Diehl, Matthias; Gurski, Tobias; Zimmermann, Frank

Working Paper

Wartbarkeit von generierten Anwendungen am Beispiel des UML2Tools-Klasseneditors

Arbeitspapiere der Nordakademie, No. 2009-09

Provided in Cooperation with:
Nordakademie - Hochschule der Wirtschaft, Elmshorn

Suggested Citation: Diehl, Matthias; Gurski, Tobias; Zimmermann, Frank (2009) : Wartbarkeit von generierten Anwendungen am Beispiel des UML2Tools-Klasseneditors, Arbeitspapiere der Nordakademie, No. 2009-09, Nordakademie - Hochschule der Wirtschaft, Elmshorn

This Version is available at:
https://hdl.handle.net/10419/38611

Arbeitspapiere der Nordakademie

ISSN 1860-0360

Nr. 2009-09

Wartbarkeit von generierten Anwendungen am Beispiel des UML2Tools–Klasseneditors

Matthias Diehl, Tobias Gurski,
Prof. Dr. Frank Zimmermann

Dezember 2009

Eine elektronische Version dieses Arbeitspapiers ist verfügbar unter:
http://www.nordakademie.de/index.php?id=ap

Köllner Chaussee 11
25337 Elmshorn
http://www.nordakademie.de

Wartbarkeit von generierten Anwendungen am Beispiel des UML2Tools–Klasseneditors

Matthias Diehl
Tobias Gurski
Prof. Dr. Frank Zimmermann

1. Dezember 2009

Zusammenfassung

Das vorliegende Papier befasst sich mit der Evaluierung der Wartbarkeit von generierten Anwendungskomponenten am Beispiel des UML2Tools–Klasseneditors, welcher mittels EMF/GMF modellgetrieben entwickelt wird. Beim MDSD[1]–Ansatz können nicht alle Anforderungen auf Modellebene an die zu generierende Anwendung spezifiziert werden. Spezielle Anforderungen sind innerhalb generierter Anwendungskomponenten manuell zu ergänzen. Untersucht wurden zum einen existierende Erweiterungsmechanismen und ihre Nachteile bezüglich der Wartbarkeit und zum anderen die aspektorientierte Programmiersprache *Object Teams* und wie sie in diesem Anwendungsfall zu einer besseren Modularität und damit Wartbarkeit führen kann. Das Resultat ist eine 100–prozentige Trennung von generiertem und manuell erstelltem Code. Die extrahierten invasiven Änderungen wurden in eine modulare, Feature–orientierte Struktur überführt. Object Teams verfügt über ein Sprachkonzept sowie über Entwurfsmuster, die es ermöglichen wartungsfreundlich manuelle Ergänzungen mit generierten Anwendungskomponenten zu koppeln.

1 Einleitung und Aufbau des Papiers

Das Ziel dieses Papiers ist die Evaluation der Wartbarkeit von generierten und nicht generierten Anwendungskomponenten am Beispiel des UML2Tools–Klasseneditors[2], welcher durch das Eclipse Modeling Framework (EMF) und das Graphical Modeling Framework (GMF) modellgetrieben entwickelt wird. Die Wartbarkeit von generierten Anwendungen ist stark an die Einhaltung der Best Practices für die modellgetriebene Softwareentwicklung geknüpft[3]. Im Rahmen dieses Papiers wird die Object Teams–Technologie vorgestellt und gezeigt inwieweit Object Teams ausgewählte Aspekte der Best Practices umsetzen kann. Ferner wird dargestellt welchen zusätzlichen Beitrag Object Teams zur Erhöhung der Wartbarkeit von generierten Anwendungen zu leisten vermag.

Im Kapitel 2 sind die vier verschiedenen Aktivitäten der Wartung Perfective Maintenance, Adaptive Maintenance, Corrective Maintenance und Preventive Maintenance kurz dargestellt.

Das Zusammenspiel zwischen generierten und nicht generierten Anwendungskomponenten wird im Kapitel 3 beschrieben. Im Kontext des MDSD–Ansatzes haben die verschiedenen Erweiterungsmechanismen (z.B. Nutzung von Extension Points oder Protected Regions) eine unterschiedlich starke Auswirkung auf die Wartbarkeit von generierten Anwendungen.

Im Kapitel 4 wird daher eine Klassifizierung in kritische, problematische bzw. neutrale Erweiterungen vorgenommen.

Anhand von Metriken wird im Kapitel 5 eine Aussage über die Wartbarkeit des generierten UML2Tools–Klasseneditors, insbesondere über das Package–Design, getroffen.

Die Möglichkeit eines Wartungszugewinns wird durch den Einsatz spezieller Design–Pattern der Object Teams–Technologie am Beispiel des UML2Tools–Klasseneditors im Kapitel 6 aufgezeigt.

[1] engl. Model–Driven Software Development

[2] Der UML2Tools–Klasseneditor ist Bestandteil des Eclipse–Projektes Model Development (MDT). Genauere Informationen sind unter http://www.eclipse.org/modeling/mdt/?project=uml2tools zu finden.

[3] **?**, , **?**, und **?**, zeigen eine Reihe verschiedener Best Practices für die modellgetriebene Softwareentwicklung auf.

2 Wartbarkeit von generierten Anwendungen

Der wesentliche Unterschied zwischen generierten und handgeschrieben Anwendungen ist, dass ausführbarer Quelltext aus Modellen generiert wird. Jedoch stoßen komplexe Anforderungen der Anwendungsdomäne oft schnell an die Grenzen der Technologiedomäne. Dieser Sachverhalt wird an einem konkreten Beispiel im Kapitel 3 verdeutlicht. Handgeschriebener Quelltext muss daher verwendet werden, um diese Lücken zu schließen. Die Art der Kopplung von manuellen mit generiertem Code ist für Wartbarkeit von generierten Anwendungen von entscheidender Bedeutung. Daher ist das Verständnis über das Zusammenspiel von generierten und nicht–generierten Quelltextartefakte wichtig. Änderungen an dem Modell haben meist Änderungen an dem handgeschriebenen Quelltext zur Folge.[4]

Die Wartbarkeit von Software ist eines der Hauptziele des Software-Engineerings. `SEIFERT` und `BENEKEN` teilen hierbei die Aktivitäten der Wartung in vier verschiedene Kategorien auf[5]. Von *Perfective Maintenance* sprechen sie, wenn die Software um eine weitere Funktionalität ergänzt werden soll. *Adaptive Maintenance* bedeutet für sie die Anpassung des Systems aufgrund neuer Technologien. Die Kategorie *Corrective Maintenance* bezeichnet die klassische Fehlerbehebung (z.B. durch Wartung oder User-Feedback über die Software) und unter *Preventive Maintenance* verstehen sie Erweiterungen oder Anpassung des Systems unter der Annahme, dass diese Änderungen später notwendig werden. Im Folgenden wird hauptsächlich die Perfective Maintenance sowie Corrective Maintainance in Bezug auf generierte Anwendungen untersucht.

