, 31, 22, 0);
64.     }
65.
66.     @BeforeEach
67.     void setUp() throws Exception {
68.       r1 = RealRule.from(baseValue, null, localDateTimeJan10220000,
69.           localDateTimeJan12220000, BASE_SCALE, VARIATOR);
70.       r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
71.           localDateTimeJan12220000, BASE_SCALE, 2);
72.       r3 = RealRule.from(baseValue, null, localDateTimeJan10220000,
73.           localDateTimeJan12220000, BASE_SCALE, 3);
74.       r4 = RealRule.from(baseValue, null, localDateTimeJan10220000,
75.           localDateTimeJan12220000, BASE_SCALE, 4);
76.
77.       rules = null;
78.       rules = ArrayUtils.add(rules, r1);
79.       rules = ArrayUtils.add(rules, r2);
80.       rules = ArrayUtils.add(rules, r3);
81.       rules = ArrayUtils.add(rules, r4);
82.
83.       subSystem = new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE);
84.     }
85.
86.     /**
87.      * Test method for {@link SubSystem#SubSystem(BaseValue, Rule[], double)}
88.      */
89.     @Test
90.     void testSubSystem() {
91.       SubSystem subsys = new SubSystem(baseValue, rules, CAPITAL,
92.           BASE_SCALE);
93.       SubSystem subsys2 = new SubSystem(baseValue, rules, CAPITAL,
94.           BASE_SCALE);
95.       assertEquals(subsys, subsys2,
96.           "Equal Objects are not considered equal");
97.     }
98.
99.     /**
100.     * Test method for {@link SubSystem#evaluateRules(Rule[])}.
101.     */
102.    @Test
103.    void testEvaluateRules_identicalRules() {
```

```
104.        r1 = RealRule.from(baseValue, null, localDateTimeJan10220000,
105.            localDateTimeJan12220000, BASE_SCALE, VARIATOR);
106.        r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
107.            localDateTimeJan12220000, BASE_SCALE, VARIATOR);
108.        Rule[] rules = { r1, r2 };
109.
110.        String expectedMessage = "The given rules are not unique. Only unique "
111.            + "rules can be used.";
112.
113.        Exception thrown = assertThrows(IllegalArgumentException.class,
114.            () -> new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE),
115.            "Non-unique rules are not properly handled");
116.
117.        assertEquals(expectedMessage, thrown.getMessage(),
118.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
119.    }
120.
121.    /**
122.     * Test method for {@link SubSystem#evaluateRules(Rule[])}.
123.     */
124.    @Test
125.    void testEvaluateRules_differentStartOfReferenceWindow() {
126.      r2 = RealRule.from(baseValue, null, localDateTimeJan09220000,
127.          localDateTimeJan12220000, BASE_SCALE, 2);
128.      Rule[] rules = { r1, r2 };
129.      String expectedMessage = "All rules need to have the same reference "
130.          + "window but rules at position 0 and 1 differ.";
131.
132.      Exception thrown = assertThrows(IllegalArgumentException.class,
133.          () -> new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE),
134.          "Differing start of reference windows are not properly handled");
135.
136.      assertEquals(expectedMessage, thrown.getMessage(),
137.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
138.    }
139.
140.    /**
141.     * Test method for {@link SubSystem#evaluateRules(Rule[])}.
142.     */
143.    @Test
144.    void testEvaluateRules_differentEndOfReferenceWindow() {
145.      r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
146.          localDateTimeJan11220000, BASE_SCALE, 2);
147.      Rule[] rules = { r1, r2 };
148.      String expectedMessage = "All rules need to have the same reference "
149.          + "window but rules at position 0 and 1 differ.";
150.
151.      Exception thrown = assertThrows(IllegalArgumentException.class,
152.          () -> new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE),
153.          "Differing end of reference windows are not properly handled");
154.
155.      assertEquals(expectedMessage, thrown.getMessage(),
156.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
157.    }
```

```
158.
159.     /**
160.      * Test method for
161.      * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
162.      */
163.     @Test
164.     void testValidateInput_baseValueNull() {
165.       BaseValue nullBaseValue = null;
166.       String expectedMessage = "Base value must not be null";
167.
168.       Exception thrown = assertThrows(IllegalArgumentException.class,
169.           () -> new SubSystem(nullBaseValue, rules, CAPITAL, BASE_SCALE),
170.           "Null base value is not properly handled");
171.
172.       assertEquals(expectedMessage, thrown.getMessage(),
173.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
174.     }
175.
176.     /**
177.      * Test method for
178.      * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
179.      */
180.     @Test
181.     void testValidateInput_rulesNull() {
182.       Rule[] nullRules = null;
183.       String expectedMessage = "Rules must not be null";
184.
185.       Exception thrown = assertThrows(IllegalArgumentException.class,
186.           () -> new SubSystem(baseValue, nullRules, CAPITAL, BASE_SCALE),
187.           "Null rules are not properly handled");
188.
189.       assertEquals(expectedMessage, thrown.getMessage(),
190.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
191.     }
192.
193.     /**
194.      * Test method for
195.      * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
196.      */
197.     @Test
198.     void testValidateInput_rulesEmptyArray() {
199.       Rule[] emptyRulesArray = {};
200.       String expectedMessage = "Rules must not be an empty array";
201.
202.       Exception thrown = assertThrows(
203.           IllegalArgumentException.class, () -> new SubSystem(baseValue,
204.               emptyRulesArray, CAPITAL, BASE_SCALE),
205.           "Empty rules array is not properly handled");
206.
207.       assertEquals(expectedMessage, thrown.getMessage(),
208.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
209.     }
210.
211.     /**
```

```
212.      * Test method for
213.      * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
214.      */
215.     @Test
216.     void testValidateInput_rules_baseValueDoesntMatch_givenBaseValue() {
217.       BaseValue newBaseValue = BaseValueFactory
218.           .jan1Jan31calcShort(BASE_VALUE_NAME);
219.       Rule[] rules = {
220.           new RealRule(newBaseValue, null, localDateTimeJan10220000,
221.               localDateTimeJan12220000, BASE_SCALE, VARIATOR) };
222.       String expectedMessage = "The base value of all rules must be equal to"
223.           + " given base value but the rule at position 0 does not comply.";
224.
225.       Exception thrown = assertThrows(IllegalArgumentException.class,
226.           () -> new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE),
227.           "Empty rules array is not properly handled");
228.
229.       assertEquals(expectedMessage, thrown.getMessage(),
230.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
231.     }
232.
233.     /**
234.      * Test method for
235.      * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
236.      */
237.     @Test
238.     void testValidateInput_capitalNaN() {
239.       double nanCapital = Double.NaN;
240.       String expectedMessage = "Given capital does not meet specifications.";
241.       String expectedCause = "Value must not be Double.NaN";
242.
243.       Exception thrown = assertThrows(IllegalArgumentException.class,
244.           () -> new SubSystem(baseValue, rules, nanCapital, BASE_SCALE),
245.           "Capital of Double.NaN is not properly handled");
246.
247.       assertEquals(expectedMessage, thrown.getMessage(),
248.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
249.       assertEquals(expectedCause, thrown.getCause().getMessage(),
250.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
251.     }
252.
253.     /**
254.      * Test method for
255.      * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
256.      */
257.     @Test
258.     void testValidateInput_capitalZeroOrLess() {
259.       double zeroCapital = 0;
260.       double negativeCapital = -1;
261.       String expectedMessage = "Given capital does not meet specifications.";
262.       String expectedCause = "Value must be a positive decimal";
263.
264.       Exception thrown = assertThrows(IllegalArgumentException.class,
265.           () -> new SubSystem(baseValue, rules, negativeCapital, BASE_SCALE),
```

```
            "Negative capital value is not properly handled");

        Exception thrown2 = assertThrows(IllegalArgumentException.class,
            () -> new SubSystem(baseValue, rules, zeroCapital, BASE_SCALE),
            "Capital of 0 is not properly handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
        assertEquals(expectedCause, thrown.getCause().getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
        assertEquals(expectedMessage, thrown2.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
        assertEquals(expectedCause, thrown2.getCause().getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for
     * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
     */
    @Test
    void testValidateInput_baseScaleNaN() {
        double baseScaleNaN = Double.NaN;
        String expectedMessage = "Given base scale does not meet "
            + "specifications.";
        String expectedCause = "Value must not be Double.NaN";

        Exception thrown = assertThrows(IllegalArgumentException.class,
            () -> new SubSystem(baseValue, rules, CAPITAL, baseScaleNaN),
            "Base scale of NaN is not properly handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
        assertEquals(expectedCause, thrown.getCause().getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for
     * {@link SubSystem#validateInput(BaseValue, Rule[], double)}.
     */
    @Test
    void testValidateInput_baseScaleZeroOrLess() {
        double baseScaleZero = 0;
        double baseScaleSubZero = -1;
        String expectedMessage = "Given base scale does not meet "
            + "specifications.";
        String expectedCause = "Value must be a positive decimal";

        Exception thrown = assertThrows(IllegalArgumentException.class,
            () -> new SubSystem(baseValue, rules, CAPITAL, baseScaleZero),
            "Base scale of zero is not properly handled");
        Exception thrown2 = assertThrows(IllegalArgumentException.class,
            () -> new SubSystem(baseValue, rules, CAPITAL, baseScaleSubZero),
```

```
            "Base scale sub zero is not properly handled");

      assertEquals(expectedMessage, thrown.getMessage(),
          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
      assertEquals(expectedCause, thrown.getCause().getMessage(),
          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);

      assertEquals(expectedMessage, thrown2.getMessage(),
          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
      assertEquals(expectedCause, thrown2.getCause().getMessage(),
          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for {@link SubSystem#calculateCombinedForecasts()}.
     */
    @Test
    void testCalculateCombinedForecasts() {
      double expectedValue1 = 13.398963140010043; // Excel: 13.3988598882083,
      // diff. approx
      // 0.0007706%
      double expectedValue2 = 20; // Excel: 20

      assertEquals(expectedValue1,
          subSystem.getCombinedForecasts()[0].getValue(),
          "Combined forecasts are not correctly calculated");
      assertEquals(expectedValue2,
          subSystem
              .getCombinedForecasts()[subSystem.getCombinedForecasts().length
                  - 1].getValue(),
          "Combined forecasts are not correctly calculated");
    }

    /**
     * Test method for
     * {@link SubSystem#backtest(LocalDateTime startOfTestWindow,
     * LocalDateTime endOfTestWindow)}.
     */
    @Test
    void testBacktest() {
      double expectedValue = 1831472.7037588374; // Excel: 1,831,582.23,
                                                 // diff.
      // approx 0.00598%

      assertEquals(expectedValue,
          subSystem.backtest(localDateTimeJan10220000,
              localDateTimeFeb05220000),
          "Backtest performance is not correctly calculated");
    }

    /**
     * Test method for
     * {@link SubSystem#backtest(LocalDateTime startOfTestWindow,
     * LocalDateTime endOfTestWindow)}.
```

```
374.      */
375.     @Test
376.     void testBacktest_positiveAndNegativeForecasts() {
377.       VolatilityDifference volDif = new VolatilityDifference(baseValue, null,
378.           localDateTimeJan10220000, localDateTimeJan12220000, 4, BASE_SCALE);
379.       Rule[] rules = { volDif };
380.       subSystem = new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE);
381.
382.       double expectedValue = 16815027.90331543; // Excel: 16815027.1988897
383.
384.       assertEquals(expectedValue,
385.           subSystem.backtest(localDateTimeJan10220000,
386.               localDateTimeFeb04220000),
387.           "Backtest performance is not correctly calculated");
388.     }
389.
390.     /**
391.      * Test method for
392.      * {@link SubSystem#backtest(LocalDateTime startOfTestWindow,
393.      * LocalDateTime endOfTestWindow)}.
394.      */
395.     @Test
396.     void testBacktest_startOfTestWindow_null() {
397.       String expectedMessage = "The given test window does not meet "
398.           + "specifications.";
399.       String expectedCause = "Start of time window value must not be null";
400.
401.       Exception thrown = assertThrows(IllegalArgumentException.class,
402.           () -> subSystem.backtest(null, localDateTimeFeb05220000),
403.           "Start of test window of null is not properly handled");
404.
405.       assertEquals(expectedMessage, thrown.getMessage(),
406.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
407.       assertEquals(expectedCause, thrown.getCause().getMessage(),
408.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
409.     }
410.
411.     /**
412.      * Test method for
413.      * {@link SubSystem#backtest(LocalDateTime startOfTestWindow,
414.      * LocalDateTime endOfTestWindow)}.
415.      */
416.     @Test
417.     void testBacktest_endOfTestWindow_null() {
418.       String expectedMessage = "The given test window does not meet "
419.           + "specifications.";
420.       String expectedCause = "End of time window value must not be null";
421.
422.       Exception thrown = assertThrows(IllegalArgumentException.class,
423.           () -> subSystem.backtest(localDateTimeJan10220000, null),
424.           "End of test window of null is not properly handled");
425.
426.       assertEquals(expectedMessage, thrown.getMessage(),
427.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
```

```
428.       assertEquals(expectedCause, thrown.getCause().getMessage(),
429.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
430.     }
431.
432.     /**
433.      * Test method for
434.      * {@link SubSystem#backtest(LocalDateTime startOfTestWindow,
435.      * LocalDateTime endOfTestWindow)}.
436.      */
437.     @Test
438.     void testBacktest_endOfTestWindow_not_after_startOfTestWindow() {
439.       String expectedMessage = "The given test window does not meet "
440.           + "specifications.";
441.       String expectedCause = "End of time window value must be after start "
442.           + "of time window value";
443.
444.       Exception thrown = assertThrows(IllegalArgumentException.class,
445.           () -> subSystem.backtest(localDateTimeJan10220000,
446.               localDateTimeJan10220000),
447.           "End of test window not after start of test window is not properly"
448.           + " handled");
449.
450.       assertEquals(expectedMessage, thrown.getMessage(),
451.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
452.       assertEquals(expectedCause, thrown.getCause().getMessage(),
453.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
454.     }
455.
456.     /**
457.      * Test method for
458.      * {@link SubSystem#calculatePerformanceValues(LocalDateTime,
459.      * LocalDateTime)}.
460.      */
461.     @Test
462.     void testCalculatePerformanceValues_static() {
463.       double expectedValue = 16454612.6646818; // Excel: 16454586.0867138,
464.       // diff. approx 0.000162%
465.
466.       ValueDateTupel[] performanceValues = subSystem
467.           .calculatePerformanceValues(localDateTimeJan10220000,
468.               localDateTimeFeb05220000);
469.       assertEquals(expectedValue,
470.           performanceValues[performanceValues.length - 2].getValue(),
471.           "Performance values are not properly calculated");
472.     }
473.
474.     /**
475.      * Test method for
476.      * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
477.      * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
478.      * double, double)}.
479.      */
480.     @Test
481.     void testCalculatePerformanceValues_instance() {
```

```
482.      double expectedValue = 16454612.6646818; // Excel: 16454586.0867138,
483.      // diff. approx 0.000162%
484.
485.      ValueDateTupel[] performanceValues = SubSystem
486.          .calculatePerformanceValues(subSystem.getBaseValue(),
487.              localDateTimeJan10220000, localDateTimeFeb05220000,
488.              subSystem.getCombinedForecasts(), subSystem.getBaseScale(),
489.              subSystem.getCapital());
490.      assertEquals(expectedValue,
491.          performanceValues[performanceValues.length - 2].getValue(),
492.          "Performance values are not properly calculated");
493.    }
494.
495.    /**
496.     * Test method for
497.     * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
498.     * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
499.     * double, double)}.
500.     */
501.    @Test
502.    void testCalculatePerformanceValues_positiveAndNegativeForecasts() {
503.      VolatilityDifference volDif4 = new VolatilityDifference(baseValue,
504.          null, localDateTimeJan10220000, localDateTimeJan12220000, 4,
505.          BASE_SCALE);
506.      VolatilityDifference volDif8 = new VolatilityDifference(baseValue,
507.          null, localDateTimeJan10220000, localDateTimeJan12220000, 8,
508.          BASE_SCALE);
509.
510.      Rule[] rules = { volDif4, volDif8 };
511.      subSystem = new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE);
512.
513.      double expectedValue = 96201.5377744669; // Excel: 96201.5377744669
514.
515.      ValueDateTupel[] performanceValues = SubSystem
516.          .calculatePerformanceValues(subSystem.getBaseValue(),
517.              localDateTimeJan10220000, localDateTimeFeb05220000,
518.              subSystem.getCombinedForecasts(), subSystem.getBaseScale(),
519.              subSystem.getCapital());
520.      assertEquals(expectedValue,
521.          performanceValues[performanceValues.length - 1].getValue(),
522.          "Performance values are not properly calculated");
523.    }
524.
525.    /**
526.     * Test method for
527.     * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
528.     * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
529.     * double, double)}.
530.     */
531.    @Test
532.    void testCalculatePerformanceValues_positiveNegativeAndZeroForecasts() {
533.      VolatilityDifference volDif4 = new VolatilityDifference(baseValue,
534.          null, localDateTimeJan10220000, localDateTimeJan12220000, 4,
535.          BASE_SCALE);
```

