Bünder, Hendrik

Working Paper

A model-driven approach for graphical user interface modernization reusing legacy services

ERCIS Working Paper, No. 30

Provided in Cooperation with:
University of Münster, European Research Center for Information Systems (ERCIS)

Suggested Citation: Bünder, Hendrik (2019) : A model-driven approach for graphical user interface modernization reusing legacy services, ERCIS Working Paper, No. 30, Westfälische Wilhelms-Universität Münster, European Research Center for Information Systems (ERCIS), Münster

This Version is available at:
https://hdl.handle.net/10419/203466

THE IS RESEARCH NETWORK

ERCIS WORKING PAPERS

Working Paper No. 30

A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy

Bünder, Hendrik

ERCIS – European Research Center for Information Systems
Westfälische Wilhelms-Universität Münster
Leonardo-Campus 3, 48149 Münster, Germany
P: +49 (0)251 83-38100 F: +49 (0)251 83-38109
E: info@ercis.org W: http://www.ercis.org/

ISSN 1614-7448

www.ercis.org

Working Papers

ERCIS — European Research Center for Information Systems
Editors: J. Becker, K. Backhaus, M. Dugas, B. Hellingrath,
T. Hoeren, S. Klein, H. Kuchen, U. Müller-Funk, H. Trautmann, G. Vossen

Working Paper No. 30

A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services

Hendrik Bünder

ISSN 1614-7448

cite as: Hendrik Bünder: A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services. In: Working Papers, European Research Center for Information Systems No. 30. Eds.: Becker, J. et al. Münster 2019.

Contents

List of Figures

Working Paper Sketch

Type

Research Report

Title

A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services.

Authors

Hendrik Bünder contact via: buender@itemis.de

Abstract

Software modernization for business applications is often driven by the need for supporting additional frontend channels such as web or mobile. At the same time, business logic encapsulated by services and the underlying persistence implementation should be reused. In contrast to writing new graphical user interfaces (GUI) utilizing the latest programming language or framework, we propose a model-driven software engineering (MDSE) approach for specifying the new graphical user interface. In addition, we utilize model-based reverse engineering (MBRE) to discover, transform and integrate existing services and their data types. By providing support for modeling an anti-corruption layer, existing services can be integrated into the new GUI model without exposing potential design flaws from the legacy system. The model of legacy services and data types combined with the newly modeled graphical user interfaces are processed by transformation and generation processes to create source code for user interface, anti-corruption layer and service calls. Thus, enabling efficient integration of existing services in new GUIs by model-based reverse engineering and ensuring flexibility to quickly adapt new graphical user interface technologies through model-driven engineering techniques in the future. First experience from real-world projects indicates that the introduced approach enables faster creation of new graphical user interfaces by relying on production-proven services and data types.

Keywords

Model-Driven Software Engineering, Model-Based Reverse Engineering, Graphical User Interface Modeling, Web-Based Modeling

1 Introduction

Modernizing legacy software applications is often focused on the graphical user interface, because new frontend channels, such as mobile or web, provide new business opportunities. Further, a modern, user-centric, and compelling graphical user interface assists in delivering sustainable competitive advantages [20]. Consequently, graphical user interfaces are not transformed into a structurally identical representation. Instead, new layouts and interactions specific for the anticipated frontend channel and the inherent constraints and limitations are modeled. To cope with fast changing programming languages, frameworks and frontend channels, a model-driven approach for creating graphical user interfaces improves productivity [15]. Model-driven software engineering techniques, such as transformations and generators, eventually turn models into source code of the target language [24]. Thereby, GUIs can be adapted to the latest technology or frontend channel efficiently by solely exchanging the transformations and generation process.