3 Zusammenspiel generierter und nicht generierter Artefakte

Der UML2Tools-Klasseneditor wird unter Verwendung des MDSD-Ansatzes entwickelt. Dieser Ansatz beinhaltet die Modell–zu–Codetransformation (M2C)[6]. Ziel der M2C-Transformation ist die generative Erzeugung plattformspezifischer Quelltextartefakte auf Basis (formaler) Modelle[7]. Im Rahmen dieses Papiers wird unter dem Begriff Quelltextartefakt Programmcode in Form von generierten Klassen verstanden. Eine besondere Bedeutung während einer M2C-Transformation nehmen Templates ein. Templates enthalten die wesentlichen Architekturgrundsätze nach denen Quelltext generiert wird. Manuelle Ergänzungen treten auf, wenn Erweiterungen nicht auf Modellebene abgebildet werden können bzw. eine Modellierung zu hohen Aufwand erzeugen würde[8]. Es besteht eine Koexistenz zwischen den M2C-Artefakten und den manuellen Erweiterungen.

Das Einbinden einer neuen Funktionalität bei MDSD ist meistens mit einer Erweiterung oder Änderung des Modells verbunden. Die Synchronisation zwischen den generierten Komponenten und dem Modell erfolgt durch die M2C-Transformation. Die einseitige M2C-Transformation wird als Forward-Engineering bezeichnet[9]. Dieses ist in einem iterativen Prozess der Softwareentwicklung ein sich häufig wiederholender Vorgang. Der generierte Quelltext ist dabei als Wegwerfprodukt zu sehen, der niemals manuell modifiziert werden sollte[10]. Es gelten für das Zusammenspiel von generierten und nicht generierten Komponenten die folgenden Best Practices:

- Klare Architektur, die definiert, welche Komponenten zu generieren und manuell zu erstellen sind
- Keine Modifikationen am Generat

Die Einhaltung dieser Best Practices ist an die Existenz definierter Schnittstellen geknüpft. Jene sind ausschlaggebend für die wartungsfreundliche Integration von nicht generierten mit generierten Anwen-

[4] Vgl. **?**, S. 245

[5] Vgl. **?**, S. 269 f.

[6] Die Modell–zu–Code–Transformation (M2C) überführt ein Quellmodell in plattformspezifischen Quelltext.

[7] Formale Modelle beschreiben einen bestimmten Aspekt einer Software vollständig. Eindeutige Regeln müssen definiert werden, die festlegen, worüber das Modell Aussagen zu lässt. Formale Modelle sind in **?**, S.31 dargestellt.

[8] Vgl. **?**,

[9] Umgekehrt wird die Transformation von Quelltext nach Modell als Reverse-Engineering bezeichnet. Rountrip-Engineering beschreibt die bidirektionale Synchronisation von Änderungen im zugrundeliegenden Modell sowie im erzeugten Quelltext.

[10] Vgl. **?**, S.159

dungskomponenten. Die Bereistellung von *Interfaces* als wichtiges OO-Konzept[11] sowie die Definition von *Extension Points* als bedeutsamer Bestandteil des Plugin-Konzepts von Eclipse sind zwei mögliche Verfahrensweisen. Ein Verstoß gegen die Best Practices führt dazu, dass das Generat nicht mehr als Wegwerfprodukt behandelt werden kann. Daraus ergibt sich in der Praxis die Notwendigkeit das Generat in die Versionskontroll–Systeme zu importieren und Unterschiede in der Realisierung des Modells im Quelltext-Generat durch den Einsatz sogenannten "Diff"–Tools[12] aufzudecken. Darüber hinaus kann es dazu kommen, dass manuelle Modifikationen im Zuge von Modellerweiterungen oder –anpassungen nicht mehr verwendet werden. Diese "Altlasten" sind schwer zu identifizieren und wirken der Produktivität im Softwareerstellungsprozess entgegen, indem z.B. ständige "Diff"s oder Lokalisieren von Änderungen erforderlich sind.

Die Handlungsempfehlung — keine Modifikationen am Generat vorzunehmen — bedeutet die physische Trennung von generiertem und manuell erzeugtem Programmcode. Dieses ist nicht immer möglich. Komplexe Anforderungen der Anwendungsdomäne lassen sich mit der Technologiedomäne nicht immer umsetzen. Ein Beispiel hierfür sind die Assoziationen der Anwendungsdomäne UML. Sie sind ein komplexeres Konstrukt, welches sich durch die Technologiedomäne EMF/GMF nicht direkt abbilden lässt. Um diesen Sachverhalt zu verdeutlichen, wird die originär generierte Klasse des UML2-Tools-Klasseneditors `AssociationCreateCommand.java` mit ihrem modifizierten Derivat verglichen. Im Quelllisting 1 ist die generierte Methode `doDefaultElementCreation()` dargestellt. Hierbei ist die Annotation *@generated* vor dem Funktionsrumpf zu beachten. Sie signalisiert, dass es sich um einen generierten Klassenbestandteil handelt. Der Methodenzweck ist die Erzeugung einer Assoziationsinstanz sowie das Setzten des Ziel- und Quellmodellelements. Weiterhin wird die instanzierte Assoziation den bisher existierenden Paketelementen hinzugefügt. Zum Schluss gibt die generierte Methode die Assoziation zurück. Der Rückgabeparameter ist das Interface EObject, welches für UML2-Modellierungsobjekte das Basisverhalten definiert.

Listing 1: originär generierte Methode

```
1   /**
2    * @generated
3    */
4   protected EObject doDefaultElementCreation() {
5        Association newElement = UMLFactory.eINSTANCE.createAssociation();
6        getContainer().getPackagedElements().add(newElement);
7        newElement.setType(getSource());
8        newElement.setType(getTarget());
9        return newElement;
10  }
```

Durch manuelle Anpassung der Methode `doDefaultElementCreation()` konnte die anwendungsdomänenspezifische Anforderung zur Gestaltung von UML–konformen Assoziationsenden umgesetzt werden. Das Resultat dieser Modifikation ist im Quelltextlisting 2 dargestellt.

Invasivität:
Alle manuellen Modifikationen an generierten Quelltextartefakten, die einen Eingriff auf Dateiebene darstellen, werden fortfolgend als *invasiv* bezeichnet.

Damit invasive Ergänzungen bei einem erneuten Generatorlauf nicht überschrieben werden, müssen diese geschützt werden. Protected Regions sind vom Generator interpretierbare Bereiche, so dass bei M2C-Transformationen invasiver Quelltext nicht übergeneriert wird. Im Kontext des UML2Tools–Klasseneditors ist das zuvor dargestellte Quelltextlisting 2 ein Beispiel für die Verwendung von Protected Regions. Die Annotation *@generated NOT* sorgt dafür, dass der Methodenrumpf nicht verändert wird. Folgende Probleme können bei dem Einsatz von Protected Regions auftreten [13]:

- Die Verwaltung der Protected Regions obliegt dem Generator. Dadurch erhöht sich die Komplexität des Generators.