```
536.        RealRule rr = RealRule.from(baseValue, null, localDateTimeJan10220000,
537.            localDateTimeJan12220000, BASE_SCALE, VARIATOR);
538.
539.        Rule[] rules = { volDif4, rr };
540.        subSystem = new SubSystem(baseValue, rules, CAPITAL, BASE_SCALE);
541.
542.        double expectedValue = 1271620.1875697833; // Excel: 1271620.18756978
543.
544.        ValueDateTupel[] performanceValues = SubSystem
545.            .calculatePerformanceValues(subSystem.getBaseValue(),
546.                localDateTimeJan10220000, localDateTimeFeb05220000,
547.                subSystem.getCombinedForecasts(), subSystem.getBaseScale(),
548.                subSystem.getCapital());
549.        assertEquals(expectedValue,
550.            performanceValues[performanceValues.length - 1].getValue(),
551.            "Performance values are not properly calculated");
552.    }
553.
554.    /**
555.     * Test method for
556.     * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
557.     * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
558.     * double, double)}.
559.     */
560.    @Test
561.    void testCalculatePerformanceValues_startOfTestWindowNull() {
562.        String expectedMessage = "The given test window does not meet "
563.            + "specifications.";
564.        String expectedCause = "Start of time window value must not be null";
565.
566.        Exception thrown = assertThrows(IllegalArgumentException.class,
567.            () -> SubSystem.calculatePerformanceValues(
568.                subSystem.getBaseValue(), null, localDateTimeFeb05220000,
569.                subSystem.getCombinedForecasts(), subSystem.getBaseScale(),
570.                subSystem.getCapital()),
571.            "Invalid start of test window is not properly handled");
572.
573.        assertEquals(expectedMessage, thrown.getMessage(),
574.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
575.        assertEquals(expectedCause, thrown.getCause().getMessage(),
576.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
577.    }
578.
579.    /**
580.     * Test method for
581.     * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
582.     * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
583.     * double, double)}.
584.     */
585.    @Test
586.    void testCalculatePerformanceValues_endOfTestWindowNull() {
587.        String expectedMessage = "The given test window does not meet "
588.            + "specifications.";
589.        String expectedCause = "End of time window value must not be null";
```

```
590.
591.       Exception thrown = assertThrows(IllegalArgumentException.class,
592.           () -> SubSystem.calculatePerformanceValues(
593.               subSystem.getBaseValue(), localDateTimeJan10220000, null,
594.               subSystem.getCombinedForecasts(), subSystem.getBaseScale(),
595.               subSystem.getCapital()),
596.           "Invalid end of test window is not properly handled");
597.
598.       assertEquals(expectedMessage, thrown.getMessage(),
599.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
600.       assertEquals(expectedCause, thrown.getCause().getMessage(),
601.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
602.     }
603.
604.     /**
605.      * Test method for
606.      * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
607.      * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
608.      * double, double)}.
609.      */
610.     @Test
611.     void testCalculatePerformanceValues_endOfTest_not_after_startOfTest() {
612.       String expectedMessage = "The given test window does not meet "
613.           + "specifications.";
614.       String expectedCause = "End of time window value must be after start "
615.           + "of time window value";
616.
617.       Exception thrown = assertThrows(IllegalArgumentException.class,
618.           () -> SubSystem.calculatePerformanceValues(
619.               subSystem.getBaseValue(), localDateTimeJan10220000,
620.               localDateTimeJan10220000, subSystem.getCombinedForecasts(),
621.               subSystem.getBaseScale(), subSystem.getCapital()),
622.           "End of test window not after start of test window is not properly"
623.           + " handled");
624.
625.       assertEquals(expectedMessage, thrown.getMessage(),
626.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
627.       assertEquals(expectedCause, thrown.getCause().getMessage(),
628.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
629.     }
630.
631.     /**
632.      * Test method for
633.      * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
634.      * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
635.      * double, double)}.
636.      */
637.     @Test
638.     void testCalculatePerformanceValues_startOfTest_not_in_baseValue() {
639.       String expectedMessage = "Given base value and test window do not "
640.           + "fit.";
641.       String expectedCause = "Given values do not include given start value "
642.           + "for time window";
643.
```

```
644.      Exception thrown = assertThrows(IllegalArgumentException.class,
645.          () -> SubSystem.calculatePerformanceValues(
646.              subSystem.getBaseValue(), localDateTime2019Dec31220000,
647.              localDateTimeFeb05220000, subSystem.getCombinedForecasts(),
648.              subSystem.getBaseScale(), subSystem.getCapital()),
649.          "Start of test window not in base values is not properly handled");
650.
651.      assertEquals(expectedMessage, thrown.getMessage(),
652.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
653.      assertEquals(expectedCause, thrown.getCause().getMessage(),
654.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
655.    }
656.
657.    /**
658.     * Test method for
659.     * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
660.     * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
661.     * double, double)}.
662.     */
663.    @Test
664.    void testCalculatePerformanceValues_endOfTestWindow_not_in_baseValue() {
665.      String expectedMessage = "Given base value and test window do not "
666.          + "fit.";
667.      String expectedCause = "Given values do not include given end value "
668.          + "for time window";
669.
670.      Exception thrown = assertThrows(IllegalArgumentException.class,
671.          () -> SubSystem.calculatePerformanceValues(
672.              subSystem.getBaseValue(), localDateTimeJan10220000,
673.              localDateTimeDec31220000, subSystem.getCombinedForecasts(),
674.              subSystem.getBaseScale(), subSystem.getCapital()),
675.          "End of test window not in base values is not properly handled");
676.
677.      assertEquals(expectedMessage, thrown.getMessage(),
678.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
679.      assertEquals(expectedCause, thrown.getCause().getMessage(),
680.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
681.    }
682.
683.    /**
684.     * Test method for
685.     * {@link SubSystem#calculatePerformanceValues(BaseValue, LocalDateTime,
686.     * LocalDateTime, de.rumford.tradingsystem.helper.ValueDateTupel[],
687.     * double, double)}.
688.     */
689.    @Test
690.    void testCalculatePerformanceValues_startOfTest_not_in_forecasts() {
691.      String expectedMessage = "Given forecasts and test window do not fit.";
692.      String expectedCause = "Given values do not include given start value "
693.          + "for time window";
694.
695.      Exception thrown = assertThrows(IllegalArgumentException.class,
696.          () -> SubSystem.calculatePerformanceValues(
697.              subSystem.getBaseValue(), localDateTimeJan09220000,
```

```
698.                 localDateTimeFeb05220000, subSystem.getCombinedForecasts(),
699.                 subSystem.getBaseScale(), subSystem.getCapital()),
700.             "Start of test window not in forecasts is not properly handled");
701.
702.         assertEquals(expectedMessage, thrown.getMessage(),
703.             MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
704.         assertEquals(expectedCause, thrown.getCause().getMessage(),
705.             MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
706.     }
707. }
```

Komponente DiversificationMultiplierTest

Listing 33: Komponente DiversificationMultiplierTest

```
1.  package de.rumford.tradingsystem;
2.
3.  import static org.junit.jupiter.api.Assertions.assertEquals;
4.  import static org.junit.jupiter.api.Assertions.assertTrue;
5.
6.  import java.time.LocalDate;
7.  import java.time.LocalDateTime;
8.  import java.time.LocalTime;
9.
10. import org.junit.jupiter.api.BeforeAll;
11. import org.junit.jupiter.api.BeforeEach;
12. import org.junit.jupiter.api.Test;
13.
14. import de.rumford.tradingsystem.RuleTest.RealRule;
15. import de.rumford.tradingsystem.helper.BaseValueFactory;
16.
17. /**
18.  * Test class for {@link DiversificationMultiplier}.
19.  *
20.  * @author Max Rumford
21.  *
22.  */
23. class DiversificationMultiplierTest {
24.
25.   static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
26.       "Incorrect Exception message";
27.
28.   static DiversificationMultiplier divMulti;
29.
30.   static RealRule ss1;
31.   static RealRule ss2;
32.   static RealRule ss3;
33.   static RealRule s1;
34.   static RealRule s2;
35.   static RealRule s3;
36.   static RealRule s4;
37.   static RealRule s5;
38.   static RealRule t1;
39.   static RealRule t2;
40.
41.   static RealRule realRule;
42.   static Rule[] variations;
43.   static BaseValue baseValue;
44.   static double variator;
45.
46.   static final String BASE_VALUE_NAME = "Base value name";
47.   static final int BASE_SCALE = 10;
48.
49.   static LocalDateTime localDateTimeJan02220000;
```

```
50.     static LocalDateTime localDateTimeJan04220000;
51.     static LocalDateTime localDateTimeJan05220000;
52.     static LocalDateTime localDateTimeJan07220000;
53.     static LocalDateTime localDateTimeJan10220000;
54.     static LocalDateTime localDateTimeJan12220000;
55.     static LocalDateTime localDateTimeJan13220000;
56.     static LocalDateTime localDateTimeJan15220000;
57.     static LocalDateTime localDateTimeJan16220000;
58.     static LocalDateTime localDateTimeJan18220000;
59.
60.     @BeforeAll
61.     static void setUpBeforeClass() {
62.       baseValue = BaseValueFactory.jan1Jan31calcShort(BASE_VALUE_NAME);
63.
64.       localDateTimeJan02220000 = LocalDateTime.of(LocalDate.of(2020, 1, 2),
65.           LocalTime.of(22, 0));
66.       localDateTimeJan04220000 = LocalDateTime.of(LocalDate.of(2020, 1, 4),
67.           LocalTime.of(22, 0));
68.       localDateTimeJan05220000 = LocalDateTime.of(LocalDate.of(2020, 1, 5),
69.           LocalTime.of(22, 0));
70.       localDateTimeJan07220000 = LocalDateTime.of(LocalDate.of(2020, 1, 7),
71.           LocalTime.of(22, 0));
72.       localDateTimeJan10220000 = LocalDateTime.of(LocalDate.of(2020, 1, 10),
73.           LocalTime.of(22, 0));
74.       localDateTimeJan12220000 = LocalDateTime.of(LocalDate.of(2020, 1, 12),
75.           LocalTime.of(22, 0));
76.       localDateTimeJan13220000 = LocalDateTime.of(LocalDate.of(2020, 1, 13),
77.           LocalTime.of(22, 0));
78.       localDateTimeJan15220000 = LocalDateTime.of(LocalDate.of(2020, 1, 15),
79.           LocalTime.of(22, 0));
80.       localDateTimeJan16220000 = LocalDateTime.of(LocalDate.of(2020, 1, 16),
81.           LocalTime.of(22, 0));
82.       localDateTimeJan18220000 = LocalDateTime.of(LocalDate.of(2020, 1, 18),
83.           LocalTime.of(22, 0));
84.     }
85.
86.     @BeforeEach
87.     void setUp() {
88.       variator = 1;
89.       ss1 = RealRule.from(baseValue, null, localDateTimeJan10220000,
90.           localDateTimeJan12220000, BASE_SCALE, 1);
91.       ss2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
92.           localDateTimeJan12220000, BASE_SCALE, .5);
93.       ss3 = RealRule.from(baseValue, null, localDateTimeJan10220000,
94.           localDateTimeJan12220000, BASE_SCALE, -1.07);
95.       Rule[] s1variations = { ss1, ss2, ss3 };
96.       s1 = RealRule.from(baseValue, s1variations, localDateTimeJan10220000,
97.           localDateTimeJan12220000, BASE_SCALE, -1);
98.       s2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
99.           localDateTimeJan12220000, BASE_SCALE, -2.32);
100.      s3 = RealRule.from(baseValue, null, localDateTimeJan10220000,
101.          localDateTimeJan12220000, BASE_SCALE, -14);
102.      Rule[] t1variations = { s1, s2, s3 };
103.      t1 = RealRule.from(baseValue, t1variations, localDateTimeJan10220000,
```

```
            localDateTimeJan12220000, BASE_SCALE, .5);

        s4 = RealRule.from(baseValue, null, localDateTimeJan10220000,
            localDateTimeJan12220000, BASE_SCALE, 0.8);
        s5 = RealRule.from(baseValue, null, localDateTimeJan10220000,
            localDateTimeJan12220000, BASE_SCALE, -4.67);
        Rule[] t2variations = { s4, s5 };
        t2 = RealRule.from(baseValue, t2variations, localDateTimeJan10220000,
            localDateTimeJan12220000, BASE_SCALE, -10);
        Rule[] realRuleVariations = { t1, t2 };

        realRule = RealRule.from(baseValue, realRuleVariations,
            localDateTimeJan10220000, localDateTimeJan12220000, BASE_SCALE,
            variator);
        variations = realRule.getVariations();

        divMulti = new DiversificationMultiplier(variations);
    }

    /**
     * Test method for
     * {@link DiversificationMultiplier#DiversificationMultiplier(double[],
     * double[][])}.
     */
    @Test
    void testDiversificationMultiplier() {
        assertTrue(divMulti instanceof DiversificationMultiplier,
            "Instance of DiversificationMultiplier not recognized");
    }

    /**
     * Test method for
     * {@link DiversificationMultiplier
     * #calculateDiversificiationMultiplierValue()}.
     */
    @Test
    void testCalculateDiversificiationMultiplierValue() {
        double expectedDiversificationMultiplier = 3.862140866820605; // Excel:
        // 3.8621408668206

        double actualDiversificationMultiplier = divMulti.getValue();

        assertEquals(expectedDiversificationMultiplier,
            actualDiversificationMultiplier,
            "Diversification multiplier value is not correctly calculated");
    }

    /**
     * Test method for
     * {@link DiversificationMultiplier
     * #calculateDiversificiationMultiplierValue()}.
     */
    @Test
    void testCalculateDiversificiationMultiplierValue_rulesHaveNoVars() {
```

```
158.        double expectedValue = 1.000109838860305; // Excel: 1.00010213205928
159.        ss1 = RealRule.from(baseValue, null, localDateTimeJan10220000,
160.            localDateTimeJan12220000, BASE_SCALE, 1);
161.        ss2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
162.            localDateTimeJan12220000, BASE_SCALE, 2);
163.        ss3 = RealRule.from(baseValue, null, localDateTimeJan10220000,
164.            localDateTimeJan12220000, BASE_SCALE, 3);
165.        RealRule ss4 = RealRule.from(baseValue, null, localDateTimeJan10220000,
166.            localDateTimeJan12220000, BASE_SCALE, 4);
167.        Rule[] rules = { ss1, ss2, ss3, ss4 };
168.
169.        divMulti = new DiversificationMultiplier(rules);
170.
171.        assertEquals(expectedValue, divMulti.getValue(),
172.            "Diversification Multiplier is not correctly calculated when rules"
173.            + " have no variations");
174.    }
175.  }
```

Komponente ValueDateTupelTest

Listing 34: Komponente ValueDateTupelTest

```
1.    package de.rumford.tradingsystem.helper;
2.
3.    import static org.junit.jupiter.api.Assertions.*;
4.
5.    import java.time.LocalDateTime;
6.
7.    import org.junit.jupiter.api.BeforeAll;
8.    import org.junit.jupiter.api.BeforeEach;
9.    import org.junit.jupiter.api.Test;
10.
11.   /**
12.    * Test class for {@link ValueDateTupel}.
13.    *
14.    * @author Max Rumford
15.    *
16.    */
17.   class ValueDateTupelTest {
18.
19.     static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
20.         "Incorrect Exception message";
21.     static final String MESSAGE_ARRAY_MUST_NOT_BE_NULL =
22.         "Given array must not be null";
23.     static final String MESSAGE_VALUE_MUST_NOT_BE_NULL =
24.         "Given value must not be null";
25.
26.     static double value1;
27.     static double value2;
28.     static double value3;
29.     static double value4;
30.     static double value5;
31.     static LocalDateTime date_20200101;
32.     static LocalDateTime date_20200102;
33.     static LocalDateTime date_20200103;
34.     static LocalDateTime date_20200104;
35.     static LocalDateTime date_20200105;
36.     static ValueDateTupel valueDateTupel1;
37.     static ValueDateTupel valueDateTupel1_;
38.     static ValueDateTupel valueDateTupel2;
39.     static ValueDateTupel valueDateTupel3;
40.     static ValueDateTupel valueDateTupel4;
41.     static ValueDateTupel valueDateTupel5;
42.
43.     @BeforeAll
44.     static void setUpBeforeClass() throws Exception {
45.       date_20200101 = LocalDateTime.of(2020, 1, 1, 0, 0);
46.       date_20200102 = LocalDateTime.of(2020, 1, 2, 0, 0);
47.       date_20200103 = LocalDateTime.of(2020, 1, 3, 0, 0);
48.       date_20200104 = LocalDateTime.of(2020, 1, 4, 0, 0);
49.       date_20200105 = LocalDateTime.of(2020, 1, 5, 0, 0);
```

```
50.         value1 = 100d;
51.         value2 = 200d;
52.         value3 = 300d;
53.         value4 = 400d;
54.         value5 = 500d;
55.
56.         valueDateTupel1 = new ValueDateTupel(date_20200101, value1);
57.         valueDateTupel2 = new ValueDateTupel(date_20200102, value2);
58.         valueDateTupel3 = new ValueDateTupel(date_20200103, value3);
59.         valueDateTupel4 = new ValueDateTupel(date_20200104, value4);
60.         valueDateTupel5 = new ValueDateTupel(date_20200105, value5);
61.       }
62.
63.       @BeforeEach
64.       void setUp() throws Exception {
65.       }
66.
67.       /**
68.        * Test method for {@link ValueDateTupel#equals(Object)}.
69.        */
70.       @Test
71.       void testEqualsObject() {
72.         valueDateTupel1_ = new ValueDateTupel(date_20200101, value1);
73.
74.         assertEquals(valueDateTupel1, valueDateTupel1_,
75.             "Two equal instances of ValueDateTupel are not equal");
76.       }
77.
78.       /**
79.        * Test method for {@link ValueDateTupel#createEmptyArray()}.
80.        */
81.       @Test
82.       void testCreateEmptyArray() {
83.         ValueDateTupel[] expectedArray = new ValueDateTupel[0];
84.
85.         ValueDateTupel[] actualArray = ValueDateTupel.createEmptyArray();
86.
87.         assertArrayEquals(expectedArray, actualArray,
88.             "A non empty array is created");
89.       }
90.
91.       /**
92.        * Test method for {@link ValueDateTupel#createEmptyArray(int)}.
93.        */
94.       @Test
95.       void testCreateEmptyArrayInt() {
96.         ValueDateTupel[] expectedArray = new ValueDateTupel[2];
97.
98.         ValueDateTupel[] actualArray = ValueDateTupel.createEmptyArray(2);
99.
100.        assertArrayEquals(expectedArray, actualArray,
101.            "The created array is not as expected");
102.      }
103.
```