For designing the static layout of graphical user interface, a variety of sketching tools, such as WindowBuilder [6] or SwingGUIBuilder [16], exist. In addition, there are approaches in research to specify the static and dynamic parts of graphical user interfaces in a technology-agnostic way in accordance with the Object Management Group's (OMG) Model-Driven Architecture (MDA) [9]. While the first is technology-specific and layout oriented, solutions from the second category provide no support for integrating existing services. Further, the GUI modeling approach as introduced by Vanderdonckt [22] requires a variety of models to be compliant with the MDA approach. Thereby, modelers have to oversee many models on a different level of abstraction and ensure their consistency to achieve reasonable results. Moreover, existing approaches often stop at the user interface level and allow only limited connections to existing services and data types. Therefore, data binding and service calls must be implemented manually. In contrast, the approach introduced by this paper provides a single graphical user-interface model that provides means to describe static and dynamic parts of the software solution. Additionally, production proven services and data types establish the foundation for retrieving, processing and storing business application data.

To establish a model of existing services and data types, model-based reverse engineering is utilized. MBRE tools, such as MoDisco [2], support the instantiation of legacy-system models used for documentation, dependency analysis, and migration. Yet, to the best of our knowledge there is no tool enabling the integration of legacy models into forward modeling of graphical user interfaces utilizing an MDSE approach.

Integrating legacy models bears the risk of exposing design flaws to the new model. Further, directly reusing services and data types not exactly matching the requirements of the new graphical user interface is disadvantageous for non-functional aspects, such as performance or maintainability. For example, a legacy service providing a customer object with twenty attributes provides the data required to show name and surname in the new GUI. However, delivering the content of twenty attributes to the frontend when only two are required increases payload and eventually response time, thus negatively effecting user experience. To enable reuse of existing services and data types specific for the anticipated usage an anti-corruption layer as introduced by behavior-driven design [23] is introduced on the model level. Besides fit-for-purpose data types, potential design flaws from the legacy system are encapsulated. Thereby, existing services can be reused while the data types used in the GUI are adapted for the expected usage subsequently ensuring sound performance and maintainability.

The approach introduced by this paper combines the automatic model detection of service models from source code with forward modeling of graphical user interfaces. Services described in arbitrary programming or markup language, such as Java or OpenAPI [19], respectively, are transformed into a technology-independent model. On top of the automatically created legacy services and data types an anti-corruption layer is modeled to encapsulate design flaws from the legacy

model and to provide data types specific for the anticipated GUI. The static and dynamic parts of the GUI model are captured through a WYSIWYG editor that provides a look and feel comparable to sketching tools hiding the meta models complexity, yet, offering mature support for specifying complex GUIs. The graphical user interface model and the inherent references to services and data types from the anti-corruption layer are leveraged to enable full generation of GUI code, data binding, and service calls.

The contribution of this paper is threefold: first, we propose an approach that enables efficient creation of graphical user interfaces built upon legacy services that are well encapsulated. Second, a prototypical *What you see is what you get* (WYSIWYG) editor for capturing the static and dynamic parts of the graphical user interface model is presented. Thirdly, a transformation and generation process is introduced in order to project the abstract GUI and service model to arbitrary programming languages or frameworks.

After presenting related work in Section 2, this paper introduces the model driven approach for applying GUI modernization in Section 3. The paper concludes in Section 4 stating the main findings and future research directions.

2 Related Work

There is a variety of tools for sketching graphical user interfaces that provide simple generation features. Industry-proven solutions, such as WindowBuilder [6] or SwingGUIBuilder [16], focus on modeling the layout of a graphical user interface. Subsequently, a generation process will create the required Java classes implementing the defined GUI. The required logic, e.g., for handling user interactions or switching pages, is left to be implemented manually.

The Inspector GUI modeler [14] enables the specification of user interface models on a textual basis. However, specifying graphical a user interface without a WYSIWG editor is cumbersome. In contrast, the GUI modeling approach introduced by Vanderdonckt [22] enables graphical modeling of layouts and widgets. The approach also requires many other models to be created and maintained to achieve reasonable results. In contrast, our approach provides a WYSIWG editor for specifying the static aspects of a user interface enhanced by the possibility to model dynamics, such as button clicks or service calls. Roubi et al. [18] introduced an approach to generate graphical user interfaces for rich internet applications with a model-view-controller architecture. Contrary to the introduced approach by this paper, the model is very close to the actual source code thereby omitting potential efficiency improvements gained through abstraction [24].