[11] OO steht für Objektorientierung
[12] z.B. Tool unterstützt durch kdiff3; Open-Source unter http://kdiff3.sourceforge.net.
[13] Vgl. **?**, S. 160

- Die Trennung zwischen Generat und manuell erzeugtem Quelltext wird verletzt, da beide Typen in demselben Quelltextartefakt(Klasse oder Datei) vorkommen. Somit muss der Softwareentwickler innerhalb von generierten Komponenten arbeiten. Dadurch ist dieser gezwungen die vom Generator interpretierbare Notation und seine Arbeitsweise zu kennen. Seine Aufgabe ist es, Programmcode zu entwickeln, der kompatibel zum iterativen M2C-Transformationsprozess ist. Änderungen am Generat, die nicht als Protected Regions deklariert sind, führen dazu, dass der hinzugefügte Programmcode bei einem erneuten Generatorlauf verloren geht.

Listing 2: invasiv modifizierte Methode

```
1   /**
2   * @generated NOT
3   */
4   protected EObject doDefaultElementCreation() {
5       Type sourceType = (Type) getSource();
6       Type targetType = (Type) getTarget();

8       boolean setNavigability = getCreateRequest().getParameter(
                AssociationEditHelper.PARAMETER_SET_TARGET_NAVIGABILITY) !=
                null;

10      Association result = targetType.createAssociation(false,
11      AggregationKind.NONE_LITERAL, CustomMessages.
12          AssociationCreateCommand_source_end, 1, 1,
13      sourceType, setNavigability, AggregationKind.NONE_LITERAL,
14          CustomMessages.AssociationCreateCommand_target_end, 1, 1);

16      return result;
17    }
```

4 Einstufung und Bewertung der manuellen Erweiterungsmechanismen

Invasive Erweiterungen führen zur Vermischung von generiertem und manuell erstelltem Programmcode innerhalb eines Software-Moduls. Es entsteht ein gemischtes Zielartifakt. Folglich muss das Mischgenerat mitgepflegt werden. Im Folgenden wird der Einfluss einer manuellen Erweiterung auf den iterativen Softwarerstellungsprozess des UML2Tools–Klasseneditors anhand von drei Klassifizierungstypen bewertet. Die nachfolgende Ampelanalogie zeigt die Unterteilung in wartungskritisch, wartungsproblematisch und wartungsneutral.

rot: wartungskritisch

gelb: wartungsproblematisch

grün: wartungsneutral

Die Kritikalität einer manuellen Erweiterung innerhalb generierter Anwendungskomponenten wird an den Kriterien

- Modelltreue und
- Invasivität

bemessen.

Modelltreue:
Im Kontext der Wartbarkeit von generierten Anwendungskomponenten bedeutet Modelltreue, die Sicherstellung, dass alle Modell–Informationen in den generierten Zielartefakten enthalten sind. Folglich müssen sich Änderungen auf Modellebene immer in den laufenden Anwendungen widerspiegeln.

4.1 Wartungskritisch

Alle Erweiterungen, die sowohl invasiv sind, als auch das Prinzip der Modelltreue nicht einhalten, schalten die Wartbarkeitsampel auf rot. Die Modelltreue muss verletzt werden, wenn auf Methodenebene zwei Zustände eintreten. Erstens ist die generierte Methode als Protected Region gekennzeichnet. Zweitens sind an dem M2C-Transformationsprozess verschiedene Modellartefakte beteiligt. Folgendes Templatelisting verdeutlicht diesen Sachverhalt.

Listing 3: Templatesicht auf die Generierung der Methode addSemanticListeners()

```
1  «DEFINE addSemanticListeners FOR gmfgen::GenNode-»
2      //...
3  switch (next.getVisualID()){
4      «FOREACH linksToListen AS nextLink-»
5      «EXPAND xpt::Common::caseVisualID FOR nextLink»
6      getLinkTargetListener().addReferenceListener(nextLink, «EXPAND MetaModel::
                MetaFeature FOR
7      nextLink.modelFacet.targetMetaFeature»);
8          break;

10     «ENDFOREACH-»
11     default:
12         break;
13     }
14 //...
```

Es handelt sich um einen Ausschnitt aus dem Template NodeEditPart.xpt, welches ein Custom–Template des UML2Tools–Klasseneditors ist. [14] Ziel dieses Templates ist die Generierung der Java–Methode `addSemanticListeners()`. Die Methodendefiniton im Template zeigt, dass die Anzahl der Fallunterscheidungen in der generierten Methode abhängig von den betroffen Modellartefakten ist.
Wird die aus der Templatedefintion erzeugte Methode innerhalb des Generats als *generated NOT* gekennzeichnet und auf Modellebene ein Target hinzugefügt oder gelöscht, überträgt sich die Anpassung des Modells nicht in die laufende Anwendung. Im konkreten Fall des UML2Tools–Klasseneditors ist die generierte Methode nicht manuell angepasst worden. Es verdeutlicht aber, dass durchaus Methoden existieren, bei denen invasive Modifikationen zu einer Verletzung der Modeltreue führen können. Diese Modifikationen müssen im Rahmen einer Modelländerung bedingten M2C-Transformation gegebenenfalls auf Verletzung der Modelltreue geprüft werden. Hierzu sind die betroffenen Quelltextstellen zu lokalisieren. Sowohl die Prüfung auf Modelltreue als auch die Lokalisierung sind als Mehraufwand für die Wartbarkeit anzusehen.

4.2 Wartungsproblematisch

Der Generator des UML2Tools–Klasseneditors verfügt über ein besonderes Feature, um die Einhaltung der Modelltreue sicherzustellen. Die Anlage einer Methode mit dem Namen, welcher sich aus der Bezeichnung des generierten Originals und dem Methodensuffix *Gen* zusammensetzt, führt dazu, dass bei erneuter M2C-Transformation in die Gen-Methode hineingeneriert wird. Dieses Generator–Feature ermöglicht bei Verwendung der Gen–Methode innerhalb der originär generierten Methode die Wahrung der Modelltreue. Somit besitzen zusätzliche invasive Erweiterungen in der originären

[14] Templates für die M2C-Transformation werden in GMF mit der Programmiersprache XPand erstellt, die seid der Version GMF 2.2 M4 Bestandteil von GMF sind.

Methode additiven Charakter. Diese invasiven Erweiterung verletzen das Trennungsprinzip von generiertem und nicht–generiertem Programmcode und schaltet die Wartbarkeitsampel auf gelb. Das Quelltextlisting 4 stellt diesen Zusammenhang dar. Anhand dieses Beispiels wird deutlich, dass der Softwareentwickler die vom Generator interpretierbare Gen–Notation verstehen muss.