```
104.    /**
105.     * Test method for
106.     * {@link ValueDateTupel#isSortedAscending(ValueDateTupel[])}.
107.     */
108.    @Test
109.    void testIsSortedAscending() {
110.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
111.          valueDateTupel2, valueDateTupel3 };
112.
113.      assertTrue(ValueDateTupel.isSortedAscending(valueDateTupelArray),
114.          "The given array is falsly marked as not in ascending order");
115.    }
116.
117.    /**
118.     * Test method for
119.     * {@link ValueDateTupel#isSortedAscending(ValueDateTupel[])}.
120.     */
121.    @Test
122.    void testIsSortedAscending_notInOrder() {
123.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
124.          valueDateTupel3, valueDateTupel2 };
125.
126.      assertFalse(ValueDateTupel.isSortedAscending(valueDateTupelArray),
127.          "The given array is falsly marked as in ascending order");
128.    }
129.
130.    /**
131.     * Test method for
132.     * {@link ValueDateTupel#isSortedAscending(ValueDateTupel[])}.
133.     */
134.    @Test
135.    void testIsSortedAscending_twoEqualDates() {
136.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
137.          valueDateTupel2, valueDateTupel2 };
138.
139.      assertFalse(ValueDateTupel.isSortedAscending(valueDateTupelArray),
140.          "The given array is falsly marked as in ascending order");
141.    }
142.
143.    /**
144.     * Test method for
145.     * {@link ValueDateTupel#isSortedAscending(ValueDateTupel[])}.
146.     */
147.    @Test
148.    void testIsSortedAscending_arrayNull() {
149.      ValueDateTupel[] valueDateTupelArray = null;
150.      String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
151.
152.      Exception thrown = assertThrows(IllegalArgumentException.class,
153.          () -> ValueDateTupel.isSortedAscending(valueDateTupelArray),
154.          "A null array is not properly handled");
155.      assertEquals(expectedMessage, thrown.getMessage(),
156.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
157.    }
```

```
158.
159.     /**
160.      * Test method for
161.      * {@link ValueDateTupel#isSortedAscending(ValueDateTupel[])}.
162.      */
163.     @Test
164.     void testIsSortedAscending_arrayContainsNull() {
165.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1, null,
166.           valueDateTupel2 };
167.       String expectedMessage = "The given array must not contain any nulls";
168.
169.       Exception thrown = assertThrows(IllegalArgumentException.class,
170.           () -> ValueDateTupel.isSortedAscending(valueDateTupelArray),
171.           "An array containing null is not properly handled");
172.       assertEquals(expectedMessage, thrown.getMessage(),
173.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
174.     }
175.
176.     /**
177.      * Test method for
178.      * {@link ValueDateTupel#isSortedDescending(ValueDateTupel[])}.
179.      */
180.     @Test
181.     void testIsSortedDescending() {
182.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel3,
183.           valueDateTupel2, valueDateTupel1 };
184.
185.       assertTrue(ValueDateTupel.isSortedDescending(valueDateTupelArray),
186.           "The given array is falsly marked as not in descending order");
187.     }
188.
189.     /**
190.      * Test method for
191.      * {@link ValueDateTupel#isSortedDescending(ValueDateTupel[])}.
192.      */
193.     @Test
194.     void testIsSortedDescending_notInOrder() {
195.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel3,
196.           valueDateTupel1, valueDateTupel2 };
197.
198.       assertFalse(ValueDateTupel.isSortedDescending(valueDateTupelArray),
199.           "The given array is falsly marked as in descending order");
200.     }
201.
202.     /**
203.      * Test method for
204.      * {@link ValueDateTupel#isSortedDescending(ValueDateTupel[])}.
205.      */
206.     @Test
207.     void testIsSortedDescending_twoEqualDates() {
208.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel2,
209.           valueDateTupel2, valueDateTupel1 };
210.
211.       assertFalse(ValueDateTupel.isSortedDescending(valueDateTupelArray),
```

```
212.            "The given array is falsly marked as in descending order");
213.    }
214.
215.    /**
216.     * Test method for
217.     * {@link ValueDateTupel#isSortedDescending(ValueDateTupel[])}.
218.     */
219.    @Test
220.    void testIsSortedDescending_arrayNull() {
221.      ValueDateTupel[] valueDateTupelArray = null;
222.      String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
223.
224.      Exception thrown = assertThrows(IllegalArgumentException.class,
225.          () -> ValueDateTupel.isSortedDescending(valueDateTupelArray),
226.          "A null array is not properly handled");
227.      assertEquals(expectedMessage, thrown.getMessage(),
228.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
229.    }
230.
231.    /**
232.     * Test method for
233.     * {@link ValueDateTupel#isSortedDescending(ValueDateTupel[])}.
234.     */
235.    @Test
236.    void testIsSortedDescending_arrayContainsNull() {
237.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel2, null,
238.          valueDateTupel1 };
239.      String expectedMessage = "The given array must not contain any null "
240.          + "LocalDateTime";
241.
242.      Exception thrown = assertThrows(IllegalArgumentException.class,
243.          () -> ValueDateTupel.isSortedDescending(valueDateTupelArray),
244.          "An array containing null is not properly handled");
245.      assertEquals(expectedMessage, thrown.getMessage(),
246.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
247.    }
248.
249.    /**
250.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
251.     */
252.    @Test
253.    void testAlignDates() {
254.      ValueDateTupel vdtCalculated20200102_200 = new ValueDateTupel(
255.          date_20200102, 200d);
256.      ValueDateTupel vdtCalculated20200104_400 = new ValueDateTupel(
257.          date_20200104, 400d);
258.      ValueDateTupel vdtCalculated20200101_200 = new ValueDateTupel(
259.          date_20200101, 200d);
260.      ValueDateTupel vdtCalculated20200105_400 = new ValueDateTupel(
261.          date_20200105, 400d);
262.      ValueDateTupel vdtCalculated20200102_250 = new ValueDateTupel(
263.          date_20200102, 250d);
264.      ValueDateTupel vdtCalculated20200103_250 = new ValueDateTupel(
265.          date_20200103, 250d);
```

```
266.        ValueDateTupel vdtCalculated20200101_300 = new ValueDateTupel(
267.            date_20200101, 300d);
268.        ValueDateTupel vdtCalculated20200102_300 = new ValueDateTupel(
269.            date_20200102, 300d);
270.        ValueDateTupel vdtCalculated20200104_300 = new ValueDateTupel(
271.            date_20200104, 300d);
272.        ValueDateTupel vdtCalculated20200105_300 = new ValueDateTupel(
273.            date_20200105, 300d);
274.        ValueDateTupel[] expectedVdtArray1 = { //
275.            valueDateTupel1, //
276.            vdtCalculated20200102_200, //
277.            valueDateTupel3, //
278.            vdtCalculated20200104_400, //
279.            valueDateTupel5 };
280.        ValueDateTupel[] expectedVdtArray2 = { //
281.            vdtCalculated20200101_200, //
282.            valueDateTupel2, //
283.            valueDateTupel3, //
284.            valueDateTupel4, //
285.            vdtCalculated20200105_400 };
286.        ValueDateTupel[] expectedVdtArray3 = { //
287.            valueDateTupel1, //
288.            vdtCalculated20200102_250, //
289.            vdtCalculated20200103_250, //
290.            valueDateTupel4, //
291.            valueDateTupel5 };
292.        ValueDateTupel[] expectedVdtArray4 = { //
293.            vdtCalculated20200101_300, //
294.            vdtCalculated20200102_300, //
295.            valueDateTupel3, //
296.            valueDateTupel4, //
297.            valueDateTupel5 };
298.        ValueDateTupel[] expectedVdtArray5 = { //
299.            valueDateTupel1, //
300.            valueDateTupel2, //
301.            valueDateTupel3, //
302.            vdtCalculated20200104_300, //
303.            vdtCalculated20200105_300 };
304.        ValueDateTupel[] expectedVdtArray6 = { //
305.            valueDateTupel1, //
306.            valueDateTupel2, //
307.            valueDateTupel3, //
308.            valueDateTupel4, //
309.            valueDateTupel5 };
310.        ValueDateTupel[][] expectedValue = { //
311.            expectedVdtArray1, //
312.            expectedVdtArray2, //
313.            expectedVdtArray3, //
314.            expectedVdtArray4, //
315.            expectedVdtArray5, //
316.            expectedVdtArray6 };
317.
318.        ValueDateTupel[] vdtArray1 = { valueDateTupel1, valueDateTupel3,
319.            valueDateTupel5 };
```

```
320.        ValueDateTupel[] vdtArray2 = { valueDateTupel2, valueDateTupel3,
321.            valueDateTupel4 };
322.        ValueDateTupel[] vdtArray3 = { valueDateTupel1, valueDateTupel4,
323.            valueDateTupel5 };
324.        ValueDateTupel[] vdtArray4 = { valueDateTupel3, valueDateTupel4,
325.            valueDateTupel5 };
326.        ValueDateTupel[] vdtArray5 = { valueDateTupel1, valueDateTupel2,
327.            valueDateTupel3 };
328.        ValueDateTupel[] vdtArray6 = { valueDateTupel1, valueDateTupel2,
329.            valueDateTupel3, valueDateTupel4, valueDateTupel5 };
330.        ValueDateTupel[][] vdtArraysArray = { vdtArray1, vdtArray2, vdtArray3,
331.            vdtArray4, vdtArray5, vdtArray6 };
332.        ValueDateTupel[][] actualValue = ValueDateTupel
333.            .alignDates(vdtArraysArray);
334.
335.        assertArrayEquals(expectedValue, actualValue,
336.            "Dates aren't correct after aligning ValueDateTuples");
337.    }
338.
339.    /**
340.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
341.     */
342.    @Test
343.    void testAlignDates_middleMissing() {
344.      date_20200101 = LocalDateTime.of(2020, 1, 1, 0, 0);
345.      date_20200102 = LocalDateTime.of(2020, 1, 2, 0, 0);
346.      date_20200103 = LocalDateTime.of(2020, 1, 3, 0, 0);
347.      valueDateTupel1 = new ValueDateTupel(date_20200101, value1);
348.      valueDateTupel2 = new ValueDateTupel(date_20200102, value2);
349.      valueDateTupel3 = new ValueDateTupel(date_20200103, value3);
350.
351.      ValueDateTupel[] expectedVdtArray1 = { //
352.          valueDateTupel1, //
353.          valueDateTupel2, //
354.          valueDateTupel3 };
355.      ValueDateTupel[] expectedVdtArray2 = { //
356.          new ValueDateTupel(LocalDateTime.of(2020, 1, 1, 0, 0), 999), //
357.          new ValueDateTupel(LocalDateTime.of(2020, 1, 2, 0, 0), 1099.6), //
358.          new ValueDateTupel(LocalDateTime.of(2020, 1, 3, 0, 0), 1200.2) };
359.      ValueDateTupel[][] expectedValue = { //
360.          expectedVdtArray1, //
361.          expectedVdtArray2 };
362.
363.      ValueDateTupel[] vdtArray1 = { valueDateTupel1, valueDateTupel2,
364.          valueDateTupel3 };
365.      ValueDateTupel[] vdtArray2 = {
366.          new ValueDateTupel(LocalDateTime.of(2020, 1, 1, 0, 0), 999),
367.          new ValueDateTupel(LocalDateTime.of(2020, 1, 3, 0, 0), 1200.2) };
368.      ValueDateTupel[][] vdtArraysArray = { vdtArray1, vdtArray2 };
369.      ValueDateTupel[][] actualValue = ValueDateTupel
370.          .alignDates(vdtArraysArray);
371.
372.      assertArrayEquals(expectedValue[0], actualValue[0],
373.          "Dates aren't correct after aligning ValueDateTuples");
```

```
374.      assertArrayEquals(expectedValue[1], actualValue[1],
375.          "Dates aren't correct after aligning ValueDateTuples");
376.    }
377.
378.    /**
379.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
380.     */
381.    @Test
382.    void testAlignDates_arrayOfArraysNull() {
383.      String expectedMessage = "Given array of arrays must not be null";
384.
385.      ValueDateTupel[][] vdtArraysArray = null;
386.      Exception thrown = assertThrows( //
387.          IllegalArgumentException.class, //
388.          () -> ValueDateTupel.alignDates(vdtArraysArray), //
389.          "Array of arrays = null is not correctly handled");
390.
391.      assertEquals(expectedMessage, thrown.getMessage(),
392.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
393.    }
394.
395.    /**
396.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
397.     */
398.    @Test
399.    void testAlignDates_arrayNull() {
400.      String expectedMessage = "The array at position 0 does not meet "
401.          + "specifications.";
402.      String expectedCause = "The given values array must not be null";
403.
404.      ValueDateTupel[] vdtArray1 = null;
405.      ValueDateTupel[] vdtArray2 = { valueDateTupel2, valueDateTupel3,
406.          valueDateTupel4 };
407.      ValueDateTupel[][] vdtArraysArray = { vdtArray1, vdtArray2 };
408.      Exception thrown = assertThrows( //
409.          IllegalArgumentException.class, //
410.          () -> ValueDateTupel.alignDates(vdtArraysArray), //
411.          "null in array of arrays is not correctly handled");
412.
413.      assertEquals(expectedMessage, thrown.getMessage(),
414.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
415.      assertEquals(expectedCause, thrown.getCause().getMessage(),
416.          MESSAGE_ARRAY_MUST_NOT_BE_NULL);
417.    }
418.
419.    /**
420.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
421.     */
422.    @Test
423.    void testAlignDates_arrayContainsNull() {
424.      String expectedMessage = "The array at position 0 does not meet "
425.          + "specifications.";
426.      String expectedCause = "Given values must not contain null.";
427.
```

```
428.      ValueDateTupel[] vdtArray1 = { valueDateTupel1, null,
429.          valueDateTupel5 };
430.      ValueDateTupel[] vdtArray2 = { valueDateTupel2, valueDateTupel3,
431.          valueDateTupel4 };
432.      ValueDateTupel[][] vdtArraysArray = { vdtArray1, vdtArray2 };
433.      Exception thrown = assertThrows( //
434.          IllegalArgumentException.class, //
435.          () -> ValueDateTupel.alignDates(vdtArraysArray), //
436.          "null in array is not correctly handled");
437.
438.      assertEquals(expectedMessage, thrown.getMessage(),
439.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
440.      assertEquals(expectedCause, thrown.getCause().getMessage(),
441.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
442.    }
443.
444.    /**
445.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
446.     */
447.    @Test
448.    void testAlignDates_arrayNotSortedAscending() {
449.      String expectedMessage = "The array at position 0 does not meet "
450.          + "specifications.";
451.      String expectedCause = "Given values are not properly sorted or there "
452.          + "are non-unique values.";
453.
454.      ValueDateTupel[] vdtArray1 = { valueDateTupel1, valueDateTupel5,
455.          valueDateTupel3 };
456.      ValueDateTupel[] vdtArray2 = { valueDateTupel2, valueDateTupel3,
457.          valueDateTupel4 };
458.      ValueDateTupel[][] vdtArraysArray = { vdtArray1, vdtArray2 };
459.      Exception thrown = assertThrows( //
460.          IllegalArgumentException.class, //
461.          () -> ValueDateTupel.alignDates(vdtArraysArray), //
462.          "Unsorted array is not correctly handled");
463.
464.      assertEquals(expectedMessage, thrown.getMessage(),
465.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
466.      assertEquals(expectedCause, thrown.getCause().getMessage(),
467.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
468.    }
469.
470.    /**
471.     * Test method for {@link ValueDateTupel#alignDates(ValueDateTupel[][])}.
472.     */
473.    @Test
474.    void testAlignDates_onlyNaN() {
475.      String expectedMessage = "The array at position 0 does not meet "
476.          + "specifications.";
477.      String expectedCause = "Given values must not contain NaN.";
478.
479.      ValueDateTupel vdt1NaN = new ValueDateTupel(date_20200101, Double.NaN);
480.      ValueDateTupel vdt3NaN = new ValueDateTupel(date_20200103, Double.NaN);
481.      ValueDateTupel vdt5NaN = new ValueDateTupel(date_20200105, Double.NaN);
```

```
482.
483.      ValueDateTupel[] vdtArray1 = { vdt1NaN, vdt3NaN, vdt5NaN };
484.      ValueDateTupel[] vdtArray2 = { valueDateTupel2, valueDateTupel3,
485.          valueDateTupel4 };
486.      ValueDateTupel[][] vdtArraysArray = { vdtArray1, vdtArray2 };
487.      Exception thrown = assertThrows( //
488.          IllegalArgumentException.class, //
489.          () -> ValueDateTupel.alignDates(vdtArraysArray), //
490.          "Only NaN values in ValueDateTupel is not correctly handled");
491.
492.      assertEquals(expectedMessage, thrown.getMessage(),
493.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
494.      assertEquals(expectedCause, thrown.getCause().getMessage(),
495.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
496.    }
497.
498.    /**
499.     * Test method for
500.     * {@link ValueDateTupel#contains(ValueDateTupel[], ValueDateTupel)}.
501.     */
502.    @Test
503.    void testContains() {
504.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
505.          valueDateTupel2, valueDateTupel3 };
506.
507.      assertTrue(
508.          ValueDateTupel.contains(valueDateTupelArray, valueDateTupel1),
509.          "An element which is in the array falsely cannot be identified");
510.    }
511.
512.    /**
513.     * Test method for
514.     * {@link ValueDateTupel#contains(ValueDateTupel[], ValueDateTupel)}.
515.     */
516.    @Test
517.    void testContains_unknownElement() {
518.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
519.          valueDateTupel2, valueDateTupel3 };
520.
521.      assertFalse(
522.          ValueDateTupel.contains(valueDateTupelArray, valueDateTupel4),
523.          "An unknown element is falsely marked as being in the given "
524.              + "array");
525.    }
526.
527.    /**
528.     * Test method for
529.     * {@link ValueDateTupel#contains(ValueDateTupel[], ValueDateTupel)}.
530.     */
531.    @Test
532.    void testContains_arrayNull() {
533.      String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
534.
535.      ValueDateTupel[] valueDateTupelArray = null;
```