Model-based reverse engineering is used to instantiate models from existing source code. Research in the area of MBRE includes approaches on different architectural levels, such as modernizing databases [10], creating databases based on COBOL programs [7], extracting business rules from Java applications [4], building 3D applications [12] and modernizing web applications [21]. In addition, approaches such as JavaMoPP [11] or MoDisco [2] analyze program code and automatically instantiate models. While the first is specialized for detecting models based on Java code, MoDisco provides broader support and additionally aims at refactoring or migrating existing services. In contrast, the approach introduced by this paper includes existing services instantiated from model-based reverse engineering efforts into a model-driven engineering approach for creating graphical user interfaces. Clavreul et. al. [3] describe an approach to integrate legacy systems based on model-driven engineering techniques. The paper describes an automatic integration between legacy systems on service level based on automatically instantiated models. In contrast, the approach proposed by this paper integrates service models into newly modeled GUI models. Thereby, enabling efficient modernization of legacy systems based on an unchanged service layer.

Reis et al. [17] present a model-based reverse engineering approach for legacy graphical user interfaces. The MBRE solution enables the automatic instantiation of graphical user interface models based on existing source code. In addition, Fleurey et al. [8] introduce an approach to automatically create a platform-independent model (PIM) from existing source code by applying static code analysis and model transformations. From the PIM, a platform-specific model for the target system is created, which is then the basis for generating source code. While both approaches provide means to migrate structurally identical graphical user interface implementations, our approach focuses on creating new graphical user interfaces specific for the envisaged use case and frontend channel while reusing existing services and data types.

Although there is research in the area of model-driven graphical user interface creation as well as in the area of model-based reverse engineering, there is to the best of our knowledge no approach integrating the solutions from the two areas.

3 Model-Driven GUI Modernization

The conceptual and technical foundations for integrating MDSE and MBRE approaches are introduced throughout the course of this section. In addition, a running example illustrates the required steps to reverse engineer an existing service, model an anti-corruption layer, and finally specify the new graphical user interface utilizing the introduced meta model.

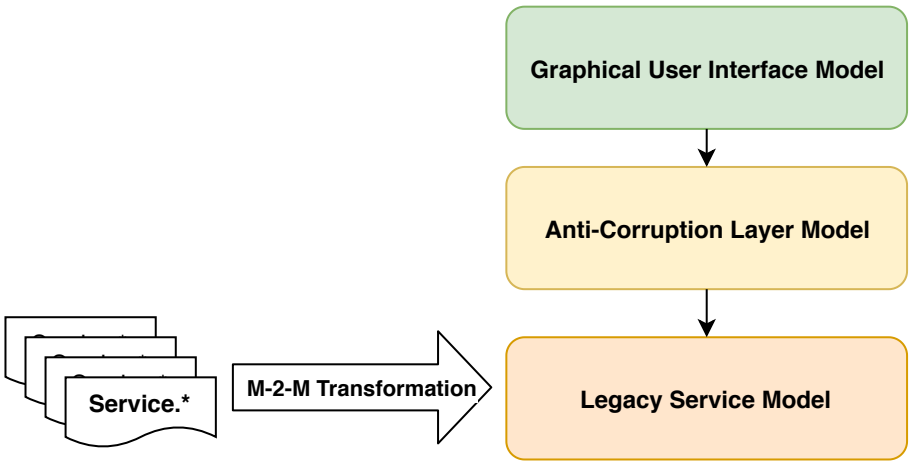

Figure 1: Model-Driven Approach for Modernizing GUIs based on Legacy Services

Conceptual and technical foundations. The modernization approach is divided into three phases. First, the existing services in format of arbitrary source code or markup language are analyzed and transformed into legacy service model instances. As shown by Figure 1 the instantiation of the legacy service model is done completely automatically by a configurable model-to-model transformation process specific for the format of the analyzed legacy services. Second, the legacy service models are encapsulated by a manually modeled anti-corruption layer. Thereby, the services and data types can be re-used without exposing their potential design flaws to the new graphical user interface. In addition, data type are model specific for their envisioned usage eventually improving non-functional requirements such as performance and maintainability. Since the creation of data types and services is partially supported by the modeling tool, the anti-corruption layer modeling is considered semi-automatic. Thirdly, the graphical user interface is modeled using the data types and services defined in the anti-corruption layer as basis. Within the graphical user interface model the layout based on the data types attributes, the service calls,

and the page flows are modeled.