Listing 4: Workaround für invasive Quelltextmodifikationen

```
1  /**
2    * @generated NOT
3    */
4    protected EObject doDefaultElementCreation() {
5      doDefaultElementCreationGen();
6      // additiver Quelltext
7    /**
8
9    * @generated
10   */
11   protected EObject doDefaultElementCreationGen() {
12     // Quelltext siehe originär generierte Methode
13   }
14 }
```

4.3 Wartungsneutral

Alle manuellen Modifikationen, die die Modelltreue wahren und nicht-invasiv sind, führen zu wartungsfreundlichen grünen Ampelfarbe. `STAHL`, `VOELTER`, `EFTINGE` und `HAASE` zeigen Architekturen wie die *dreistufige Vererbung* (Abb. 1) sowie die *Entwurfsmuster-basierte Integration*[15] auf, die eine nicht invasive Kopplung von manuellen Erweiterungen mit generierten Anwendungskomponenten ermöglicht.

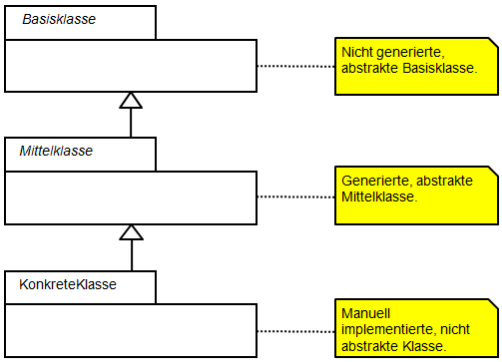

Abbildung 1: Dreistufige Vererbung

Eine auf Java–basierende Zielplattform gestattet den Einsatz der dreistufigen Vererbung. Eine abstrakte Basisklasse implementiert die Funktionalität die für alle Komponenten einer Art gleich ist. Diese Implementierung wird als Teil der Plattform betrachtet. Eine generierte, abstrakte Mittelklasse implementiert alle Funktionalitäten, die aus den Informationen des Modells abgeleitet werden können. Eine manuell implementierte, nicht abstrakte Klasse erbt von der generierten, abstrakten Mittelklasse und implementiert alle Funktionalitäten, die nicht aus dem Modell gewonnen werden können. Jene wird innerhalb der Anwendung verwendet. Die aufzubauende Vererbungshierarchie ist in Abbildung 1 dargestellt.

Die dreistufige Vererbung wird im Rahmen des UML2Tools–Klasseneditors nicht verwendet. Eine weitere Technik zur Erzielung neutraler Erweiterungen ist der Einsatz von Custom-Templates. Es findet

[15] Vgl.?, S.162

eine Verlagerung der manuellen Erweiterungen in zusätzlich erzeugte Templates statt. Manuelle Erweiterungen, die im Eclipse-Umfeld durch das Plugin-Konzept realisiert werden, sind ebenfalls als neutrale Modifikationen zu werten. Pluginspezifische Extension Points können genutzt werden, um nicht invasive Ergänzungen vorzunehmen. Die andockenden Plugins dienen in der Regel einem bestimmten Zweck, so dass sie durch eine hohe Verständlichkeit gekennzeichnet sind. Weiterhin ist das Plugin-Konzept durch eine sehr lose Kopplung bestimmt, welches die Modifizierbarkeit begünstigt. Die Eclipse Service-Plattform und die bereitgestellten GMF Extension-Points bieten eine Vielzahl von Erweiterungsmöglichkeiten. Es ist jedoch zu konstatieren, dass nicht alle Extension-Points von Beginn an berücksichtigt werden können.

5 Wartbarkeit des UML2Tools–Klasseneditors

Im Folgenden wird die Wartbarkeit des UML2Tool-Klasseneditors anhand der Wartbarkeitsaspekte Modifizierbarkeit und Verständlichkeit evaluiert. Hierzu wird zunächst das Package Design des Generats des UML2Tool-Klasseneditors untersucht. Im Fortfolgenden ist unter dem Generat, das generierte Zielartefakt auf Grundlage der Version 0.81 des UML2Tools–Klasseneditors zu verstehen. Die Wartbarkeit der generierten EMF/GMF Anwendungskomponenten ist genau dann von Bedeutung, wenn der generierte Quelltext invasiv modifiziert wird. Somit ist der Entwickler gezwungen, Änderungen im Kontext der vom Generator erzeugten Architektur durchzuführen. Damit eine Aussage bezüglich der Modifizierbarkeit getroffen werden kann, steht im Nachfolgenden die Kopplung zwischen den Paketen sowie die Paketkohäsion des Generats im Vordergrund. Für die Erstellung des Kopplungsgraphen und die Messung der Kopplungsintensität wird das Tool STAN4J genutzt. STAN4J ist ein kommerzielles Strukturanalyse Tool, welches sowohl als Eclipse-Plugin als auch in Form einer Stand-Alone Anwendung verfügbar ist[16].