```
536.       Exception thrown = assertThrows(
537.           IllegalArgumentException.class, () -> ValueDateTupel
538.               .contains(valueDateTupelArray, valueDateTupel4),
539.           "Array of null is not properly handled.");
540.
541.       assertEquals(expectedMessage, thrown.getMessage(),
542.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
543.     }
544.
545.     /**
546.      * Test method for
547.      * {@link ValueDateTupel#containsDate(ValueDateTupel[], LocalDateTime)}.
548.      */
549.     @Test
550.     void testContainsDate() {
551.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
552.           valueDateTupel2, valueDateTupel3 };
553.
554.       assertTrue(
555.           ValueDateTupel.containsDate(valueDateTupelArray, date_20200101),
556.           "A DateTime which is in the array falsely cannot be identified");
557.     }
558.
559.     /**
560.      * Test method for
561.      * {@link ValueDateTupel#containsDate(ValueDateTupel[], LocalDateTime)}.
562.      */
563.     @Test
564.     void testContainsDate_unknownDate() {
565.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
566.           valueDateTupel2, valueDateTupel3 };
567.
568.       assertFalse(
569.           ValueDateTupel.containsDate(valueDateTupelArray, date_20200104),
570.           "An unknown DateTime is falsely marked as being in the given "
571.               + "array");
572.     }
573.
574.     /**
575.      * Test method for
576.      * {@link ValueDateTupel#containsDate(ValueDateTupel[], LocalDateTime)}.
577.      */
578.     @Test
579.     void testContainsDate_arrayNull() {
580.       String expectedMessage = "Given array must not be null";
581.
582.       ValueDateTupel[] valueDateTupelArray = null;
583.       Exception thrown = assertThrows(
584.           IllegalArgumentException.class, () -> ValueDateTupel
585.               .containsDate(valueDateTupelArray, date_20200101),
586.           "Null array is not properly handled");
587.
588.       assertEquals(expectedMessage, thrown.getMessage(),
589.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
```

```
590.     }
591.
592.     /**
593.      * Test method for
594.      * {@link ValueDateTupel#containsDate(ValueDateTupel[], LocalDateTime)}.
595.      */
596.     @Test
597.     void testContainsDate_dateTimeNull() {
598.       String expectedMessage = MESSAGE_VALUE_MUST_NOT_BE_NULL;
599.
600.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
601.           valueDateTupel2, valueDateTupel3 };
602.       Exception thrown = assertThrows(IllegalArgumentException.class,
603.           () -> ValueDateTupel.containsDate(valueDateTupelArray, null),
604.           "LocalDateTime of null is not properly handled");
605.
606.       assertEquals(expectedMessage, thrown.getMessage(),
607.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
608.     }
609.
610.     /**
611.      * Test method for
612.      * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
613.      */
614.     @Test
615.     void testAddOneAt_position_0() {
616.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel2,
617.           valueDateTupel3, valueDateTupel4 };
618.       ValueDateTupel[] expectedValue = { valueDateTupel1, valueDateTupel2,
619.           valueDateTupel3, valueDateTupel4 };
620.
621.       ValueDateTupel[] actualValue = ValueDateTupel
622.           .addOneAt(valueDateTupelArray, valueDateTupel1, 0);
623.
624.       assertArrayEquals(expectedValue, actualValue,
625.           "Value cannot be added correctly at position 0");
626.     }
627.
628.     /**
629.      * Test method for
630.      * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
631.      */
632.     @Test
633.     void testAddOneAt_position_end() {
634.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
635.           valueDateTupel2, valueDateTupel3 };
636.       ValueDateTupel[] expectedValue = { valueDateTupel1, valueDateTupel2,
637.           valueDateTupel3, valueDateTupel4 };
638.
639.       ValueDateTupel[] actualValue = ValueDateTupel
640.           .addOneAt(valueDateTupelArray, valueDateTupel4, 3);
641.
642.       assertArrayEquals(expectedValue, actualValue,
643.           "Value cannot be added correctly at last position");
```

```
644.     }
645.
646.     /**
647.      * Test method for
648.      * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
649.      */
650.     @Test
651.     void testAddOneAt_position_between() {
652.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
653.           valueDateTupel3, valueDateTupel4 };
654.       ValueDateTupel[] expectedValue = { valueDateTupel1, valueDateTupel2,
655.           valueDateTupel3, valueDateTupel4 };
656.
657.       ValueDateTupel[] actualValue = ValueDateTupel
658.           .addOneAt(valueDateTupelArray, valueDateTupel2, 1);
659.
660.       assertArrayEquals(expectedValue, actualValue,
661.           "Value cannot be added correctly at inbetween position");
662.     }
663.
664.     /**
665.      * Test method for
666.      * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
667.      */
668.     @Test
669.     void testAddOneAt_arrayNull() {
670.       String expectedMessage = "Given array must not be null";
671.
672.       ValueDateTupel[] valueDateTupelArray = null;
673.       Exception thrown = assertThrows(
674.           IllegalArgumentException.class, () -> ValueDateTupel
675.               .addOneAt(valueDateTupelArray, valueDateTupel1, 0),
676.           "Array of null is not properly handled");
677.
678.       assertEquals(expectedMessage, thrown.getMessage(),
679.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
680.     }
681.
682.     /**
683.      * Test method for
684.      * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
685.      */
686.     @Test
687.     void testAddOneAt_valueNull() {
688.       String expectedMessage = MESSAGE_VALUE_MUST_NOT_BE_NULL;
689.
690.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
691.           valueDateTupel3, valueDateTupel4 };
692.       ValueDateTupel vdtNull = null;
693.       Exception thrown = assertThrows(IllegalArgumentException.class,
694.           () -> ValueDateTupel.addOneAt(valueDateTupelArray, vdtNull, 0),
695.           "New value of null is not properly handled");
696.
697.       assertEquals(expectedMessage, thrown.getMessage(),
```

```
698.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
699.      }
700.
701.      /**
702.       * Test method for
703.       * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
704.       */
705.      @Test
706.      void testAddOneAt_position_negative() {
707.        String expectedMessage = "Cannot add a value at position < 0. Given "
708.            + "position is -1";
709.
710.        ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
711.            valueDateTupel3, valueDateTupel4 };
712.        int negativePosition = -1;
713.        Exception thrown = assertThrows(IllegalArgumentException.class,
714.            () -> ValueDateTupel.addOneAt(valueDateTupelArray, valueDateTupel1,
715.                negativePosition),
716.            "Position < 0 is not properly handled");
717.
718.        assertEquals(expectedMessage, thrown.getMessage(),
719.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
720.      }
721.
722.      /**
723.       * Test method for
724.       * {@link ValueDateTupel#addOneAt(ValueDateTupel[], ValueDateTupel, int)}.
725.       */
726.      @Test
727.      void testAddOneAt_position_greater_arrayLength() {
728.        String expectedMessage = "Cannot add a value at position > 3. Given "
729.            + "position is 4.";
730.
731.        ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
732.            valueDateTupel3, valueDateTupel4 };
733.        int tooLargeAPosition = 4;
734.        Exception thrown = assertThrows(IllegalArgumentException.class,
735.            () -> ValueDateTupel.addOneAt(valueDateTupelArray, valueDateTupel1,
736.                tooLargeAPosition),
737.            "Position > array.length is not properly handled");
738.
739.        assertEquals(expectedMessage, thrown.getMessage(),
740.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
741.      }
742.
743.      /**
744.       * Test method for
745.       * {@link ValueDateTupel#getElement(ValueDateTupel[], LocalDateTime)}.
746.       */
747.      @Test
748.      void testContainsLocalDateTime() {
749.        ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
750.            valueDateTupel3, valueDateTupel4 };
751.        ValueDateTupel expectedValue = valueDateTupel1;
```

```
752.
753.       ValueDateTupel actualValue = ValueDateTupel
754.           .getElement(valueDateTupelArray, date_20200101);
755.
756.       assertEquals(expectedValue, actualValue,
757.           "Value cannot be properly found");
758.     }
759.
760.     /**
761.      * Test method for
762.      * {@link ValueDateTupel#getElement(ValueDateTupel[], LocalDateTime)}.
763.      */
764.     @Test
765.     void testContainsLocalDateTime_dateNotFound() {
766.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
767.           valueDateTupel3, valueDateTupel4 };
768.
769.       ValueDateTupel actualValue = ValueDateTupel
770.           .getElement(valueDateTupelArray, date_20200102);
771.
772.       assertNull(actualValue,
773.           "Date non existant in array is not properly handled");
774.     }
775.
776.     /**
777.      * Test method for
778.      * {@link ValueDateTupel#getElement(ValueDateTupel[], LocalDateTime)}.
779.      */
780.     @Test
781.     void testContainsLocalDateTime_arrayNull() {
782.       String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
783.
784.       ValueDateTupel[] valueDateTupelArray = null;
785.       Exception thrown = assertThrows(
786.           IllegalArgumentException.class, () -> ValueDateTupel
787.               .getElement(valueDateTupelArray, date_20200101),
788.           "Array of null is not properly handled");
789.
790.       assertEquals(expectedMessage, thrown.getMessage(),
791.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
792.     }
793.
794.     /**
795.      * Test method for
796.      * {@link ValueDateTupel#getElement(ValueDateTupel[], LocalDateTime)}.
797.      */
798.     @Test
799.     void testContainsLocalDateTime_dateToBeFoundNull() {
800.       String expectedMessage = MESSAGE_VALUE_MUST_NOT_BE_NULL;
801.
802.       ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
803.           valueDateTupel3, valueDateTupel4 };
804.       LocalDateTime dateNull = null;
805.       Exception thrown = assertThrows(IllegalArgumentException.class,
```

```
806.          () -> ValueDateTupel.getElement(valueDateTupelArray, dateNull),
807.          "Date to be found of null is not properly handled");
808.
809.      assertEquals(expectedMessage, thrown.getMessage(),
810.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
811.    }
812.
813.    /**
814.     * Test method for {@link ValueDateTupel#getValues(ValueDateTupel[])}.
815.     */
816.    @Test
817.    void testGetValues() {
818.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
819.          valueDateTupel3, valueDateTupel4 };
820.      double[] expectedValues = { //
821.          valueDateTupel1.getValue(), // 100
822.          valueDateTupel3.getValue(), // 300
823.          valueDateTupel4.getValue() // 400
824.      };
825.
826.      double[] actualValues = ValueDateTupel.getValues(valueDateTupelArray);
827.
828.      assertArrayEquals(expectedValues, actualValues,
829.          "Incorrect values are gotten from array");
830.    }
831.
832.    /**
833.     * Test method for {@link ValueDateTupel#getValues(ValueDateTupel[])}.
834.     */
835.    @Test
836.    void testGetValues_arrayNull() {
837.      String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
838.
839.      ValueDateTupel[] valueDateTupelArray = null;
840.      Exception thrown = assertThrows(IllegalArgumentException.class,
841.          () -> ValueDateTupel.getValues(valueDateTupelArray),
842.          "Null array is not correctly handled");
843.
844.      assertEquals(expectedMessage, thrown.getMessage(),
845.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
846.    }
847.
848.    /**
849.     * Test method for {@link ValueDateTupel#getDates(ValueDateTupel[])}.
850.     */
851.    @Test
852.    void testGetDates() {
853.      ValueDateTupel[] valueDateTupelArray = { valueDateTupel1,
854.          valueDateTupel3, valueDateTupel4 };
855.      LocalDateTime[] expectedValues = { //
856.          valueDateTupel1.getDate(), // date_20200101
857.          valueDateTupel3.getDate(), // date_20200102
858.          valueDateTupel4.getDate() // date_20200103
859.      };
```

```
860.
861.        LocalDateTime[] actualValues = ValueDateTupel
862.            .getDates(valueDateTupelArray);
863.
864.        assertArrayEquals(expectedValues, actualValues,
865.            "Incorrect values are gotten from array");
866.      }
867.
868.      /**
869.       * Test method for {@link ValueDateTupel#getDates(ValueDateTupel[])}.
870.       */
871.      @Test
872.      void testGetDates_arrayNull() {
873.        String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
874.
875.        ValueDateTupel[] valueDateTupelArray = null;
876.        Exception thrown = assertThrows(IllegalArgumentException.class,
877.            () -> ValueDateTupel.getDates(valueDateTupelArray),
878.            "Null array is not correctly handled");
879.
880.        assertEquals(expectedMessage, thrown.getMessage(),
881.            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
882.      }
883.
884.      /**
885.       * Test method for
886.       * {@link ValueDateTupel#getElements(ValueDateTupel[], LocalDateTime,
887.       * LocalDateTime)}.
888.       */
889.      @Test
890.      void testGetElements() {
891.        ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2,
892.            valueDateTupel3, valueDateTupel4, valueDateTupel5 };
893.        ValueDateTupel[] expectedValue = { valueDateTupel2, valueDateTupel3,
894.            valueDateTupel4 };
895.
896.        ValueDateTupel[] actualValue = ValueDateTupel.getElements(vdtArray,
897.            date_20200102, date_20200104);
898.
899.        assertArrayEquals(expectedValue, actualValue,
900.            "Elements cannot be correctly retrieved.");
901.      }
902.
903.      /**
904.       * Test method for
905.       * {@link ValueDateTupel#getElements(ValueDateTupel[], LocalDateTime,
906.       * LocalDateTime)}.
907.       */
908.      @Test
909.      void testGetElements_dtFromNull() {
910.        ValueDateTupel[] expectedValue = { valueDateTupel1, valueDateTupel2,
911.            valueDateTupel3, valueDateTupel4 };
912.
913.        ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2,
```

```
914.          valueDateTupel3, valueDateTupel4, valueDateTupel5 };
915.      ValueDateTupel[] actualValue = ValueDateTupel.getElements(vdtArray,
916.          null, date_20200104);
917.
918.      assertArrayEquals(expectedValue, actualValue,
919.          "Elements cannot be correctly retrieved when dtFrom is null.");
920.    }
921.
922.    /**
923.     * Test method for
924.     * {@link ValueDateTupel#getElements(ValueDateTupel[], LocalDateTime,
925.     * LocalDateTime)}.
926.     */
927.    @Test
928.    void testGetElements_dtToNull() {
929.      ValueDateTupel[] expectedValue = { valueDateTupel2, valueDateTupel3,
930.          valueDateTupel4, valueDateTupel5 };
931.
932.      ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2,
933.          valueDateTupel3, valueDateTupel4, valueDateTupel5 };
934.      ValueDateTupel[] actualValue = ValueDateTupel.getElements(vdtArray,
935.          date_20200102, null);
936.
937.      assertArrayEquals(expectedValue, actualValue,
938.          "Elements cannot be correctly retrieved when dtTo is null.");
939.    }
940.
941.    /**
942.     * Test method for
943.     * {@link ValueDateTupel#getElements(ValueDateTupel[], LocalDateTime,
944.     * LocalDateTime)}.
945.     */
946.    @Test
947.    void testGetElements_dtFromEqualsDtTo() {
948.      ValueDateTupel[] expectedValue = { valueDateTupel2 };
949.
950.      ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2,
951.          valueDateTupel3, valueDateTupel4, valueDateTupel5 };
952.      ValueDateTupel[] actualValue = ValueDateTupel.getElements(vdtArray,
953.          date_20200102, date_20200102);
954.
955.      assertArrayEquals(expectedValue, actualValue,
956.          "Elements cannot be correctly retrieved when dtFrom equals dtTo.");
957.    }
958.
959.    /**
960.     * Test method for
961.     * {@link ValueDateTupel#getElements(ValueDateTupel[], LocalDateTime,
962.     * LocalDateTime)}.
963.     */
964.    @Test
965.    void testGetElements_dtFromNotInArray() {
966.      ValueDateTupel[] vdtArray = { valueDateTupel4, valueDateTupel5 };
967.
```

```
968.       assertNull(
969.           ValueDateTupel.getElements(vdtArray, date_20200102, date_20200105),
970.           "dtFrom not in array is not properly handled.");
971.     }
972.
973.     /**
974.      * Test method for
975.      * {@link ValueDateTupel#getElements(ValueDateTupel[], LocalDateTime,
976.      * LocalDateTime)}.
977.      */
978.     @Test
979.     void testGetElements_dtToNotInArray() {
980.       ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2 };
981.
982.       assertNull(
983.           ValueDateTupel.getElements(vdtArray, date_20200102, date_20200105),
984.           "dtTo not in array is not properly handled.");
985.     }
986.
987.     /**
988.      * Test method for
989.      * {@link ValueDateTupel#getPosition(ValueDateTupel[], LocalDateTime)}.
990.      */
991.     @Test
992.     void testGetPosition() {
993.       int expectedValueFirstPosition = 0;
994.       int expectedValueLastPosition = 4;
995.
996.       ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2,
997.           valueDateTupel3, valueDateTupel4, valueDateTupel5 };
998.       int actualValueFirstPosition = ValueDateTupel.getPosition(vdtArray,
999.           date_20200101);
1000.      int actualValueLastPosition = ValueDateTupel.getPosition(vdtArray,
1001.          date_20200105);
1002.
1003.      assertEquals(expectedValueLastPosition, actualValueLastPosition,
1004.          "Position cannot be correctly retrieved");
1005.      assertEquals(expectedValueFirstPosition, actualValueFirstPosition,
1006.          "Position cannot be correctly retrieved");
1007.    }
1008.
1009.    /**
1010.     * Test method for
1011.     * {@link ValueDateTupel#getPosition(ValueDateTupel[], LocalDateTime)}.
1012.     */
1013.    @Test
1014.    void testGetPosition_arrayNull() {
1015.      String expectedMessage = MESSAGE_ARRAY_MUST_NOT_BE_NULL;
1016.
1017.      ValueDateTupel[] vdtArray = null;
1018.      Exception thrown = assertThrows(IllegalArgumentException.class,
1019.          () -> ValueDateTupel.getPosition(vdtArray, date_20200102),
1020.          "Array of null is not properly handled");
1021.
```