The combination of imported services, anti-corruption layer, and GUI model is input for a transformation and generation process, creating representations of the technology-agnostic model for arbitrary technologies or frameworks, such as JavaScript [25] or Angular [1], respectively. While user interface and service implementation can be generated in the same programming language, the separation of the two models also enables generation of different target technologies. Thereby, service and user interface can evolve separately in terms of implementation technology, while the abstract description remains unchanged.

Running example. To exemplify the process of building a new user interface based on an existing service and its input and output data types, a new web based interface for changing customer master data should be created. In the example application a GUI leveraging the Java Swing GUI framework exists that should be replaced by a browser based dialog to change customer master data.

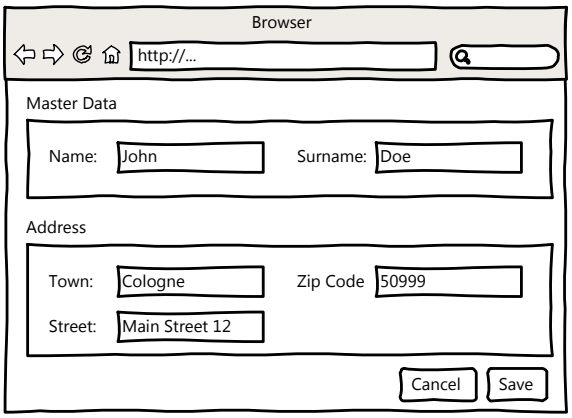

Figure 2: Graphical User Interface Prototype for Customer Master Data Change Dialog

The web-based graphical user interface should show the name and surname as well as the town, zip code and street name of the customer's main address. All fields should be editable and by clicking the save button the changed data should be saved. Figure 2 illustrates the expected GUI that should be integrated into various browser based graphical user interfaces.

3.1 Service Model Discovery

Conceptual and technical foundations. The approach for GUI modernization is designed to provide new graphical user interfaces on top of legacy services and data types. Since existing services are often not available in format of a formal model, the first phase is concerned with instantiating service models. The configurable model-to-model transformation process is adapted to ingest the format, e.g., UML, Cobol or Java service descriptions. Based on the existing format the abstract syntax tree (AST) of the input model is created. Subsequently, the transformation process turns the AST of the existing model into the technology-agnostic model representing the legacy service model. Figure 3 illustrates the adjustments made to the service-model discovery process as represented more generally by Figure 1.

Figure 3 shows the process for instantiating service models from existing Java source code. The model-to-model transformation can be divided into two main parts. The first part utilizes the Eclipse Java Development Toolkit (JDT) [5] to analyze existing Java source code files and to

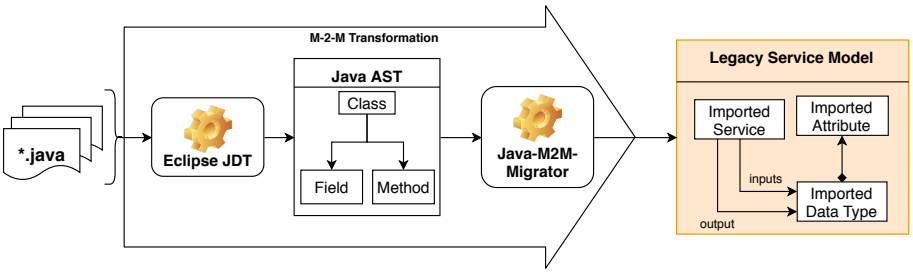

Figure 3: Model-to-Model Transformation for Service Model Detection from Java Source Code

instantiate the respective abstract syntax tree (AST). In addition, the JDT also provides a mature API to analyze and interact with the AST.