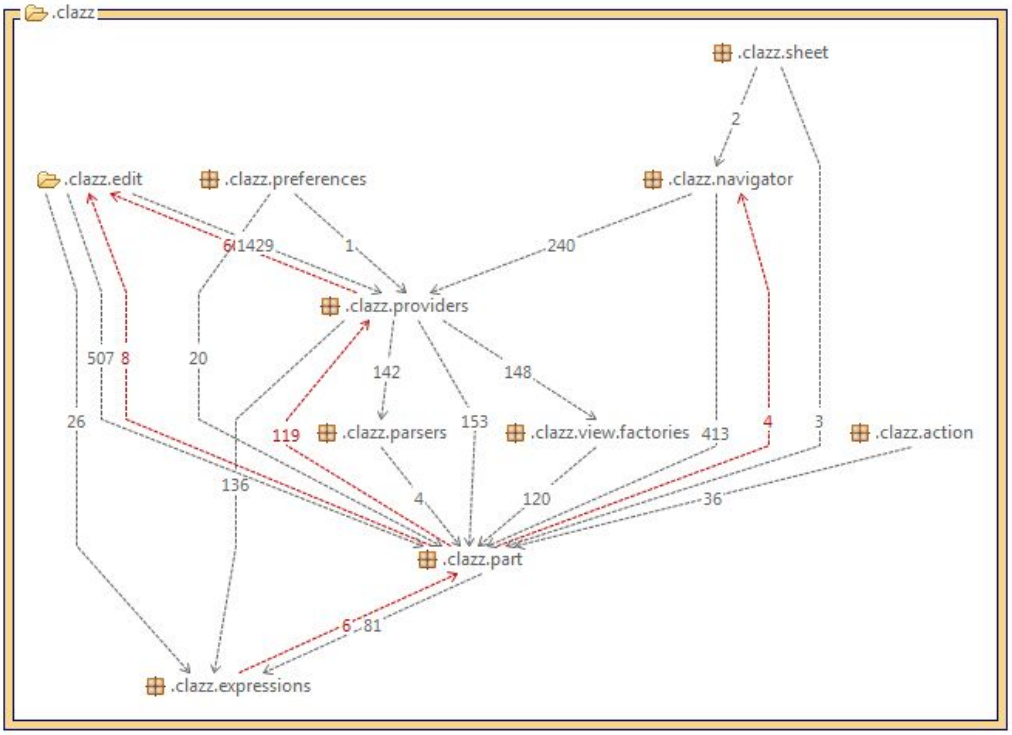

Abbildung 2: Kopplungsgraphen des Generats des UMLToolsKlasseneditors

In Abbildung 2 ist der Kopplungsgraph des UML2Tools–Klasseneditors auf Paket-Ebene dargestellt. Die Zahlen an den Pfeilen geben die direkten und indirekten Beziehungen zwischen den jeweiligen Paketen an. Zum Beispiel hat das Paket `clazz.part` 119 direkte bzw. indirekte Abhängigkeiten zu dem Paket `clazz.providers`. Zu erkennen ist, dass eine starke Kopplung zwischen den Paketen sowie mehrere Zyklen den Kopplungsgraphen prägen. Nach dem *Acylic Dependencies Principle* (ADP) müssen die Abhängigkeiten zwischen Paketen einen azyklisch gerichteten Graphen bilden. Es dürfen keine Kreisbeziehungen vorliegen[17]. Bei zyklischen Paketen ist die Wiederverwendung, die Durchführung von Test sowie die Wartung isoliert nicht möglich. Die Abbildungen 3 und 4 zeigen die Abhängigkeitsbeziehungen der Klassen innerhalb zwei exemplarisch ausgewählter Pakete des UML2Tools–Klasseneditors.

[16] STAN4J ist zu finden unter http://stan4j.com/
[17] Vgl. **?**, S. 18-21

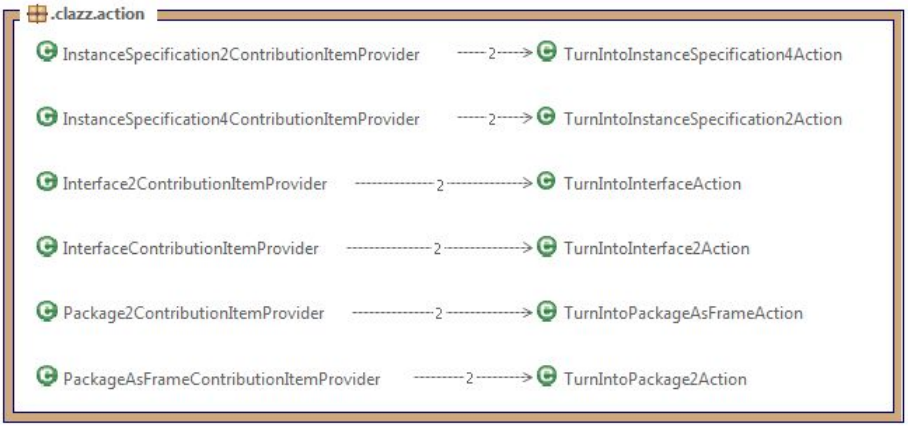

Abbildung 3: Klassen-Kopplungsgraph des Paketes `org.eclipse.uml2.diagram.clazz.action`

Die "Provider"–Klassen des Paketes `.clazz.action` der Abbildung 3 stehen in keiner Beziehung zu einander. Genauso verhält es sich mit den "Action"–Klassen des Paketes.

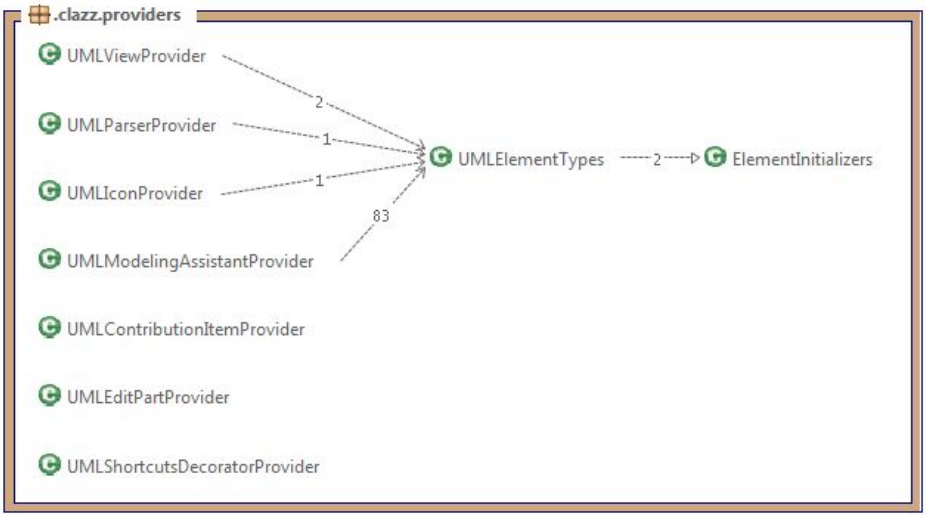

Abbildung 4: Klassen-Kopplungsgraph des Paketes `org.eclipse.uml2.diagram.clazz.providers`

Die Abbildung 4 zeigt, dass die Klassen `UMLContributionItemProvider`, `UMLEditPartProvider` und `UMLShortcutsDecoratorProvider` keine Beziehungen oder Abhängigkeiten zu den anderen Klassen des Paketes `.clazz.Providers` haben.

Die Abhängigkeit von einer Klasse, bedeutet die Abhängigkeit von allen anderen Klassen. Nach dem *Common Reuse Principle* sind nur Klassen innerhalb eines Pakets zu gruppieren, die wirklich enge Klassenbeziehungen aufweisen[18]. Das in den Abbildungen 3 und 4 dargestellte Package Design zeigt eine geringe Paketkohäsion. Die starke Kopplung sowie die geringe Kohäsion lassen auf ein wartungsunfreundliches Package Design des Generats des UML2Tools–Klasseneditors schließen.
Ein Nachweis wird im Folgenden über die Average Component Dependency (ACD) Metrik auf Paket-Ebene geführt[19]. Die Metrik Average Component Dependency (ACD) definiert die Anzahl der Abhängigkeiten eines Paketes C von anderen Paketen. Es werden sowohl die direkten als auch die indirekten

[18] Vgl. **?**, S. 17 f.
[19] **?**, gibt einen guten Überblick über Metriken zur Beurteilung der Wartbarkeit von Software

Paketbeziehungen gezählt. Die relative ACD ist die ermittelte Anzahl der Paketabhängigkeiten im Verhältnis zu allen Paketen[20]. Die Anwendung der relativen ACD-Metrik auf das Generat ergibt einen Prozentsatz von 63,46. Ein relativer ACD-Wert größer als 50 Prozent ist ein Indikator für schwer wartbares Package Desgin[21]. Besonders im Bezug auf die Modifizierbarkeit zeigt der Wert, dass eine Änderung in einem Paket Auswirkung auf 63,46 Prozent der anderen Pakete haben kann.
Es kann bei diesem Package Design auch von einem *technischen Packaging* gesprochen werden. Der Generator verteilt die erzeugten Klassen ihren Aufgaben nach in Pakete. Infolgedessen werden z.B. alle EditParts dem Paket `org.eclipse.uml2.diagram.clazz.edit.parts` und alle Factories dem Paket `org.eclipse.uml2.diagram.clazz.view.factories` zugewiesen. Das technische Packaging ist ausschlaggebend für die starke Kopplung sowie die geringe Kohäsion.
Im Folgenden wird am Beispiel des UML2Tools–Klasseneditors gezeigt, wie sich das technische Packaging auf die Verständlichkeit auswirkt. Im Zuge der Entwicklung des UML2Tools–Klasseneditors wurden 154 Methoden des Generats invasiv modifiziert. Dieses entspricht einem Anteil von ca. 3 Prozent. Die Modifikationen verteilen sich auf 68 der 672 Klassen und 7 der 13 Packages, welches in der folgenden Tabelle 1 dargestellt ist.