```
1022.       assertEquals(expectedMessage, thrown.getMessage(),
1023.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
1024.     }
1025.
1026.     /**
1027.      * Test method for
1028.      * {@link ValueDateTupel#getPosition(ValueDateTupel[], LocalDateTime)}.
1029.      */
1030.     @Test
1031.     void testGetPosition_arrayEmpty() {
1032.       int expectedValue = Integer.MIN_VALUE;
1033.
1034.       ValueDateTupel[] vdtArray = {};
1035.       assertEquals(expectedValue,
1036.           ValueDateTupel.getPosition(vdtArray, date_20200102),
1037.           "Empty array is not properly handled");
1038.     }
1039.
1040.     /**
1041.      * Test method for
1042.      * {@link ValueDateTupel#getPosition(ValueDateTupel[], LocalDateTime)}.
1043.      */
1044.     @Test
1045.     void testGetPosition_dtToBeFoundNull() {
1046.       String expectedMessage = MESSAGE_VALUE_MUST_NOT_BE_NULL;
1047.
1048.       ValueDateTupel[] vdtArray = { valueDateTupel1, valueDateTupel2,
1049.           valueDateTupel3, valueDateTupel4, valueDateTupel5 };
1050.       Exception thrown = assertThrows(IllegalArgumentException.class,
1051.           () -> ValueDateTupel.getPosition(vdtArray, null),
1052.           "Date to be found of null is not properly handled");
1053.
1054.       assertEquals(expectedMessage, thrown.getMessage(),
1055.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
1056.     }
1057.   }
```

Komponente UtilTest

Listing 35: Komponente UtilTest

```
1.    package de.rumford.tradingsystem.helper;
2.
3.    import static org.junit.jupiter.api.Assertions.*;
4.
5.    import java.time.LocalDateTime;
6.
7.    import org.junit.jupiter.api.BeforeAll;
8.    import org.junit.jupiter.api.BeforeEach;
9.    import org.junit.jupiter.api.Test;
10.
11.   import de.rumford.tradingsystem.BaseValue;
12.   import de.rumford.tradingsystem.Rule;
13.   import de.rumford.tradingsystem.RuleTest.RealRule;
14.
15.   /**
16.    * Test class for {@link Util}.
17.    *
18.    * @author Max Rumford
19.    *
20.    */
21.   class UtilTest {
22.
23.     static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
24.         "Incorrect Exception message";
25.
26.     static final String BASE_VALUE_NAME = "Base value name";
27.     static BaseValue baseValue;
28.
29.     static LocalDateTime localDateTimeJan09220000;
30.     static LocalDateTime localDateTimeJan10220000;
31.     static LocalDateTime localDateTimeJan11220000;
32.     static LocalDateTime localDateTimeJan12220000;
33.
34.     static final double VARIATOR = 1;
35.     static final double BASE_SCALE = 10;
36.
37.     static Rule r1;
38.     static Rule r2;
39.     static Rule r3;
40.
41.     @BeforeAll
42.     static void setUpBeforeClass() throws Exception {
43.       baseValue = BaseValueFactory.jan1Feb05calcShort(BASE_VALUE_NAME);
44.       localDateTimeJan09220000 = LocalDateTime.of(2020, 01, 9, 22, 0);
45.       localDateTimeJan10220000 = LocalDateTime.of(2020, 01, 10, 22, 0);
46.       localDateTimeJan11220000 = LocalDateTime.of(2020, 01, 11, 22, 0);
47.       localDateTimeJan12220000 = LocalDateTime.of(2020, 01, 12, 22, 0);
48.     }
49.
```

```
50.     @BeforeEach
51.     void setUp() throws Exception {
52.       r1 = RealRule.from(baseValue, null, localDateTimeJan10220000,
53.           localDateTimeJan12220000, BASE_SCALE, VARIATOR);
54.       r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
55.           localDateTimeJan12220000, BASE_SCALE, 2);
56.       r3 = RealRule.from(baseValue, null, localDateTimeJan10220000,
57.           localDateTimeJan12220000, BASE_SCALE, 3);
58.     }
59.
60.     /**
61.      * Test method for
62.      * {@link Util#adjustForStandardDeviation(double, double)}.
63.      */
64.     @Test
65.     void testAdjustForStandardDeviation() {
66.       double value = 100d;
67.       double standardDeviation = 2.5d;
68.       double expectedValue = 40d;
69.
70.       double actualValue = Util.adjustForStandardDeviation(value,
71.           standardDeviation);
72.
73.       assertEquals(expectedValue, actualValue,
74.           "Standard deviation adjusted value is not correctly calculated");
75.     }
76.
77.     /**
78.      * Test method for
79.      * {@link Util#adjustForStandardDeviation(double, double)}.
80.      */
81.     @Test
82.     void testAdjustForStandardDeviation_sd0() {
83.       double value = 100d;
84.       double standardDeviation = 0d;
85.
86.       double actualValue = Util.adjustForStandardDeviation(value,
87.           standardDeviation);
88.
89.       assertTrue(Double.isNaN(actualValue),
90.           "Stanard deviation of zero is not properly handled");
91.     }
92.
93.     /**
94.      * Test method for {@link Util#areRulesUnique(Rule[])}.
95.      */
96.     @Test
97.     void testAreRulesUnique_identicalRules() {
98.       r1 = RealRule.from(baseValue, null, localDateTimeJan10220000,
99.           localDateTimeJan12220000, BASE_SCALE, VARIATOR);
100.      r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
101.          localDateTimeJan12220000, BASE_SCALE, VARIATOR);
102.      Rule[] rules = { r1, r2 };
103.
```

```
104.        assertFalse(Util.areRulesUnique(rules),
105.            "Identical rules are not identified.");
106.    }
107.
108.    /**
109.     * Test method for {@link Util#areRulesUnique(Rule[])}.
110.     */
111.    @Test
112.    void testAreRulesUnique_uniqueRules() {
113.      Rule[] variations = { r3 };
114.      r2 = RealRule.from(baseValue, variations, localDateTimeJan10220000,
115.          localDateTimeJan12220000, BASE_SCALE, VARIATOR);
116.      Rule[] rules = { r1, r2 };
117.      assertTrue(Util.areRulesUnique(rules),
118.          "Unique rules are not identified.");
119.
120.      r2 = RealRule.from(baseValue, null, localDateTimeJan09220000,
121.          localDateTimeJan12220000, BASE_SCALE, VARIATOR);
122.      Rule[] rules2 = { r1, r2 };
123.      assertTrue(Util.areRulesUnique(rules2),
124.          "Unique rules are not identified.");
125.
126.      r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
127.          localDateTimeJan11220000, BASE_SCALE, VARIATOR);
128.      Rule[] rules3 = { r1, r2 };
129.      assertTrue(Util.areRulesUnique(rules3),
130.          "Unique rules are not identified.");
131.
132.      @SuppressWarnings("unused")
133.      double diffBaseScale = (BASE_SCALE - 1 <= 0 ? BASE_SCALE + 1
134.          : BASE_SCALE - 1);
135.      r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
136.          localDateTimeJan12220000, diffBaseScale, VARIATOR);
137.      Rule[] rules4 = { r1, r2 };
138.      assertTrue(Util.areRulesUnique(rules4),
139.          "Unique rules are not identified.");
140.
141.      double diffVariator = VARIATOR - 1;
142.      r2 = RealRule.from(baseValue, null, localDateTimeJan10220000,
143.          localDateTimeJan12220000, BASE_SCALE, diffVariator);
144.      Rule[] rules5 = { r1, r2 };
145.      assertTrue(Util.areRulesUnique(rules5),
146.          "Unique rules are not identified.");
147.    }
148.
149.    /**
150.     * Test method for {@link Util#areRulesUnique(Rule[])}.
151.     */
152.    @Test
153.    void testAreRulesUnique_rules_null() {
154.      String expectedMessage = "The given rules must not be null";
155.      Rule[] rules = null;
156.
157.      Exception thrown = assertThrows(IllegalArgumentException.class,
```

```
158.          () -> Util.areRulesUnique(rules),
159.          "Rules array of null is not properly handled");
160.
161.      assertEquals(expectedMessage, thrown.getMessage(),
162.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
163.    }
164.
165.    /**
166.     * Test method for {@link Util#areRulesUnique(Rule[])}.
167.     */
168.    @Test
169.    void testAreRulesUnique_rulesContains_null() {
170.      String expectedMessage = "The given array must not contain nulls";
171.      Rule[] rules = { null };
172.
173.      Exception thrown = assertThrows(IllegalArgumentException.class,
174.          () -> Util.areRulesUnique(rules),
175.          "Rules array containing null is not properly handled");
176.
177.      assertEquals(expectedMessage, thrown.getMessage(),
178.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
179.    }
180.
181.    /**
182.     * Test method for {@link Util#areRulesUnique(Rule[])}.
183.     */
184.    @Test
185.    void testAreRulesUnique_rules_empty() {
186.      String expectedMessage = "The given array of rules must not be empty.";
187.      Rule[] rules = {};
188.
189.      Exception thrown = assertThrows(IllegalArgumentException.class,
190.          () -> Util.areRulesUnique(rules),
191.          "Empty rules array is not properly handled");
192.
193.      assertEquals(expectedMessage, thrown.getMessage(),
194.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
195.    }
196.
197.    /**
198.     * Test method for {@link Util#calculateForecast(double, double)}.
199.     */
200.    @Test
201.    void testCalculateForecast() {
202.      double unscaledForecast = 2.5d;
203.      double scalar = 4d;
204.      double expectedValue = 10d;
205.
206.      double actualValue = Util.calculateForecast(unscaledForecast, scalar);
207.
208.      assertEquals(expectedValue, actualValue,
209.          "Forecast is not correctly calculated");
210.    }
211.
```

```
212.    /**
213.     * Test method for {@link Util#calculateAverage(double[])}.
214.     */
215.    @Test
216.    void testCalculateAverage() {
217.      double[] values = { 1, 2, 3 };
218.      double expectedValue = 2;
219.
220.      double actualValue = Util.calculateAverage(values);
221.
222.      assertEquals(expectedValue, actualValue,
223.          "Average value is not properly calculated");
224.    }
225.
226.    /**
227.     * Test method for {@link Util#calculateAverage(double[])}.
228.     */
229.    @Test
230.    void testCalculateAverage_withNegatives() {
231.      double[] values = { 1, 2, -3, 4 };
232.      double expectedValue = 1;
233.
234.      double actualValue = Util.calculateAverage(values);
235.
236.      assertEquals(expectedValue, actualValue,
237.          "Average value of values containing negatives is not properly "
238.          + "calculated");
239.    }
240.
241.    /**
242.     * Test method for {@link Util#calculateAverage(double[])}.
243.     */
244.    @Test
245.    void testCalculateAverage_arrayNull() {
246.      double[] values = null;
247.      String expectedMessage = "Given array must not be null";
248.
249.      Exception thrown = assertThrows(IllegalArgumentException.class,
250.          () -> Util.calculateAverage(values),
251.          "Array of null is not properly handled");
252.
253.      assertEquals(expectedMessage, thrown.getMessage(),
254.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
255.    }
256.
257.    /**
258.     * Test method for
259.     * {@link Util#calculateForecastScalar(double[], double)}.
260.     */
261.    @Test
262.    void testCalculateForecastScalar() {
263.      double[] values = { 10d, 4d, -1d, 6d, -4d };
264.      double baseScale = 10d;
265.      double expectedValue = 2d;
```

```
266.
267.      double actualValue = Util.calculateForecastScalar(values, baseScale);
268.
269.      assertEquals(expectedValue, actualValue,
270.          "Forecast scalar is not correctly calculated");
271.    }
272.
273.    /**
274.     * Test method for
275.     * {@link Util#calculateForecastScalar(double[], double)}.
276.     */
277.    @Test
278.    void testCalculateForecastScalar_absoluteAverage0() {
279.      double[] values = { 0d, -0d };
280.      double baseScale = 10d;
281.      double expectedValue = Double.NaN;
282.
283.      double actualValue = Util.calculateForecastScalar(values, baseScale);
284.
285.      assertEquals(expectedValue, actualValue,
286.          "Forecast scalar is not correctly calculated when average of "
287.          + "absolute values is zero");
288.    }
289.
290.    /**
291.     * Test method for
292.     * {@link Util#calculateForecastScalar(double[], double)}.
293.     */
294.    @Test
295.    void testCalculateForecastScalar_noValues() {
296.      double[] values = {};
297.      double baseScale = 10d;
298.      String expectedMessage = "Given array of values must not be empty";
299.
300.      Exception thrown = assertThrows(IllegalArgumentException.class,
301.          () -> Util.calculateForecastScalar(values, baseScale),
302.          "Empty values array is not properly handled");
303.
304.      assertEquals(expectedMessage, thrown.getMessage(),
305.          MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
306.    }
307.
308.    /**
309.     * Test method for
310.     * {@link Util#calculateForecastScalar(double[], double)}.
311.     */
312.    @Test
313.    void testCalculateForecastScalar_baseScale0() {
314.      String expectedMessage = "Given base scale does not meet "
315.          + "specifications.";
316.      String expectedCause = "Value must be a positive decimal";
317.      double[] values = { 10d, 4d, -1d, 6d, -4d };
318.      double baseScale = 0;
319.
```

```
320.       Exception thrown = assertThrows(IllegalArgumentException.class,
321.           () -> Util.calculateForecastScalar(values, baseScale),
322.           "Base scale of 0 is not properly handled");
323.
324.       assertEquals(expectedMessage, thrown.getMessage(),
325.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
326.       assertEquals(expectedCause, thrown.getCause().getMessage(),
327.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
328.     }
329.
330.     /**
331.      * Test method for {@link Util#calculateReturn(double, double)}.
332.      */
333.     @Test
334.     void testCalculateReturn_former200_latter300() {
335.       double formerValue = 200d;
336.       double latterValue = 300d;
337.       double expectedValue = 0.5d;
338.
339.       double actualValue = Util.calculateReturn(formerValue, latterValue);
340.
341.       assertEquals(expectedValue, actualValue,
342.           "Positive return is not correctly calculated");
343.     }
344.
345.     /**
346.      * Test method for {@link Util#calculateReturn(double, double)}.
347.      */
348.     @Test
349.     void testCalculateReturn_former300_latter200() {
350.       double formerValue = 400d;
351.       double latterValue = 200d;
352.       double expectedValue = -0.5d;
353.
354.       double actualValue = Util.calculateReturn(formerValue, latterValue);
355.
356.       assertEquals(expectedValue, actualValue,
357.           "Negative return is not correctly calculated");
358.     }
359.
360.     /* Test method for {@link Util#calculateReturn(double, double)}. */
361.     @Test
362.     void testCalculateReturn_formerValue0() {
363.       double formerValue = 0d;
364.       double latterValue = 10d;
365.       double expectedValue = Double.NaN;
366.
367.       double actualValue = Util.calculateReturn(formerValue, latterValue);
368.
369.       assertEquals(expectedValue, actualValue,
370.           "Former value of 0 is not properly handled");
371.     }
372.
373.     /**
```

```
374.      * Test method for {@link Util#calculateCorrelationOfRows(double[][])}.
375.      */
376.     @Test
377.     void testCalculateCorrelationsOfThreeRows() {
378.       double[] row1 = { -20, -20, -12.31, -5.34 };
379.       double[] row2 = { -20, -20, -20, -17.93 };
380.       double[] row3 = { -9.59, -10.62, -9.8, -9.23 };
381.       double[][] values = { row1, row2, row3 };
382.       // Excel: 0.857736784518697, 0.688500766307298, 0.656405176216209
383.       double[] expectedValue = { 0.8577367845186973, 0.6885007663072988,
384.           0.6564051762162094 };
385.
386.       double[] actualValue = Util.calculateCorrelationOfRows(values);
387.
388.       assertArrayEquals(expectedValue, actualValue,
389.           "Correlations between three rows are not correctly calculated");
390.     }
391.
392.     /**
393.      * Test method for {@link Util#calculateCorrelationOfRows(double[][])}.
394.      */
395.     @Test
396.     void testCalculateCorrelationsOfThreeRows_valuesContainsArrWithNan() {
397.       double[][] values = { { 0, 4 }, { 2, Double.NaN }, { 1, 0.2 } };
398.       double[] expectedCorrelations = { Double.NaN, -1, Double.NaN };
399.
400.       double[] actualCorrelations = Util.calculateCorrelationOfRows(values);
401.
402.       assertArrayEquals(expectedCorrelations, actualCorrelations,
403.           "Correlations containing NaN are not correctly calculated");
404.     }
405.
406.     /**
407.      * Test method for {@link Util#calculateCorrelationOfRows(double[][])}.
408.      */
409.     @Test
410.     void testCalculateCorrelationsOfThreeRows_allEqaulValues() {
411.       double[] row1 = { 1, 1, 1 };
412.       double[] row2 = { 2, 3, 4 };
413.       double[] row3 = { 3, 3, 5 };
414.       double[][] values = { row1, row2, row3 };
415.       String expectedmessage = "Correlations cannot be calculated caused by "
416.           + "all identical values in row at position 0.";
417.
418.       Exception thrown = assertThrows(IllegalArgumentException.class,
419.           () -> Util.calculateCorrelationOfRows(values),
420.           "Arrays containing all equal values are not properly handled.");
421.
422.       assertEquals(expectedmessage, thrown.getMessage(),
423.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
424.     }
425.
426.     /**
427.      * Test method for
```

```
428.      * {@link Util#calculateWeightsForThreeCorrelations(double[])}.
429.      */
430.     @Test
431.     void testCalculateWeightsForThreeCorrelations() {
432.       // Excel: 0.3, 0.366666666666667, 0.333333333333333
433.       double[] expectedValue = { .3, 0.3666666666666667, 1d / 3d };
434.       double[] correlations = { .5, .6, .4 };
435.
436.       double[] actualValue = Util
437.           .calculateWeightsForThreeCorrelations(correlations);
438.
439.       assertArrayEquals(expectedValue, actualValue,
440.           "Weights for 3 correlations are not correctly calculated");
441.     }
442.
443.     /**
444.      * Test method for
445.      * {@link Util#calculateWeightsForThreeCorrelations(double[])}.
446.      */
447.     @Test
448.     void testCalculateWeightsForThreeCorrelations_negativeCorrelations() {
449.       double[] correlations1 = { .5, .6, -.4 };
450.       double[] correlations2 = { .5, .6, 0 };
451.
452.       double[] actualValue1 = Util
453.           .calculateWeightsForThreeCorrelations(correlations1);
454.       double[] actualValue2 = Util
455.           .calculateWeightsForThreeCorrelations(correlations2);
456.
457.       assertArrayEquals(actualValue1, actualValue2,
458.           "Weights for negative correlations are not correctly calculated");
459.     }
460.
461.     /**
462.      * Test method for
463.      * {@link Util#calculateWeightsForThreeCorrelations(double[])}.
464.      */
465.     @Test
466.     void testCalculateWeightsForThreeCorrelations_threeEqualWeights() {
467.       double[] expectedValue = { 1d / 3d, 1d / 3d, 1d / 3d };
468.       double[] correlations = { 1, 1, 1 };
469.
470.       double[] actualValue = Util
471.           .calculateWeightsForThreeCorrelations(correlations);
472.
473.       assertArrayEquals(expectedValue, actualValue,
474.           "Weights for 3 equal correlations are not correctly calculated");
475.     }
476.
477.     /* Test method for {@link Util#getPositionFromForecast(double)}. */
478.     @Test
479.     void testGetPositionFromForecast() {
480.       final String LONG = "Long";
481.       final String SHORT = "Short";
```