Based on the JDT Java API the *Java-M2M-Migrator* transforms the technology-specific service descriptions into technology-agnostic model representations. The migrator was written using the programming language Xtend [26], that offers sophisticated support for graph creation, traversal, and manipulation. Since the Java AST itself holds no knowledge about how services are implemented, the *Java-M2M-Migrator* has to be aware of the semantics within the Java AST. For example not every method of every class is a service, e.g. *toString()* or *hash()* might not be considered a service. Therefore, the *Java-M2M-Migrator* has to understand the specific semantics withing the Java-AST to identify valid services. In the given example, all direct methods of a class implementing the interface *ServiceProvider* are considered for the legacy service model while all methods inherited or overwritten are ignored.

Once the *Java-M2M-Migrator* has identified all valid services, the input and output parameters, as well as their transitive closure, are computed. For each data type used as parameter an *Imported Data Type* with the same name and documentation is instantiated. Additionally, each attribute of the parameters is turned into an *Imported Attribute*. At this point the transition from technology-specific to technology-agnostic representation is performed. While attributes in the Java AST obviously have Java specific data types, such as String, int, or boolean, the legacy service model provides technology-agnostic data types that are further detailed by properties. For example, an *int* from the Java AST will be transformed to a *number* data type with properties specifying maximal and minimal length, digits, etc. As soon as this technology-agnostic attribute is transformed back to a Java representation the technology-specific type will again be *int*. Yet, when the attribute is used in another programming language, e.g. Fortran, it will be translated into an *integer*.

By providing a technology-agnostic representation of the attributes, stubs for arbitrary languages from which the imported service with its imported data types and attributes is used can be generated. In addition, documentation of services and data types is stored and later used for generating system and user documentation for the new system. Further, the imported artifacts hold the technical names of attributes and data types from the legacy system. Thus, enabling generation of advanced mappings beyond simple data type mapping for service calls. Most importantly, legacy services, data types and attributes have to be available in the technology independent model to provide the basis for the graphical user interface models. However, instead of directly using the technology-agnostic representations of the legacy services, an additional anti-corruption layer ensuring encapsulation and cohesion is modeled as explained in Subsection 3.2.

Running example. To exemplify the approach, services written in the programming language Java are analyzed to be transformed into the legacy service model. Consequently, the model-to-model transformation step as shown by Figure 3 is implemented for the Java programming language. The new graphical user interface as described in Section 3 should load the customer

master data based on the Java Services shown in Listing 1.

```
public class CustomerService implements ServiceProvider {
 public Customer getCustomer(CustomerKey id){
  return dbAccess.loadCustomer(id);
 }
 ...
}
```

Listing 1: Legacy Service Implemented in Java

Listing 1 shows parts of a Java class analyzed by the *Java-M2M-Migrator*. In accordance to the semantics, the method `getCustomer` is extracted by the migration process and transformed to an artifact of the technology-agnostic model. The `Customer` object used as parameter in Listing 1 represents the starting point for creating the transitive closure. Figure 4 illustrates the legacy model as instantiated by the *Java-M2M-Migrator*. The transitive closure of the `Customer` object in this example includes the `CustomerKey` as well as the `Address` and their attributes represented as imported data type and imported attributes, respectively.