Package	**Klassen**	**Methoden**
org.eclipse.uml2.diagram.clazz.edit.commands	14	35
org.eclipse.uml2.diagram.clazz.edit.helpers	1	1
org.eclipse.uml2.diagram.clazz.edit.parts	20	21
org.eclipse.uml2.diagram.clazz.edit.policies	6	9
org.eclipse.uml2.diagram.clazz.part	4	23
org.eclipse.uml2.diagram.clazz.providers	1	43
org.eclipse.uml2.diagram.clazz.view	22	22
Summe	68	154

Tabelle 1: Überblick über alle invasiven Modifikationen

Werden die Modifikationen den zugehörigen Features zugeordnet ergibt sich die Tabelle 2.

Feature:
Ein Feature beschreibt im Rahmen dieses Papiers ein Modellelement in der UML.

[20] Vgl.**?**, S. 60
[21] Im Kapitel ACD and Testability auf http://blog.stan4j.com/ ist die ACD-Metrik als Komplexitätsmetrik genauer erklärt.

Feature	**Package**	**Klassen**	**Methoden**
Classifier	1	1	21
Port	3	6	15
Association	5	12	35
Dependency	3	3	3
Generalisation	6	11	18
Instance Specification	3	3	10
Package	5	5	9
Template Binding	2	2	4
Comprehensive Features	3	5	13

Tabelle 2: Feature-orientierter Überblick über alle invasiven Modifikationen

Die Modifikation des Features Assoziation verteilen sich über 35 Methoden in zwölf Klassen in fünf verschiedenen Paketen. Im Rahmen eines iterativen Softwareentwicklungsprozesses muss der Entwickler über wartungsunfreundlich viele Pakete navigieren, um neue Erweiterungen hinzuzufügen bzw. bereits getätigte zu lokalisieren und ggf. zu ändern. Das Verständnis wird durch das technische Packaging erschwert.
Es wäre nun aber vermessen das Generat des UMLTools-Klasseneditors generell als wartungsunfreundlichen zu bezeichnen. Wird z.B. die ACD-Metrik auf der Klassenebene des Generats angewendet, ergibt sich ein Wert von 9,6 Prozent. Dieses ist ein Indikator für ein sehr wartungsfreundliches Klassendesign.

6 Object Teams — Einsatzmöglichkeiten bei MDSD

Im Rahmen eines Kooperationsprojektes mit der Technischen Universität Berlin wird der UML2Tools-Klasseneditor unter Verwendung der Object Teams-Technologie refactort. ObjectTeams (OT) ist eine aspektorientierte Java–Erweiterung, die an der Technischen Universität Berlin entwickelt wurde. Das von Object Teams eingeführte Sprachkonzept besteht aus Teams und Rollen. Teams modularisieren Kollaborationen mehrerer Klassen. Unter Kollaboration ist eine Zusammenlegung von Klassen, die durch gemeinsame Interaktionen gekennzeichnet sind und einen bestimmten Zweck verfolgen, zu verstehen. Rollen adaptieren das Verhalten externer Klasse[22].

Die Verwendung von Object Teams verfolgt im Rahmen dieser Arbeit das Ziel, die als wartungskritisch und wartungsproblematisch identifizierten Modifikationen in eine modulare feature-orientierte Struktur außerhalb des Generats zu überführen. Als Ausgangspunkt für das Refactoring dient die Version 0.8.1 des UML2Tools–Klasseneditors. Zunächst werden alle invasiven Modifikationen identifiziert und einem Feature zugeordnet. Die Feature-Zuordnung ist für die Überführung der Modifikationen in die Parallelarchitektur von großer Bedeutung.

feature-orientierte Parallenarchitektur:
Die Parallelarchitektur kapselt alle manuellen Ergänzungen. Die manuellen Erweiterungen sind nach Features in Teams strukturiert.

Die feature-orientierte Parallelarchitektur bietet die Möglichkeit alle invasiven Modifikationen in ein Team-Package zu überführen. Das Team-Package beinhaltet Rollen, die die Klassen des Generats adaptieren. Jede Rolle adaptiert dabei genau ein Klasse. Die feature-relevanten Modifikationen sind nun nicht mehr in vielen Packages verteilt, sondern sie werden in einem Team lokalisiert. Das Resultat der Extraktion aller invasiven Änderungen, die das Feature Assoziation betrifft, ist die Parallelarchitektur in Abbildung 5.

[22] Eine detailierte Beschreibung der Object Teams-Technologie ist der Sprachdefinition auf http://www.objectteams.org/def/1.2/index.html zu entnehmen.

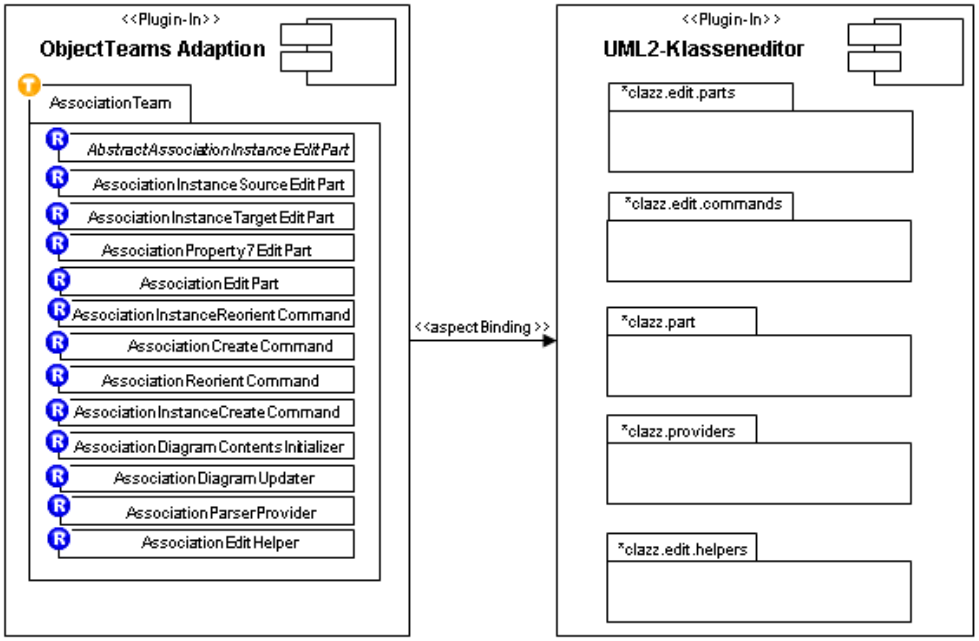

Abbildung 5: feature-orientierte Parallelarchitektur

Es wird die Anforderung zur Einhaltung des Trennungsprinzips von generierten und nicht-generierten Anwendungskomponenten umgesetzt. Die feature-orientierte Parallelarchitektur bietet die Möglichkeit innerhalb des Softwareentwicklungsprozesses die Einteilung der Aufgaben nach Features zu strukturieren. Die Aufgaben können isoliert von einander bearbeitet werden. Bei invasiver Modifikation im Generat dagegen sind die Entwickler auf Grund des technischen Packaging gezwungen zur selben Zeit an gleichen Paketen zu arbeiten sowie ein Feature über mehre Pakete zu implementieren. Dieses birgt ein hohes Konfliktpotenzial (z.B. Merges) — Quelltextabgleiche und ein hoher Koordinationsaufwand sind meist die Folge.