```
482.        final String HOLD = "Hold";
483.
484.        String actualValueLong = Util.getPositionFromForecast(5);
485.        String actualValueShort = Util.getPositionFromForecast(-10);
486.        String actualValueHold = Util.getPositionFromForecast(0);
487.
488.        assertEquals(LONG, actualValueLong,
489.            "Position literal for positive forecasts is not inferred.");
490.        assertEquals(SHORT, actualValueShort,
491.            "Position literal for negative forecasts is not inferred.");
492.        assertEquals(HOLD, actualValueHold,
493.            "Position literal for forecasts of 0 is not inferred.");
494.      }
495.    }
```

Komponente ValidatorTest

Listing 36: Komponente ValidatorTest

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  import static org.junit.jupiter.api.Assertions.assertEquals;
4.  import static org.junit.jupiter.api.Assertions.assertThrows;
5.
6.  import java.time.LocalDateTime;
7.
8.  import org.junit.jupiter.api.Test;
9.
10. import de.rumford.tradingsystem.BaseValue;
11. import de.rumford.tradingsystem.RuleTest.RealRule;
12.
13. /**
14.  * Test class for {@link Validator}.
15.  *
16.  * @author Max Rumford
17.  *
18.  */
19. class ValidatorTest {
20.
21.   static final String MESSAGE_INCORRECT_EXCEPTION_MESSAGE =
22.       "Incorrect Exception message";
23.
24.   /**
25.    * Test method for {@link Validator#validateCorrelations(double[])}.
26.    */
27.   @Test
28.   void testValidateCorrelations_arrayNull() {
29.     String expectedMessage = "Correlations array must not be null";
30.     double[] correlations = null;
31.
32.     Exception thrown = assertThrows(IllegalArgumentException.class,
33.         () -> Validator.validateCorrelations(correlations),
34.         "Array of null is not properly handled");
35.
36.     assertEquals(expectedMessage, thrown.getMessage(),
37.         MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
38.   }
39.
40.   /**
41.    * Test method for {@link Validator#validateCorrelations(double[])}.
42.    */
43.   @Test
44.   void testValidateCorrelations_arrayOfLengthNotThree() {
45.     String expectedMessage = "There must be exactly three correlation "
46.         + "values in the given array";
47.     double[] correlations = { 0, 0 };
48.
49.     Exception thrown = assertThrows(IllegalArgumentException.class,
```

```
            () -> Validator.validateCorrelations(correlations),
            "Array of length != 3 is not properly handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for {@link Validator#validateCorrelations(double[])}.
     */
    @Test
    void testValidateCorrelations_arrayContainsNan() {
        String expectedMessage = "NaN-values are not allowed. Correlation at "
            + "position 1 is NaN.";
        double[] correlations = { 0, Double.NaN, 0 };

        Exception thrown = assertThrows(IllegalArgumentException.class,
            () -> Validator.validateCorrelations(correlations),
            "Array containing Double.NaN is not properly handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for {@link Validator#validateCorrelations(double[])}.
     */
    @Test
    void testValidateCorrelations_arrayContainsValueGreaterThan1() {
        String expectedMessage = "Correlation at position 1 is greater than 1";
        double[] correlations = { 0, 2, 0 };

        Exception thrown = assertThrows(IllegalArgumentException.class,
            () -> Validator.validateCorrelations(correlations),
            "Array containing values greater than 1 is not properly handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
    }

    /**
     * Test method for {@link Validator#validateCorrelations(double[])}.
     */
    @Test
    void testValidateCorrelations_arrayContainsValueLessThanNegative1() {
        String expectedMessage = "Correlation at position 1 is less than -1";
        double[] correlations = { 0, -2, 0 };

        Exception thrown = assertThrows(IllegalArgumentException.class,
            () -> Validator.validateCorrelations(correlations),
            "Array containing values less than -1 is not properly handled");

        assertEquals(expectedMessage, thrown.getMessage(),
            MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
```

```
104.     }
105.
106.     /**
107.      * Test method for {@link Validator#validateRow(ValueDateTupel[])}.
108.      */
109.     @Test
110.     void testValidateRow_onlyNan() {
111.       ValueDateTupel v1 = new ValueDateTupel(
112.           LocalDateTime.of(2019, 1, 1, 22, 0), Double.NaN);
113.       ValueDateTupel v2 = new ValueDateTupel(
114.           LocalDateTime.of(2019, 1, 1, 22, 0), Double.NaN);
115.       ValueDateTupel[] values = { v1, v2 };
116.
117.       String expectedMessage = "Row contains only Double.NaN. Rows must "
118.           + "contain at least one value != Double.NaN";
119.
120.       Exception thrown = assertThrows(IllegalArgumentException.class,
121.           () -> Validator.validateRow(values),
122.           "Row only conataning NaNs is not properly handled");
123.
124.       assertEquals(expectedMessage, thrown.getMessage(),
125.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
126.     }
127.
128.     /**
129.      * Test method for {@link Validator#validateRow(ValueDateTupel[])}.
130.      */
131.     @Test
132.     void testValidateRow_oneNonNan() {
133.       ValueDateTupel v1 = new ValueDateTupel(
134.           LocalDateTime.of(2019, 1, 1, 22, 0), Double.NaN);
135.       ValueDateTupel v2 = new ValueDateTupel(
136.           LocalDateTime.of(2019, 1, 1, 22, 0), 1d);
137.       ValueDateTupel[] values = { v1, v2 };
138.
139.       Validator.validateRow(values);
140.     }
141.
142.     /**
143.      * Test method for {@link Validator#validateRow(ValueDateTupel[])}.
144.      */
145.     @Test
146.     void testValidateRow_firstNonNan() {
147.       ValueDateTupel v1 = new ValueDateTupel(
148.           LocalDateTime.of(2019, 1, 1, 22, 0), 1d);
149.       ValueDateTupel v2 = new ValueDateTupel(
150.           LocalDateTime.of(2019, 1, 1, 22, 0), Double.NaN);
151.       ValueDateTupel[] values = { v1, v2 };
152.
153.       Validator.validateRow(values);
154.     }
155.
156.     /**
157.      * Test method for
```

```
158.      * {@link Validator#validateRulesVsBaseScale(
159.      * de.rumford.tradingsystem.Rule[], double)}.
160.      */
161.     @Test
162.     void testValidateRulesVsBaseScale() {
163.       final double baseScale = 10;
164.       BaseValue baseValue = BaseValueFactory.jan1Jan31calcShort("TEST");
165.       LocalDateTime startOfRefWindow = LocalDateTime.of(2020, 1, 10, 22, 0);
166.       LocalDateTime endOfRefWindow = LocalDateTime.of(2020, 1, 12, 22, 0);
167.
168.       RealRule rr1 = new RealRule(baseValue, null, startOfRefWindow,
169.           endOfRefWindow, baseScale, 1);
170.       RealRule rr2 = new RealRule(baseValue, null, startOfRefWindow,
171.           endOfRefWindow, baseScale, 2);
172.
173.       RealRule[] rules = { rr1, rr2 };
174.
175.       Validator.validateRulesVsBaseScale(rules, baseScale);
176.     }
177.
178.     /**
179.      * Test method for
180.      * {@link Validator#validateRulesVsBaseScale(
181.      * de.rumford.tradingsystem.Rule[], double)}.
182.      */
183.     @Test
184.     void testValidateRulesVsBaseScale_rulesNull() {
185.       final double baseScale = 10;
186.       RealRule[] rules = null;
187.
188.       Validator.validateRulesVsBaseScale(rules, baseScale);
189.     }
190.
191.     /**
192.      * Test method for
193.      * {@link Validator#validateRulesVsBaseScale(
194.      * de.rumford.tradingsystem.Rule[], double)}.
195.      */
196.     @Test
197.     void testValidateRulesVsBaseScale_individualRuleNull() {
198.       final double baseScale = 10;
199.       BaseValue baseValue = BaseValueFactory.jan1Jan31calcShort("TEST");
200.       LocalDateTime startOfRefWindow = LocalDateTime.of(2020, 1, 10, 22, 0);
201.       LocalDateTime endOfRefWindow = LocalDateTime.of(2020, 1, 12, 22, 0);
202.
203.       RealRule rr1 = new RealRule(baseValue, null, startOfRefWindow,
204.           endOfRefWindow, baseScale, 1);
205.       RealRule rr2 = null;
206.
207.       RealRule[] rules = { rr1, rr2 };
208.
209.       Validator.validateRulesVsBaseScale(rules, baseScale);
210.     }
211.
```

```
212.     /**
213.      * Test method for
214.      * {@link Validator#validateRulesVsBaseScale(
215.      * de.rumford.tradingsystem.Rule[], double)}.
216.      */
217.     @Test
218.     void testValidateRulesVsBaseScale_wrongBaseScaleInRule() {
219.       final double baseScale = 10;
220.       String expectedMessage = "The rule at index 0 does not share the given"
221.           + " base scale of " + baseScale + ".";
222.       final double wrongBaseScale = 20;
223.       BaseValue baseValue = BaseValueFactory.jan1Jan31calcShort("TEST");
224.       LocalDateTime startOfRefWindow = LocalDateTime.of(2020, 1, 10, 22, 0);
225.       LocalDateTime endOfRefWindow = LocalDateTime.of(2020, 1, 12, 22, 0);
226.
227.       RealRule rr1 = new RealRule(baseValue, null, startOfRefWindow,
228.           endOfRefWindow, wrongBaseScale, 1);
229.       RealRule rr2 = new RealRule(baseValue, null, startOfRefWindow,
230.           endOfRefWindow, baseScale, 2);
231.
232.       RealRule[] rules = { rr1, rr2 };
233.
234.       Exception thrown = assertThrows(IllegalArgumentException.class,
235.           () -> Validator.validateRulesVsBaseScale(rules, baseScale),
236.           "Incorrect base scale is not properly handled.");
237.
238.       assertEquals(expectedMessage, thrown.getMessage(),
239.           MESSAGE_INCORRECT_EXCEPTION_MESSAGE);
240.     }
241.   }
```