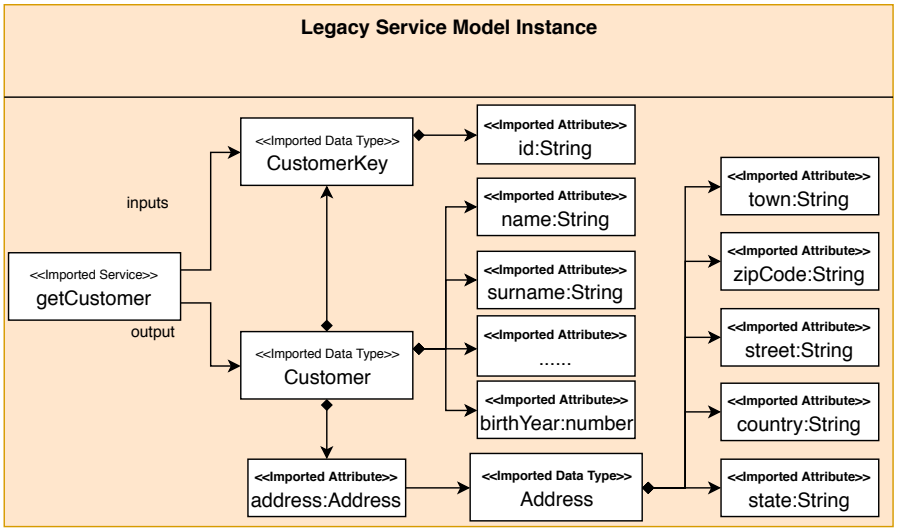

Figure 4: Legacy Model Instantiated by Java-M2M-Migrator

For the example use case the `name` and `surname` attributes of the `Customer` object are required. The `birthYear` attribute exemplified how an int attribute in the Java class from the legacy system is transformed into an imported attribute of type number. The properties further specifying this attribute to enable transformation back to the correct integer format for the Java language, are not part of the illustration in Figure 4. Further, from the `Address` data type only `town`, `zipCode`, and `street` are required.

3.2 Anti-Corruption Layer Modeling

The automatic import of services and their data types into the legacy service model is the basis for all further modeling. However, since software source code and the API deteriorate over time as explained by Martin et al. [13], the introduced approach avoids using the services directly. In

order to encapsulate inherent design flaws of legacy services, the second phase of the introduced approach is concerned with modeling an anti-corruption layer. Thereby, a dedicated part of the model converts existing legacy data types and attributes to the internal data types and attributes of the new model.

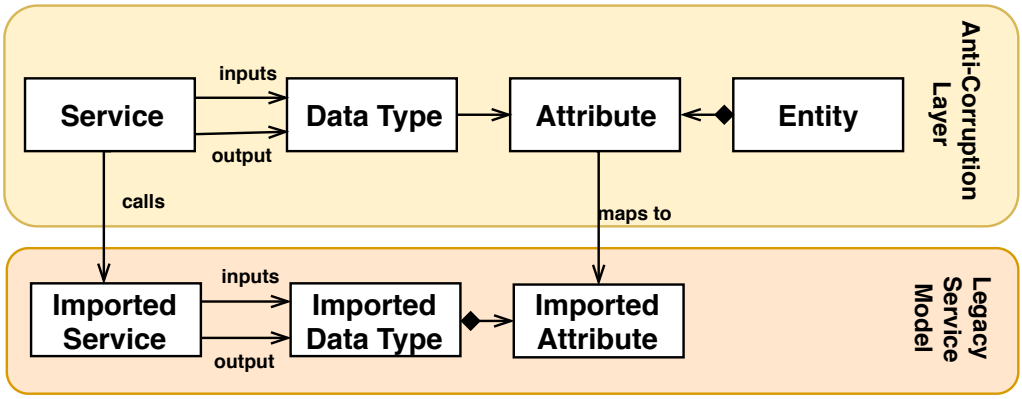

Figure 5: Anti-Corruption Layer Modeling for Integrating Legacy Services

Figure 5 illustrates a meta model used for modeling the anti-corruption layer from which the anticipated implementation is generated eventually. On the lower part of Figure 5 the artifacts created from the legacy system as explained in Section 3.1 are shown. In the middle part the meta model elements representing the anti-corruption layer model are illustrated. To avoid mapping the same attribute multiple times, attribute definitions were separated from their usage through *DataTypes* by introducing two concepts. First, in the anti-corruption layer an *Entity* class contains all attributes and relations that logically belong together, e.g. all attributes of the entity *Customer* or *Address*. The mapping is then specified between attributes of imported data types and the entities attributes. Second, we define *data types* in the anti-corruption layer model that do not hold own attributes, but rather reference attributes aggregated by *Entity* classes. These data types are then used as input and output for services or as basis for graphical user interface descriptions. By separating the attributes from their usage scenarios mapping is done once and the mapping information can be leveraged at multiple occasions. Thereby, maintenance efforts for mapping definitions are decreased and the cohesion between legacy system and new model is increased.