Object Teams bietet eine sehr komfortablen Weg querschnittliche softwarespezifische Belange zu modularisieren. Ein anderes Anwendungsszenario ist das zentralisierte Bug-Fixing. Im Kontext des UML2Tools-Klasseneditor ziehen die Abhängigkeiten der Eclipse-, der EMF/GMF sowie der UML2-Komponenten untereinander oftmals die Realisierung eines Workarounds nach sich. Durch die Object Teams-Technologie können Workarounds nicht-invasiv verortet werden.[23] Ist der Workaround z.B durch eine neue EMF/GMF-Version obsolet geworden, so kann der vorgenommene Workaround OT–seitig abgeschaltet werden. Weiterhin ermöglicht Object Teams durch Einsatz verschiedener Sprachkonstrukte und Design Patterns die Modifikation qualitativ zu verbessern. Im Folgenden werden Techniken vorgestellt, die im Zuge des qualitativen Refactorings des UML2Tools–Klasseneditors Anwendung gefunden haben.

6.1 Basecalls

Basecalls sind ein Mittel zur Wahrung der Modelltreue generierter Anwendungskomponenten durch den Einsatz von Object Teams. Sie sind mit dem in Kapitel 3 eingeführten Konzept der Gen-Methoden vergleichbar. Sie ermöglichen es, dass stetige Modelländerungen innerhalb eines iterativen Software-Entwicklungsprozess in die Anwendung übernommen werden. Quelltextlisting 5 zeigt die Verwendung des generierten Verhaltes der adaptierten Basisklasse `AssociationEditPart.java`, welches durch das Schlüsselwort *base* hervorgehoben ist. Der additive Aufruf der Methode `refreschDecorations` erfolgt

[23] Im Rahmen dieser Arbeit konnten 65 Workarounds, die über 51 verschiedene Klassen verstreut waren, in einem Team gekapselt werden.

im Gegensatz zum Konzept der Gen-Methoden nicht–invasiv. Das Schlüsselwort *replace* ersetzt die originäre Methode durch die adaptierende Methode in der laufenden Anwendung.

Listing 5: Beispiel für ein Basecall aus dem AssociationTeam

```
1    protected class AssociationEditPart playedBy AssociationEditPart {

3    createConnectionFigure <- replace createConnectionFigure;
4    callin Connection createConnectionFigure() {
5        Connection con = base.createConnectionFigure();
6        refreshDecorations((AssociationLinkFigure) con);
7        return con;
8    }
9    }
```

Die Wahrung der Modelltreue durch den Einsatz von Basecalls wird im Rahmen des Papieres als ein Wartbarkeitszugewinn verstanden.

6.2 Baseclass Generalization

Generierte Anwendungskomponenten verfügen oftmals über Quelltextartefakte, die aufgrund der generischen Templates ähnlich bzw. identisch sind. Werden nun manuelle Modifikationen an diesen Artefakten notwendig, so ist der Entwickler bei einem klassischen invasiven Vorgehen gezwungen, gleiche Änderungen an mehreren Stellen im generierten Artefakt durchzuführen. Die invasiv modifizierte Methode `getParserElement()` der Klassen `AssociationInstanceSourceEditPart.java` und `AssociationInstanceSourceEditPart.java` repräsentiert solche eine Konstellation. Durch den Einsatz von Object Teams kann ein Zugewinn an Kohäsion durch die Verwendung einer *Baseclass Generalization* erreicht werden. Die in der feature-orientierten Parallelarchitektur abstrakte Basisklasse kapselt das identische Verhalten. Die Rollen AssociationInstanceSourceEditPart und AssociationInstanceTargetEditPart adaptieren die gleichnamigen Klassen des Generats mit dem Verhalten der Basisrolle. In der Abbildung 6 ist der kohäsive Zugewinn dargestellt.

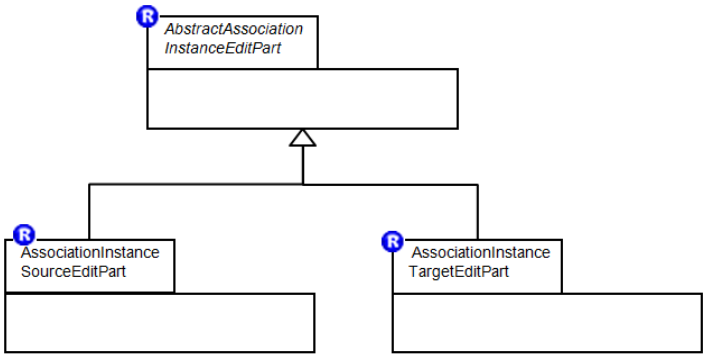

Abbildung 6: Kapselung der identischen Anpassungen in der abstrakten Rolle AbstractAssociationInstanceEditPart

6.3 OMA-Pattern

Das OMA-Pattern (Observer-Mediator-Actuator) findet Anwendung, wenn die Umsetzung einer Wartungsanforderung (z.B. über mehrere Plugin-Grenzen hinweg) entweder

- nur durch unter Verwendung des Kontextes (z.B. vorangegangene Methodenaufrufe, Objektinstanziierung, Parameterübergabe) eines bestimmten Methodenaufrufs umgesetzt werden können, oder

- durch obig eingeleiteten Kontext schneller bzw. eleganter umgesetzt werden können.