Komponente DataSourceTest

Listing 37: Komponente DataSourceTest

```
1.  package de.rumford.tradingsystem.helper;
2.
3.  import static org.junit.jupiter.api.Assertions.assertEquals;
4.  import static org.junit.jupiter.api.Assertions.assertThrows;
5.  import static org.junit.jupiter.api.Assertions.fail;
6.
7.  import java.io.BufferedWriter;
8.  import java.io.File;
9.  import java.io.FileWriter;
10. import java.io.IOException;
11. import java.nio.file.Path;
12.
13. import org.apache.commons.lang3.ArrayUtils;
14. import org.apache.commons.lang3.RandomStringUtils;
15. import org.junit.jupiter.api.*;
16.
17. /**
18.  * Test class for {@link DataSource}.
19.  *
20.  * @apiNote Tests might fail every 17 in 36^20 cases due to the random
21.  *          generating of file names.
22.  * @author Max Rumford
23.  *
24.  */
25. class DataSourceTest {
26.
27.    final static int FILE_NAME_LENGTH = 20;
28.    final static int FOLDER_NAME_LENGTH = FILE_NAME_LENGTH;
29.
30.    /*
31.     * Have all temporary created files inside a folder inside
32.     * src/test/resources
33.     */
34.    private static String workingDir = Path
35.        .of("src", "test", "resources",
36.            RandomStringUtils.randomAlphanumeric(FOLDER_NAME_LENGTH))
37.        .toString();
38.
39.    /*
40.     * Generate random file names as not to accidentally overwrite any
41.     * existing files
42.     */
43.    final static String FILE_NAME_CORRECT_FILE_EUR = Path
44.        .of(workingDir,
45.            RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
46.        .toString();
47.    final static String FILE_NAME_CORRECT_FILE_US = Path
48.        .of(workingDir,
49.            RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
```

```
50.          .toString();
51.      final static String FILE_NAME_CORRECT_FILE_EUR_YMD = Path
52.          .of(workingDir,
53.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
54.          .toString();
55.      final static String FILE_NAME_CORRECT_FILE_EU_THOUSANDS_SEPARATOR =
56.          workingDir + RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH);
57.      final static String FILE_NAME_NULL = null;
58.      final static String FILE_NAME_UNKOWN = Path
59.          .of(workingDir,
60.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
61.          .toString();
62.      final static String FILE_NAME_DIRECTORY = Path
63.          .of(workingDir,
64.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
65.          .toString();
66.      final static String FILE_NAME_FILE_HAS_HEADINGS = Path
67.          .of(workingDir,
68.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
69.          .toString();
70.      final static String FILE_NAME_FOUR_COLUMNS = Path
71.          .of(workingDir,
72.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
73.          .toString();
74.      final static String FILE_NAME_DAY_NON_INTEGER = Path
75.          .of(workingDir,
76.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
77.          .toString();
78.      final static String FILE_NAME_MONTH_NON_INTEGER = Path
79.          .of(workingDir,
80.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
81.          .toString();
82.      final static String FILE_NAME_YEAR_NON_INTEGER = Path
83.          .of(workingDir,
84.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
85.          .toString();
86.      final static String FILE_NAME_HOUR_NON_INTEGER = Path
87.          .of(workingDir,
88.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
89.          .toString();
90.      final static String FILE_NAME_MINUTE_NON_INTEGER = Path
91.          .of(workingDir,
92.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
93.          .toString();
94.      final static String FILE_NAME_SECOND_NON_INTEGER = Path
95.          .of(workingDir,
96.              RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
97.          .toString();
98.      final static String FILE_NAME_DATE_VALUE_OUT_OF_RANGE = Path
99.          .of(workingDir,
100.             RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
101.         .toString();
102.     final static String FILE_NAME_COURSE_VALUE_INVALID = Path
103.         .of(workingDir,
```

```
104.             RandomStringUtils.randomAlphanumeric(FILE_NAME_LENGTH))
105.         .toString();
106.     static String[] FILE_NAMES = { //
107.         FILE_NAME_CORRECT_FILE_EUR, //
108.         FILE_NAME_CORRECT_FILE_US, //
109.         FILE_NAME_CORRECT_FILE_EUR_YMD, //
110.         FILE_NAME_CORRECT_FILE_EU_THOUSANDS_SEPARATOR, //
111.         FILE_NAME_DIRECTORY, //
112.         FILE_NAME_FILE_HAS_HEADINGS, //
113.         FILE_NAME_FOUR_COLUMNS, //
114.         FILE_NAME_DAY_NON_INTEGER, //
115.         FILE_NAME_MONTH_NON_INTEGER, //
116.         FILE_NAME_YEAR_NON_INTEGER, //
117.         FILE_NAME_HOUR_NON_INTEGER, //
118.         FILE_NAME_MINUTE_NON_INTEGER, //
119.         FILE_NAME_SECOND_NON_INTEGER, //
120.         FILE_NAME_DATE_VALUE_OUT_OF_RANGE, //
121.         FILE_NAME_COURSE_VALUE_INVALID,//
122.     };
123.
124.     static BufferedWriter bw_eu_ok;
125.     static BufferedWriter bw_us_ok;
126.     static BufferedWriter bw_eu_ymd;
127.     static BufferedWriter bw_eu_thousands_separator;
128.     static BufferedWriter bw_headings;
129.     static BufferedWriter bw_four_columns;
130.     static BufferedWriter bw_day_non_integer;
131.     static BufferedWriter bw_month_non_integer;
132.     static BufferedWriter bw_year_non_integer;
133.     static BufferedWriter bw_hour_non_integer;
134.     static BufferedWriter bw_minute_non_integer;
135.     static BufferedWriter bw_second_non_integer;
136.     static BufferedWriter bw_date_value_out_of_range;
137.     static BufferedWriter bw_course_value_invalid;
138.
139.     static BufferedWriter[] bufferedWriters = {};
140.
141.     /**
142.      * @throws java.lang.Exception
143.      */
144.     @BeforeAll
145.     static void setUpBeforeClass() throws Exception {
146.       new File(workingDir).mkdirs();
147.       bw_eu_ok = new BufferedWriter(
148.           new FileWriter(new File(FILE_NAME_CORRECT_FILE_EUR)));
149.       bw_us_ok = new BufferedWriter(
150.           new FileWriter(new File(FILE_NAME_CORRECT_FILE_US)));
151.       bw_eu_ymd = new BufferedWriter(
152.           new FileWriter(new File(FILE_NAME_CORRECT_FILE_EUR_YMD)));
153.       bw_eu_thousands_separator = new BufferedWriter(new FileWriter(
154.           new File(FILE_NAME_CORRECT_FILE_EU_THOUSANDS_SEPARATOR)));
155.
156.       /*
157.        * No Writer for FILE_NAME_NULL and FILE_NAME_UNKOWN
```

```
158.        */
159.       new File(FILE_NAME_DIRECTORY).mkdirs();
160.       bw_headings = new BufferedWriter(
161.           new FileWriter(new File(FILE_NAME_FILE_HAS_HEADINGS)));
162.       bw_four_columns = new BufferedWriter(
163.           new FileWriter(new File(FILE_NAME_FOUR_COLUMNS)));
164.       bw_day_non_integer = new BufferedWriter(
165.           new FileWriter(new File(FILE_NAME_DAY_NON_INTEGER)));
166.       bw_month_non_integer = new BufferedWriter(
167.           new FileWriter(new File(FILE_NAME_MONTH_NON_INTEGER)));
168.       bw_year_non_integer = new BufferedWriter(
169.           new FileWriter(new File(FILE_NAME_YEAR_NON_INTEGER)));
170.       bw_hour_non_integer = new BufferedWriter(
171.           new FileWriter(new File(FILE_NAME_HOUR_NON_INTEGER)));
172.       bw_minute_non_integer = new BufferedWriter(
173.           new FileWriter(new File(FILE_NAME_MINUTE_NON_INTEGER)));
174.       bw_second_non_integer = new BufferedWriter(
175.           new FileWriter(new File(FILE_NAME_SECOND_NON_INTEGER)));
176.       bw_date_value_out_of_range = new BufferedWriter(
177.           new FileWriter(new File(FILE_NAME_DATE_VALUE_OUT_OF_RANGE)));
178.       bw_course_value_invalid = new BufferedWriter(
179.           new FileWriter(new File(FILE_NAME_COURSE_VALUE_INVALID)));
180.
181.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_eu_ok);
182.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_us_ok);
183.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_eu_ymd);
184.       bufferedWriters = ArrayUtils.add(bufferedWriters,
185.           bw_eu_thousands_separator);
186.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_headings);
187.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_four_columns);
188.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_day_non_integer);
189.       bufferedWriters = ArrayUtils.add(bufferedWriters,
190.           bw_month_non_integer);
191.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_year_non_integer);
192.       bufferedWriters = ArrayUtils.add(bufferedWriters, bw_hour_non_integer);
193.       bufferedWriters = ArrayUtils.add(bufferedWriters,
194.           bw_minute_non_integer);
195.       bufferedWriters = ArrayUtils.add(bufferedWriters,
196.           bw_second_non_integer);
197.       bufferedWriters = ArrayUtils.add(bufferedWriters,
198.           bw_date_value_out_of_range);
199.       bufferedWriters = ArrayUtils.add(bufferedWriters,
200.           bw_course_value_invalid);
201.     }
202.
203.     /**
204.      * @throws java.lang.Exception
205.      */
206.     @AfterAll
207.     static void tearDownAfterClass() throws Exception {
208.       /* Delete all files after tests are done */
209.       for (String fileName : FILE_NAMES) {
210.         File file = new File(fileName);
211.         try {
```

```
212.          file.delete();
213.        } catch (Exception e) {
214.          e.printStackTrace();
215.        }
216.      }
217.      File dir = new File(workingDir);
218.      dir.delete();
219.    }
220.
221.    /**
222.     * @throws java.lang.Exception
223.     */
224.    @BeforeEach
225.    void setUp() throws Exception {
226.    }
227.
228.    /**
229.     * @throws java.lang.Exception
230.     */
231.    @AfterEach
232.    void tearDown() throws Exception {
233.    }
234.
235.    /**
236.     * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
237.     */
238.    @Test
239.    void testGetDataFromCsv_EU_ok() {
240.      String line1 = "01.01.1981;22:00:00;480,92";
241.      String line2 = "02.01.1981;22:00:00;490,04";
242.      String line3 = "05.01.1981;22:00:00;493,05";
243.      String line4 = "06.01.1981;22:00:00;494,97";
244.      String line5 = "07.01.1981;22:00:00;489,89";
245.      String line6 = "08.01.1981;22:00:00;489,32";
246.      final int expectedValue = 6;
247.      try {
248.        bw_eu_ok.write(line1);
249.        bw_eu_ok.newLine();
250.        bw_eu_ok.write(line2);
251.        bw_eu_ok.newLine();
252.        bw_eu_ok.write(line3);
253.        bw_eu_ok.newLine();
254.        bw_eu_ok.write(line4);
255.        bw_eu_ok.newLine();
256.        bw_eu_ok.write(line5);
257.        bw_eu_ok.newLine();
258.        bw_eu_ok.write(line6);
259.        bw_eu_ok.newLine();
260.        bw_eu_ok.close();
261.      } catch (IOException e) {
262.        e.printStackTrace();
263.        fail("File could not be written");
264.      }
265.
```

```
266.      ValueDateTupel[] values = {};
267.      try {
268.        values = DataSource.getDataFromCsv(FILE_NAME_CORRECT_FILE_EUR,
269.            CsvFormat.EU);
270.      } catch (IllegalArgumentException e) {
271.        e.printStackTrace();
272.        fail("IllegalArgumentException getting Data");
273.      } catch (IOException e) {
274.        e.printStackTrace();
275.        fail("IOException getting Data");
276.      }
277.
278.      final int actualValue = values.length;
279.
280.      assertEquals(expectedValue, actualValue,
281.          "Number of read files are not correct");
282.    }
283.
284.    /**
285.     * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
286.     */
287.    @Test
288.    void testGetDataFromCsv_US_ok() {
289.      String line1 = "01/01/1981,22:00:00,480.92";
290.      String line2 = "02/01/1981,22:00:00,490.04";
291.      String line3 = "05/01/1981,22:00:00,493.05";
292.      String line4 = "06/01/1981,22:00:00,494.97";
293.      String line5 = "07/01/1981,22:00:00,489.89";
294.      String line6 = "08/01/1981,22:00:00,489.32";
295.      final int expectedValue = 6;
296.      try {
297.        bw_us_ok.write(line1);
298.        bw_us_ok.newLine();
299.        bw_us_ok.write(line2);
300.        bw_us_ok.newLine();
301.        bw_us_ok.write(line3);
302.        bw_us_ok.newLine();
303.        bw_us_ok.write(line4);
304.        bw_us_ok.newLine();
305.        bw_us_ok.write(line5);
306.        bw_us_ok.newLine();
307.        bw_us_ok.write(line6);
308.        bw_us_ok.newLine();
309.        bw_us_ok.close();
310.      } catch (IOException e) {
311.        e.printStackTrace();
312.        fail("File could not be written");
313.      }
314.
315.      ValueDateTupel[] values = {};
316.      try {
317.        values = DataSource.getDataFromCsv(FILE_NAME_CORRECT_FILE_US,
318.            CsvFormat.US);
319.      } catch (IllegalArgumentException e) {
```

```
320.         e.printStackTrace();
321.         fail("IllegalArgumentException getting Data");
322.       } catch (IOException e) {
323.         e.printStackTrace();
324.         fail("IOException getting Data");
325.       }
326.
327.       final int actualValue = values.length;
328.
329.       assertEquals(expectedValue, actualValue,
330.           "Number of read files are not correct");
331.     }
332.
333.     /**
334.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
335.      */
336.     @Test
337.     void testGetDataFromCsv_EuYMD_ok() {
338.       String line1 = "1981.01.01;22:00:00;480,92";
339.       String line2 = "1981.01.02;22:00:00;490,04";
340.       String line3 = "1981.01.03;22:00:00;493,05";
341.       String line4 = "1981.01.04;22:00:00;494,97";
342.       String line5 = "1981.01.05;22:00:00;489,89";
343.       String line6 = "1981.01.06;22:00:00;489,32";
344.       final int expectedValue = 6;
345.       try {
346.         bw_eu_ymd.write(line1);
347.         bw_eu_ymd.newLine();
348.         bw_eu_ymd.write(line2);
349.         bw_eu_ymd.newLine();
350.         bw_eu_ymd.write(line3);
351.         bw_eu_ymd.newLine();
352.         bw_eu_ymd.write(line4);
353.         bw_eu_ymd.newLine();
354.         bw_eu_ymd.write(line5);
355.         bw_eu_ymd.newLine();
356.         bw_eu_ymd.write(line6);
357.         bw_eu_ymd.newLine();
358.         bw_eu_ymd.close();
359.       } catch (IOException e) {
360.         e.printStackTrace();
361.         fail("File could not be written");
362.       }
363.
364.       ValueDateTupel[] values = {};
365.       try {
366.         values = DataSource.getDataFromCsv(FILE_NAME_CORRECT_FILE_EUR_YMD,
367.             CsvFormat.EU_YEAR_MONTH_DAY);
368.       } catch (IllegalArgumentException e) {
369.         e.printStackTrace();
370.         fail("IllegalArgumentException getting Data");
371.       } catch (IOException e) {
372.         e.printStackTrace();
373.         fail("IOException getting Data");
```

```
374.       }
375.
376.       final int actualValue = values.length;
377.
378.       assertEquals(expectedValue, actualValue,
379.           "Number of read files are not correct");
380.     }
381.
382.     /**
383.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
384.      */
385.     @Test
386.     void testGetDataFromCsv_EU_thousandsSeparator() {
387.       String line1 = "1981.01.01;22:00:00;1.480";
388.       String line2 = "1981.01.02;22:00:00;490,19";
389.       String line3 = "1981.01.03;22:00:00;2.493,99";
390.       String line4 = "1981.01.04;22:00:00;494";
391.       String line5 = "1981.01.05;22:00:00;999.999.489,2";
392.       String line6 = "1981.01.06;22:00:00;489";
393.       final int expectedValue = 6;
394.       try {
395.         bw_eu_thousands_separator.write(line1);
396.         bw_eu_thousands_separator.newLine();
397.         bw_eu_thousands_separator.write(line2);
398.         bw_eu_thousands_separator.newLine();
399.         bw_eu_thousands_separator.write(line3);
400.         bw_eu_thousands_separator.newLine();
401.         bw_eu_thousands_separator.write(line4);
402.         bw_eu_thousands_separator.newLine();
403.         bw_eu_thousands_separator.write(line5);
404.         bw_eu_thousands_separator.newLine();
405.         bw_eu_thousands_separator.write(line6);
406.         bw_eu_thousands_separator.newLine();
407.         bw_eu_thousands_separator.close();
408.       } catch (IOException e) {
409.         e.printStackTrace();
410.         fail("File could not be written");
411.       }
412.
413.       ValueDateTupel[] values = {};
414.       try {
415.         values = DataSource.getDataFromCsv(
416.             FILE_NAME_CORRECT_FILE_EU_THOUSANDS_SEPARATOR,
417.             CsvFormat.EU_YEAR_MONTH_DAY);
418.       } catch (IllegalArgumentException e) {
419.         e.printStackTrace();
420.         fail("IllegalArgumentException getting Data");
421.       } catch (IOException e) {
422.         e.printStackTrace();
423.         fail("IOException getting Data");
424.       }
425.
426.       final int actualValue = values.length;
427.
```

```
428.       assertEquals(expectedValue, actualValue,
429.           "Number of read files are not correct");
430.     }
431.
432.     /**
433.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
434.      */
435.     @Test
436.     void testGetDataFromCsv_fileName_null() {
437.       String expectedValue = "The given path cannot be processed";
438.       Exception thrown = assertThrows(IllegalArgumentException.class,
439.           () -> DataSource.getDataFromCsv(FILE_NAME_NULL, CsvFormat.US),
440.           "null file name is not properly handled");
441.       assertEquals(expectedValue, thrown.getMessage(),
442.           "Incorrect Exception message");
443.     }
444.
445.     /**
446.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
447.      */
448.     @Test
449.     void testGetDataFromCsv_nonExistant_file() {
450.       String expectedValue = "Given source path does not point to an existing"
451.           + " destination";
452.       Exception thrown = assertThrows(IOException.class,
453.           () -> DataSource.getDataFromCsv(FILE_NAME_UNKOWN, CsvFormat.US),
454.           "Unknown file name is not properly handled");
455.       assertEquals(expectedValue, thrown.getMessage(),
456.           "Incorrect Exception message");
457.     }
458.
459.     /**
460.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
461.      */
462.     @Test
463.     void testGetDataFromCsv_directory() {
464.       String expectedValue = "Given source path does not point to a file";
465.       Exception thrown = assertThrows(IOException.class,
466.           () -> DataSource.getDataFromCsv(FILE_NAME_DIRECTORY, CsvFormat.US),
467.           "Directory as file name is not properly handled");
468.       assertEquals(expectedValue, thrown.getMessage(),
469.           "Incorrect Exception message");
470.     }
471.
472.     /**
473.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
474.      */
475.     @Test
476.     void testGetDataFromCsv_headings() {
477.       String line0 = "DATE,TIME,VALUE";
478.       String line1 = "01/01/1981,22:00:00,480.92";
479.       try {
480.         bw_headings.write(line0);
481.         bw_headings.newLine();
```

```
482.         bw_headings.write(line1);
483.         bw_headings.newLine();
484.         bw_headings.close();
485.       } catch (IOException e) {
486.         e.printStackTrace();
487.         fail("File could not be written");
488.       }
489.
490.       assertThrows(
491.           Exception.class, () -> DataSource
492.               .getDataFromCsv(FILE_NAME_FILE_HAS_HEADINGS, CsvFormat.US),
493.           "Headings in CSV are not properly handled");
494.     }
495.
496.     /**
497.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
498.      */
499.     @Test
500.     void testGetDataFromCsv_four_columns() {
501.       String line1 = "01/01/1981,22:00:00,480.92,x";
502.       String expectedValue = "The passed CSV does not have an appropriate "
503.           + "number of columns";
504.       try {
505.         bw_four_columns.write(line1);
506.         bw_four_columns.newLine();
507.         bw_four_columns.close();
508.       } catch (IOException e) {
509.         e.printStackTrace();
510.         fail("File could not be written");
511.       }
512.
513.       Exception thrown = assertThrows(
514.           IllegalArgumentException.class, () -> DataSource
515.               .getDataFromCsv(FILE_NAME_FOUR_COLUMNS, CsvFormat.US),
516.           "Headings in CSV are not properly handled");
517.       assertEquals(expectedValue, thrown.getMessage(),
518.           "Incorrect Exception message");
519.     }
520.
521.     /**
522.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
523.      */
524.     @Test
525.     void testGetDataFromCsv_day_non_integer() {
526.       String line1 = "01/0z/1981,22:00:00,480.92";
527.       String expectedValue = "The date values of the read CSV file cannot be"
528.           + " parsed into numbers. Failing value >01/0z/1981<";
529.       try {
530.         bw_day_non_integer.write(line1);
531.         bw_day_non_integer.newLine();
532.         bw_day_non_integer.close();
533.       } catch (IOException e) {
534.         e.printStackTrace();
535.         fail("File could not be written");
```

```
536.       }
537.
538.       Exception thrown = assertThrows(IllegalArgumentException.class,
539.           () -> DataSource.getDataFromCsv(FILE_NAME_DAY_NON_INTEGER,
540.               CsvFormat.US),
541.           "Unparsable day in date field is not properly handled");
542.       assertEquals(expectedValue, thrown.getMessage(),
543.           "Incorrect Exception message");
544.     }
545.
546.     /**
547.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
548.      */
549.     @Test
550.     void testGetDataFromCsv_month_non_integer() {
551.       String line1 = "0z/01/1981,22:00:00,480.92";
552.       String expectedValue = "The date values of the read CSV file cannot be"
553.           + " parsed into numbers. Failing value >0z/01/1981<";
554.       try {
555.         bw_month_non_integer.write(line1);
556.         bw_month_non_integer.newLine();
557.         bw_month_non_integer.close();
558.       } catch (IOException e) {
559.         e.printStackTrace();
560.         fail("File could not be written");
561.       }
562.
563.       Exception thrown = assertThrows(IllegalArgumentException.class,
564.           () -> DataSource.getDataFromCsv(FILE_NAME_MONTH_NON_INTEGER,
565.               CsvFormat.US),
566.           "Unparsable month in date field is not properly handled");
567.       assertEquals(expectedValue, thrown.getMessage(),
568.           "Incorrect Exception message");
569.     }
570.
571.     /**
572.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
573.      */
574.     @Test
575.     void testGetDataFromCsv_year_non_integer() {
576.       String line1 = "01/01/19q1,22:00:00,480.92";
577.       String expectedValue = "The date values of the read CSV file cannot be"
578.           + " parsed into numbers. Failing value >01/01/19q1<";
579.       try {
580.         bw_year_non_integer.write(line1);
581.         bw_year_non_integer.newLine();
582.         bw_year_non_integer.close();
583.       } catch (IOException e) {
584.         e.printStackTrace();
585.         fail("File could not be written");
586.       }
587.
588.       Exception thrown = assertThrows(IllegalArgumentException.class,
589.           () -> DataSource.getDataFromCsv(FILE_NAME_YEAR_NON_INTEGER,
```

```
590.                 CsvFormat.US),
591.             "Unparsable year in date field is not properly handled");
592.       assertEquals(expectedValue, thrown.getMessage(),
593.             "Incorrect Exception message");
594.     }
595.
596.     /**
597.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
598.      */
599.     @Test
600.     void testGetDataFromCsv_hour_non_integer() {
601.       String line1 = "01/01/1981,2x:00:00,480.92";
602.       String expectedValue = "The time values of the read CSV file cannot be"
603.             + " parsed into numbers. Failing value >2x:00:00<";
604.       try {
605.         bw_hour_non_integer.write(line1);
606.         bw_hour_non_integer.newLine();
607.         bw_hour_non_integer.close();
608.       } catch (IOException e) {
609.         e.printStackTrace();
610.         fail("File could not be written");
611.       }
612.
613.       Exception thrown = assertThrows(IllegalArgumentException.class,
614.             () -> DataSource.getDataFromCsv(FILE_NAME_HOUR_NON_INTEGER,
615.                 CsvFormat.US),
616.             "Unparsable hour in time field is not properly handled");
617.       assertEquals(expectedValue, thrown.getMessage(),
618.             "Incorrect Exception message");
619.     }
620.
621.     /**
622.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
623.      */
624.     @Test
625.     void testGetDataFromCsv_minute_non_integer() {
626.       String line1 = "01/01/1981,22:x0:00,480.92";
627.       String expectedValue = "The time values of the read CSV file cannot be"
628.             + " parsed into numbers. Failing value >22:x0:00<";
629.       try {
630.         bw_minute_non_integer.write(line1);
631.         bw_minute_non_integer.newLine();
632.         bw_minute_non_integer.close();
633.       } catch (IOException e) {
634.         e.printStackTrace();
635.         fail("File could not be written");
636.       }
637.
638.       Exception thrown = assertThrows(IllegalArgumentException.class,
639.             () -> DataSource.getDataFromCsv(FILE_NAME_MINUTE_NON_INTEGER,
640.                 CsvFormat.US),
641.             "Unparsable minute in time field is not properly handled");
642.       assertEquals(expectedValue, thrown.getMessage(),
643.             "Incorrect Exception message");
```

```
644.     }
645.
646.     /**
647.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
648.      */
649.     @Test
650.     void testGetDataFromCsv_second_non_integer() {
651.       String line1 = "01/01/1981,22:00:0x,480.92";
652.       String expectedValue = "The time values of the read CSV file cannot be"
653.           + " parsed into numbers. Failing value >22:00:0x<";
654.       try {
655.         bw_second_non_integer.write(line1);
656.         bw_second_non_integer.newLine();
657.         bw_second_non_integer.close();
658.       } catch (IOException e) {
659.         e.printStackTrace();
660.         fail("File could not be written");
661.       }
662.
663.       Exception thrown = assertThrows(IllegalArgumentException.class,
664.           () -> DataSource.getDataFromCsv(FILE_NAME_SECOND_NON_INTEGER,
665.               CsvFormat.US),
666.           "Unparsable second in time field is not properly handled");
667.       assertEquals(expectedValue, thrown.getMessage(),
668.           "Incorrect Exception message");
669.     }
670.
671.     /**
672.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
673.      */
674.     @Test
675.     void testGetDataFromCsv_date_out_of_range() {
676.       String line1 = "15/32/1981,22:00:00,480.92";
677.       String expectedValue = "The date or time values of the read CSV file "
678.           + "cannot be parsed into a LocalDateTime instance. Failing values "
679.           + ">15/32/1981< and >22:00:00<.";
680.       try {
681.         bw_date_value_out_of_range.write(line1);
682.         bw_date_value_out_of_range.newLine();
683.         bw_date_value_out_of_range.close();
684.       } catch (IOException e) {
685.         e.printStackTrace();
686.         fail("File could not be written");
687.       }
688.
689.       Exception thrown = assertThrows(IllegalArgumentException.class,
690.           () -> DataSource.getDataFromCsv(FILE_NAME_DATE_VALUE_OUT_OF_RANGE,
691.               CsvFormat.US),
692.           "Out of range date is not properly handled");
693.       assertEquals(expectedValue, thrown.getMessage(),
694.           "Incorrect Exception message");
695.     }
696.
697.     /**
```

```
698.      * Test method for {@link DataSource#getDataFromCsv(String, CsvFormat)}.
699.      */
700.     @Test
701.     void testGetDataFromCsv_course_value_invalid() {
702.       String line1 = "01/01/1981,22:00:00,14x0.92";
703.       String expectedValue = "The course value >14x0.92< cannot be parsed";
704.       try {
705.         bw_course_value_invalid.write(line1);
706.         bw_course_value_invalid.newLine();
707.         bw_course_value_invalid.close();
708.       } catch (IOException e) {
709.         e.printStackTrace();
710.         fail("File could not be written");
711.       }
712.
713.       Exception thrown = assertThrows(
714.           IllegalArgumentException.class, () -> DataSource
715.               .getDataFromCsv(FILE_NAME_COURSE_VALUE_INVALID, CsvFormat.US),
716.           "Out of range date is not properly handled");
717.       assertEquals(expectedValue, thrown.getMessage(),
718.           "Incorrect Exception message");
719.     }
720.   }
```