Based on the modeled artifacts and relations different parts of the anti-corruption layer are generated. Obviously, the specified “calls” reference enables the generation of service calls from the anti-corruption layer to the legacy system. In addition, the model contains all information required to map input and output data types. Since the mapping is between imported data type and entity attributes, the definition of data types in the new anti-corruption layer model is decoupled from their mapping. In contrast, if a data type mapping per data type is required, modelers will tend to model rather large new data structures to only specify one mapping and reuse the large data structure. Thereby, increased payloads for service calls negatively affect performance and subsequently user experience when interacting with the generated application. The introduced approach eliminates this restriction.

More complex mappings of attributes with different format can also be mapped using the anti-corruption layer modeling. For example the legacy system might hold and transport date fields as String of eight characters while the new system handles dates as objects of a class Date. The introduced approach not only generates a data type mapping, but also enables the generation of data format transformations, e.g. from String to date object and vice versa.

Besides data and data format mapping, the information included in the anti-corruption layer can be used to map attributes in additional usage scenarios, e.g., for exceptions and their messages. In

the legacy system, exception messages may contain a reference to a specific attribute of the input data type. Based on this information, a GUI framework is able to highlight the widget that is bound to the attribute contained in the exception message that contains a wrong or unexpected value. To provide the same sophisticated user feedback in the new graphical user interface, the attribute references included in the exceptions must be mapped to the attributes of the anti-corruption layer model.

Based on the attribute mapping provided by the anti-corruption layer, transformation and generations can provide mapping code not only for attributes used in services but also for attributes used in exception messages. As mentioned above the mapping is done between the imported type and the attributes of the entity which are then reused for different purposes, such as describing the GUI. This implementation of the ”Don’t repeat yourself” principle allows to generate the mapping for exception messages.

Running example. After instantiating the legacy service model in the previous section the example continues with specifying the anti-corruption layer. The modeling activities can be divided into two phases. First, the instance of *Entity, Attribute*, and *Service* are tailored for the requirements of the graphical user interface modeled in the next step. Second, the mapping information is provided to state how attributes from the legacy model are mapped to attributes from the anti-corruption layer. In addition, it is specified which legacy services are called by the new graphical user interface specific services.

```
1  entity Customer{
2    id: String
3    givenName : String
4    surname : String
5    mainAddress : Address
6  }
7  entity Address{
8    town : String
9    zipCode : String
10   street : String
11 }
12 datatype CustomerWithAddress{
13   from Customer(givenName, surname)
14   from Adress(town, zipCode, street)
15 }
16 datatype CustomerKey{
17   from Customer(id)
18 }
```

Listing 2: Modeling Anti-Corruption Layer Classes

Listing 2 shows a textual representation of the *Entity* and *DataType* objects modeled to encapsulate mapping information and provide data for the specific graphical user interface. To simplify the example, the entities contain only attributes that are required by the graphical user interface as described by Figure 2. The *DataType* objects do not hold own attributes, but rather state which attributes they use by specifying the *from* keyword followed by a reference to the respective *Entity* and the reused attributes within the parentheses.

Listing 3 shows the mapping information and service calls from new to existing services. The mapping statement starts with the keyword *map* followed by the keyword *legacy* and a colon. After the colon, a reference to the imported data type from the legacy model is given. Followed by the $->$ symbol and the class from the anti-corruption layer introduced by the keyword *acl* followed by a colon is specified. Within the curly braces the mapping for each attribute with the

legacy attribute on the left and the new model attribute on the right side of the $->$ is given. Two specialties worth mentioning in this example are the mapping of “name” from the legacy system to “givenName” in the anti-corruption layer model and “address” to “mainAddress”. Thereby, it is exemplified how attributes from the legacy model can be mapped to new names in the anti-corruption layer model.

```
map legacy:Customer -> acl:Customer {
name -> givenName
surname -> surname
address -> mainAddress
}

map legacy:Address -> acl:Address {
town -> town
zipCode -> zipCode
street -> street
}

service CustomerWithAdress loadCustomerWithAdress(CustomerKey)
        calls legacy:getCustomer
```