Fehler im UML2Tools-Klasseneditor: Beim Ziehen einer AssoziationClass in ein Paket, welches sich in der frame-notation befindet, tritt unter bestimmten Umständen fälschlicherweise eine 'semantic refresh failed'-Exception auf. Dies wird mit dem OMA-Pattern folgendermaßen verhindert:

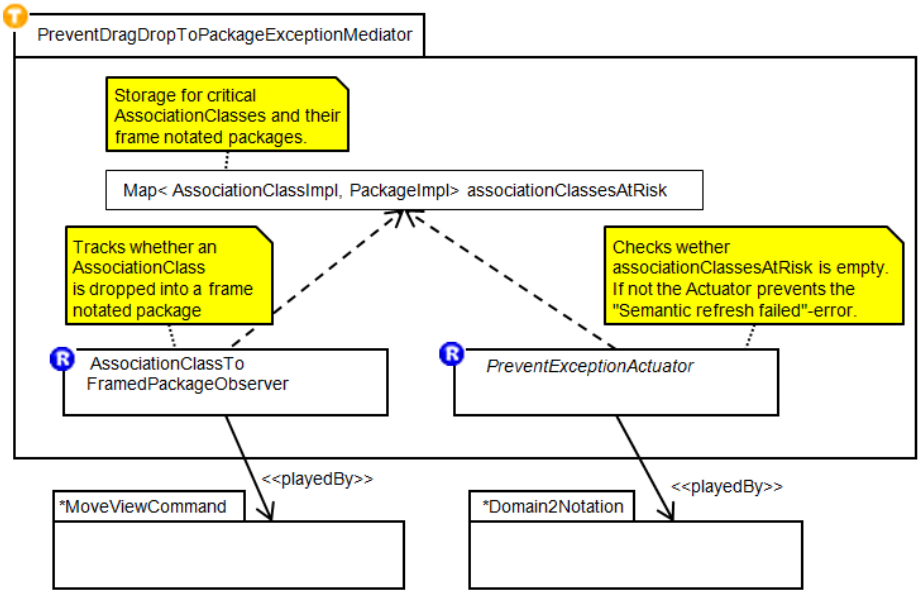

Abbildung 7: Funktionsweise des OMA-Pattern

- Damit ein Methoden-Overwrite nur im Kontext der zuvor in das frame-notated-Package verschobenen AssociationClass stattfindet, stellt das Mediator-Team den Puffer associationClassesAtRisk-Map für die kritischen AssociationClasses bereit.
- Der Observer hat sein Join-Punkt im Common-Plugin (org.eclipse.uml2.diagram.common) und prüft, ob eine AssociationClass in ein frame-notated-package geschoben wurde. Ist dieses der Fall, speichert er das Objekt in der associationClassesAtRisk-Map.
- Der Actuator prüft, ob die associationClassesAtRisk-Map leer ist. Befinden sich Objekte in der Map, wird die Methode `getHinted()` der Klasse `Domain2Notation.java` überschrieben, um das Werfen einer Exception zu verhindern.

Das OMA-Pattern stellt — vor allem im Kontext voneinander abhängiger Eclipse-Plugins wie z.B. GMF/EMF, UML2 und dem UML2Tools-Klasseneditor — im Rahmen einiger Anforderungen nicht nur die einzige Möglichkeit keine invasiven Änderungen an sämtlichen voneinander abhängigen Plugins vorzunehmen dar, sondern kann den Entwicklungs- und damit auch Wartungsprozess stark beschleunigen. Hierbei ist jedoch darauf zu achten, dass unter Anwendung des OMA-Pattern keine unerwünschten Seiteneffekte auftreten.

7 Fazit

In dem Papier wurde gezeigt, dass manuelle Anpassungen bei der modellgetriebenen Entwicklung des UML2Tools-Klasseneditors mit EMF/GMF notwendig sind. Die manuellen Änderungen wurden anhand ihres Einflusses auf den iterativen Wartungsprozess in die Kategorien wartungskritisch, wartungsproblematisch und wartungsneutral unterteilt. Invasive Modifikationen an generiertem Quelltext sind wartungskritisch oder zumindest wartungsproblematisch. Im Rahmen des Refactorings des UML2Tools–Klasseneditors konnten keine wartungskritischen Modifikationen lokalisiert werden. Zur Wahrung der Modelltreue wird in den wartungskritschen Anwendungsbereichen auf das vorgestellte

Generator-Feature zurückgegriffen. Dieses ist ein Beleg dafür, dass die Entwickler des UML2Tools–Klasseneditors auf einem sehr hohen qualitativen MDSD-Niveau arbeiten. Jedoch ergab die Analyse des Package-Designs des UML2Tools–Klasseneditors eine stark technische Ausprägung, welche die Wartbarkeit der invasiven Modifikationen nicht begünstigt. Mit Hilfe der Object Teams-Technologie konnten alle invasiven Modifikationen des UML2Tools–Klasseneditors zu 100% in ein Plugin extrahiert werden. Es wurde eine physische Trennung von Generat und Modifikation erreicht. Folglich kann das Generat wieder als Wegwerfprodukt betrachtet werden. Durch den Aufbau einer feature-orientierten Parallelarchitektur konnte ein Zugewinn auf der Bedeutungsebene erzielt werden. Die Modelltreue konnte durch die Verwendung von Basecalls realisiert werden. Unter Einsatz von eleganten Object Teams Design Pattern gelang es, manuelle Modifikationen in der feature-orientierten Parallelarchitekur kohäsiver zu verorten. Der Ansatz bestehende Modifikationen auszulagern birgt intrinsisch das Problem, die Stärken der Object Teams-Technologie nicht auszuschöpfen. Bei einem Ansatz von vornherein Object Teams einzusetzen, sind weitere elegantere Lösungsansätze sowie ein erhöhter Wartungsnutzen zu erwarten.
Liegen die zu erweiternden Anwendungkomponeten nur als Kompilat vor, ist die Option der invasiven Modifikation nicht gegeben. In diesem Fall bietet die Object Teams–Technologie eine Lösung für die Adaptierung des Kompilats.

Literatur

André Fleischer (2007): Metriken im praktischen Einsatz. ObjektSpektrum,, Nr. 3, 58–62 ⟨URL: ⟩ – Zugriff am 18.05.2009

Baier, A./Bereszewski, M. (2009): Modellbasierte Softwareentwicklung: Status Quo und Ausblick. No address in ⟨URL: ⟩ – Zugriff am 27.03.2009

Beck, C./Stuhr, O. (2008): STAN - Strukturanalyse für Java. JavaSpektrum,, Nr. 5, 44–49 ⟨URL: ⟩ – Zugriff am 18.05.2009

Bettin, J. (2003): Best Practices for Component-Based Development and Model-Driven Architecture. No address in ⟨URL: ⟩ – Zugriff am 21.06.2009

Efftinge, S./Friese, P./Köhnlein, J. (2008): Best Practices für modellgetriebene Softwareentwicklung. Javamagazin,, Nr. 5, 14–20 ⟨URL: ⟩ – Zugriff am 21.06.2009

Martin, R. C. (2000): Design Principles and Design Patterns. No address in ⟨URL: ⟩ – Zugriff am 19.05.2009

Seifert, T./Beneken, G. (2005): Evolution and Maintenance of MDA Applications. In Model-Driven Software Development. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, Springer-11645 /Dig. Serial], ISBN 354025613–X, 269–288

Seifert, T./Beneken, G./Baehr, N. (2004): Engineering long-lived applications using MDA. In IASTED Conf. on Software Engineering and Applications., 241–246

Stahl, T. et al. (2007): Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management. 2. Auflage. Heidelberg: dpunkt ⟨URL: ⟩, ISBN 3898644488