Folgende Bände sind bisher in dieser Reihe erschienen:

Band 1 (2005)
Hermeier, Burghard / Frère, Eric / Heuermann, Marina
Ergebnisse und Effekte des Modellprojektes „Fit machen fürs Rating..."
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 2 (2006)
Hermeier, Burghard / Platzköster, Charlotte
Ergebnisse der ersten bundesweiten FOM-Marktstudie „Industrie-Dienstleistungen"
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 3 (2006)
Kern, Uwe / Pankow, Michael
Die Stärkung des traditionellen 3-stufigen Vertriebswegs im Sanitärmarkt durch den Einsatz neuer Medien
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 4 (2006)
Kürble, Peter
Die unternehmensinterne Wertschöpfungskette bei Dienstleistungen am Beispiel der TV-Programmveranstalter
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 5 (2007)
Klumpp, Matthias
Begriff und Konzept Berufswertigkeit
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 6 (2007)
Klumpp, Matthias / Jasper, Anke
Efficient Consumer Response (ECR) in der Logistikpraxis des Handels
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 7 (2007)
Klumpp, Matthias / Koppers, Laura
Kooperationsanforderungen im Supply Chain Management (SCM)
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 8 (2008)
Klumpp, Matthias
Das deutsche System der Berufsbildung im europäischen und internationalen Qualifikationsrahmen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 9 (2008)
Göke, Michael
Homo oeconomicus im Hörsaal – Die Rationalität studentischer Nebengespräche in Lehrveranstaltungen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 10 (2008)
Klumpp, Matthias / Rybnikova, Irma
Internationaler Vergleich und Forschungsthesen zu Studienformen in Deutschland
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 11 (2008)
Kratzsch, Uwe
Eine ökonomische Analyse einer Ausweitung des Arbeitnehmer-Entsendegesetzes
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 12 (2009)
Friedrich, Klaus
Organisationsentwicklung – Lernprozesse im Unternehmen durch Mitarbeiterbefragungen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 13 (2009)
Chaudhuri, Arun
Die Outsourcing/Offshoring Option aus der Perspektive der Neuen Institutionenökonomie
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 14 (2009)
Seng, Anja / Fleddermann, Nicole / Klumpp, Matthias
Der Bologna-Prozess
Hintergründe – Zielsetzung – Anforderungen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 15 (2009)
Jäschke, Thomas
Qualitätssteigerung bei gleichzeitigen Einsparungen –
Widerspruch oder Zukunft in der hausärztlichen Versorgung?
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 16 (2010)
Schütte, Michael
Beiträge zur Gesundheitsökonomie
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 17 (2010)
Bode, Olaf H. / Brimmen, Frank / Redeker, Ute
Die Einführung eines Mindestlohns in Deutschland –
Eine Makroökonomische Analyse
Introduction of a Minimum Wage in Germany – A Macroeconomic Analysis
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 18 (2011)
Nietsch, Cornelia / Weiffenbach, Hermann
Wirtschaftsethik – Einflussfaktoren ethischen Verhaltens in Unternehmen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 19 (2011)
Frère, Eric / Schyra, Andreas
Ausgewählte steuerliche Einflussfaktoren der Unternehmensbewertung
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 20 (2011)
Schulenburg, Nils / Jesgarzewski, Tim
Das Direktionsrecht des Arbeitgebers – Einsatzmöglichkeiten und Grenzen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 21 (2011)
Fichtner-Rosada, Sabine
Interaktive Hochschuldidaktik als Erfolgsfaktor im Studium für Berufstätige – Herausforderung und kompetenzorientierte Umsetzung
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 22 (2011)

Kern, Uwe / Negri, Michael, Whyte, Ligia
Needs of the Internet Industry
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 23 (2011)
Schütte, Michael
Management in ambulanten Sektor des Gesundheitswesens
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 24 (2011)
Holtfort, Thomas
Intuition, Risikowahrnehmung und Investmententscheidungen – Behaviorale Einflussfaktoren auf das Risikoverhalten privater Anleger
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 25 (2012)
Heinemann, Stefan / Hüsgen, Thomas / Seemann, Volker
Die Mindestliquiditätsquote –
Konkrete Auswirkungen auf den Wertpapier-Eigenbestand der Sparkassen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 26 (2012)
Hose, Christian / Lübke, Karsten / Nolte, Thomas / Obermeier, Thomas
Rating und Risikomanagement – Chancen und Risiken der Architektur des Ratingprozesses für die Validität der Ratingergebnisse
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 27 (2012)
Serfas, Sebastian
Illustrating the distortive impact of cognitive biases on knowledge generation, focusing on unconscious availability-induced distortions and SMEs
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 28 (2012)
Wollenweber, Leif-Erik
Customer Relationship Management im Mittelstand
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 29 (2012)
Nentwig, Holger / Obermeier, Thomas / Scholl, Guido
Ökonomische Fitness
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 30 (2012)
Büser, Tobias / Stein, Holger / von Königsmarck, Imke
Führungspraxis und Motivation – Empirische 360-Grad-Analyse auf Grundlage des MoKoCha-Führungsmodells und des Team Management Systems (TMS)
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 31 (2012)
Schulenburg, Nils / Knauer, Stefan
Altersgerechte Personalentwicklung – Bewertung von Instrumenten vor dem Hintergrund des demografischen Wandels
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 32 (2013)
Kinne, Peter
Balanced Governance – Komplexitätsbewältigung durch ausgewogenes Managen im Spannungsfeld erfolgskritischer Polaritäten
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 33 (2013)
Holtfort, Thomas
Beiträge zur Verhaltensökonomie: Einfluss von Priming-Effekten auf rationale vs. intuitive Entscheidungen bei komplexen Sachverhalten
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 34: (2013)
Mahood, Ed / Kameas, Achilles / Negri, Michael
Labelisation and Certification of e-Jobs – Theoretical considerations and practical approaches to foster employability in a dynamic industry
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 35 (2013)
Gondek, Christian / Heinemann, Stefan
An insight into Drivers of Customer Satisfaction – An empirical Study of a global automotive brand
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 36 (2013)
Rödder, Sascha / Schütte, Michael
Medizinische Versorgungszentren –
Chancen und Risiken der Implementierung im ambulanten Sektor des Gesundheitswesens
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 37 (2013)
Abele, Thomas / Ecke, Astrid
Erfolgsfaktoren von Innovationen in reifen Märkten
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 38 (2013)
Vatanparast, Mir Farid
Betriebswirtschaftliche Elemente im Social Entrepreneurship
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 39 (2013)
Seidel, Marcel
Die Anwendung heuristischer Regeln –
Eine Übersicht am Beispiel von Fusionen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 40 (2013)
Coburger, Dieter
Vertragsabschlüsse auf Internetplattformen – Rechtliche Risiken und Gestaltungsmöglichkeiten am Beispiel der Internetplattform eBay
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 41 (2013)
Kraus, Hans
Big Data − Einsatzfelder und Herausforderungen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 42 (2013)
Schmitz, Elmar
Textsammlung zur deutsch-chinesischen Wissenschaftsdialog
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 43 (2014)
Bruns, Kerstin
Führungskraft und Frau – manchmal ein Teufelskreis
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 44 (2014)
Deeken, Michael
Merkmale zukunftsfähiger Unternehmen – Erkenntnisse am Beispiel der Vermögensverwaltungsbranche
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 45 (2014)
Holzkämper. Hilko
Reformoptionen der Pflegeversicherung –
Eine ordnungstheoretische Analyse
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 46 (2014)
Kiefer, Markus
Neue Potenziale für die Krisenkommunikation von Unternehmen –
Social Media und die Kommunikation von großen Infrastrukturprojekten
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 47 (2014)
Hose, Christian / Lübke, Carsten / Nolte, Thomas / Obermeier, Thomas
Nachhaltigkeit als betriebswirtschaftlicher Wettbewerbsfaktor –
Eine Propensity Score Analyse Deutscher Aktiengesellschaften
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 48 (2014)
Chiwitt, Ulrich
Ratingagenturen – Fluch oder Segen?
Eine kritische Bestandsaufnahme
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 49 (2014)
Kipp, Volker
Aktuelle Entwicklungen in der Finanzierung mittelständischer Unternehmen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 50 (2014)
Nastansky, Andreas
Systemisches Risiko und systemrelevante Finanzinstitute
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 51 (2014)
Schat, Hans-Dieter
Direkte Beteiligung von Beschäftigten – Historische Entwicklung und aktuelle Umsetzung
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 52 (2014)
Sosa, Fabian
Anwaltskanzleien und Exportversicherungen – Konfliktlösungen für internationale Handelsgeschäfte
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 53 (2014)
Hose, Christian / Lübke, Karsten / Nolte, Thomas / Obermeier, Thomas
Einführung von Elektromobilität in Deutschland – Eine Bestandsaufnahme von Barrieren und Lösungsansätzen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 54 (2015)
Klukas, Jörg
Trend Empfehlungsmarketing in der Personalbeschaffung – Einordnung und empirische Analyse
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 55 (2015)
Wohlmann, Monika
Finanzmarktintegration in Mittelosteuropa: Eine empirische Analyse der integrativen Wirkung des Euro
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 56 (2015)
Rudolph, Elke
Crossmedia-Kommunikation, Komponenten, Planung, Implementierung und Prozesskontrolle- illustriert mit Beispielen aus der Entertainmentbranche
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 57 (2015)
Cervelló-Royo, Roberto / Guijarro Martínez, Francisco / Pfahler, Thomas / Preuss, Marion
Residential trade and industry –
European market analysis, future trends and influencing factors
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 58 (2016)
Hose, Christian / Obermeier, Thomas / Potthast, Robin
Demografischer Wandel: Implikationen für die Finanz- und Immobilienwirtschaft
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 59 (2016)
Fritsche, Charmaine
Cross-Sectional Tests of the Capital Asset Pricing Model –
in Stock Markets of the U.K. and the U.S.
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 60 (2016)
Löhr, Andreas / Ibragimov, Mansur
Determinants of Capital Structure in Times of Financial Crisis –
An Empirical Study with Focus on TecDAX Companies
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 61 (2016)
Dreesen, Heinz / Heuser, Elena / Holtfort, Thomas
Neuorganisation der Bankenaufsicht –
Auswirkungen und kritische Würdigung des einheitlichen europäischen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 62 (2016)
Kinne, Peter
Querschnitts-Disziplinen und ihr Synergiepotenzial zum Abbau dysfunktionaler Eigenkomplexität
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 63 (2016)
Schaff, Arnd / Gottschald, Jan
Prozessoptimierung im Produktentstehungs- und Intellectual Property Management Prozess unter besonderer Berücksichtigung von Schutzrechtsaspekten
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 64 (2016)
Richardt, Susanne
Chances and Challenges for Media-Based Instruction in Higher Education
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 65 (2016)
Godbersen, Hendrik
Die Führung von Apotheken mit Relationship Marketing –
Theorie, Empirie und Anwendung
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 66 (2016)
Ahrendt, Bernd
Komplexe Entscheidungssituationen für Führungskräfte im Kontext von Führungskonzepten und Selbstcoaching als Selbstreflexionsprozess für die Praxis
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 67 (2017)
Herlyn, Estelle
Zur Bedeutung von Nachhaltigkeit für die ökonomische Ausbildung
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 68 (2017)
Dotzauer, Andreas
Coaching in Theorie und Praxis –
Eine Bestandsaufnahme aus interdisziplinärer Perspektive
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 69 (2018)
Kotas, Carsten
Real Estate Crowdfunding in Deutschland –
Eine empirische Untersuchung vom 01.01.2012 - 31.12.2017
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 70 (2018)
Brademann, Isabell / Piorr, Rüdiger
Das affektive Commitment der Generation Z –
Eine empirische Analyse des Bindungsbedürfnisses an
Unternehmen und dessen Einflussfaktoren
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 71 (2018)
Bauerle, Christoph T.
Haftung in der Anlageberatung –
Die Empfehlung zum unterlassenen Wertpapierkauf aus rechtlicher Sicht
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 72 (2019)
Schwegler, Ulrike
Den Wandel gestalten: zukunftsorientiert führen –
Empirische Erkenntnisse und praktische Handlungsoptionen
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)

Band 73 (2019)
Heupel, Thomas / Hohoff, Christoph / Landherr, Gerrit
Internationalisierung der FOM Forschung – Berichte aus dem Europäischen Forschungsraum
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)
ISBN 978-3-89275-093-2 (Print) – ISBN 978-3-89275-094-9 (eBook)

Band 74 (2019)
Mann, Gerald
60 Jahre „Wohlstand für alle“ – Ludwig Erhard und die Soziale Marktwirtschaft
ISSN 1865-5610 (Print) - ISSN 2569-5800 (eBook)
ISBN 978-3-89275-095-2 (Print) – ISBN 978-3-89275-094-3 (eBook)

Band 75 (2019)
Schindler, Uwe
Customer Integration: Wettbewerbsvorteil durch intangible Faktoren
Erkenntnisse einer Studie aus dem Bereich der industriellen Fördertechnik
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)
ISBN 978-3-89275-113-7 (Print) – ISBN 978-3-89275-114-4 (eBook)

Band 76 (2020)
Behrens, Yvonne / Elsenheimer, Laura / Kantermann, Thomas /
Wiesener, Marc
Integration von berufsbegleitend Studierenden in die Forschung: Evaluation des digitalen Master-Forschungsforums 2020 der FOM Hochschule
ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)
ISBN 978-3-89275-160-1 (Print) – ISBN 978-3-89275-161-8 (eBook)

ISSN 1865-5610 (Print) – ISSN 2569-5800 (eBook)
ISBN 978-3-89275-164-9 (Print) – ISBN 978-3-89275-165-6 (eBook)

FOM Hochschule

FOM. Die Hochschule. Für Berufstätige.

Die mit bundesweit über 57.000 Studierenden größte private Hochschule Deutschlands führt seit 1993 Studiengänge für Berufstätige durch, die einen staatlich und international anerkannten Hochschulabschluss (Bachelor/Master) erlangen wollen.

Die FOM ist der anwendungsorientierten Forschung verpflichtet und verfolgt das Ziel, adaptionsfähige Lösungen für betriebliche bzw. wirtschaftsnahe oder gesellschaftliche Problemstellungen zu generieren. Dabei spielt die Verzahnung von Forschung und Lehre eine große Rolle: Kongruent zu den Masterprogrammen sind Institute und KompetenzCentren gegründet worden. Sie geben der Hochschule ein fachliches Profil und eröffnen sowohl Wissenschaftlerinnen und Wissenschaftlern als auch engagierten Studierenden die Gelegenheit, sich aktiv in den Forschungsdiskurs einzubringen.

Weitere Informationen finden Sie unter **fom.de**

Im Forschungsblog werden unter dem Titel „FOM forscht" Beiträge und Interviews rund um aktuelle Forschungsthemen und -aktivitäten der FOM Hochschule veröffentlicht.

Besuchen Sie den Blog unter **fom-blog.de**