Listing 3: Specifying Mapping Information

In addition to the mapping of legacy data types to anti-corruption layer entities, Listing 3 exemplifies the definition of a service and its “calls” relation to the legacy model. Based on the previously stated mapping information, the transformation and generation process will identify the required mapping for the input and output data types of the newly modeled service in accordance with the called legacy service. Besides concisely defining calls of existing services, the underlying models can be validated to verify that the used data types are properly mapped to be used to call the respective service.

While the mapping information will be used by transformation and generation processes to automatically create the source code for the mapping layer, the modeled data types serve as basis for the graphical user interface.

3.3 Graphical User Interface Modeling

Conceptual and technical foundations. As shown by Figure 6 the graphical user interface metamodel is built on top of the meta model for modeling an anti-corruption layer for legacy services. The graphical user interface model describes the static structure as well as the dynamic interactions between user and graphical user interface.

The static design of the graphical user interface specifying the arrangement of widgets representing the data-type attributes is captured by *Layouts*. Besides specifying their position it is also possible to alter the widget’s labels or their types, such as *ComboBox* or *RadioButtonGroup*, for enumeration attributes. Again, the ”Don’t repeat yourself” principle is implemented by not copying the attribute but rather referencing it. Thereby, the data model of the user interface is described on the same basis as the data types used in the services to transport the data at runtime. Consequently, data binding and data propagation to the services can be generated completely. In addition, buttons for specifying user interaction with the GUI are placed on the layout.

Layouts are wrapped by *Visuals* that represent either panels or whole pages. While a panel *Visual* holds a direct reference to a *Layout*, a page *Visual* references an arbitrary number of

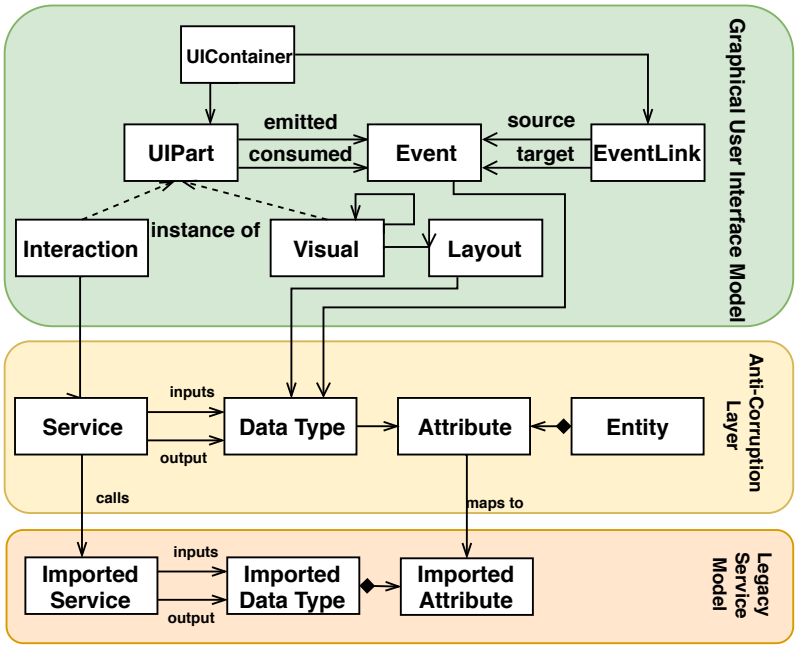

Figure 6: Graphical User Interface Modeling based on Legacy Services

Visuals. Thus, graphical user interfaces with multiple panels can be realized by combining and nesting *UI-Parts*. Multi-page graphical user interfaces are modeled using *UI-Containers* that hold an arbitrary number of *UI-Parts*. In that case, the *UI-Container* represents one user interface for a specific use case. Depending on the requirements one user interface consists of an arbitrary number of pages, panels and layouts. The compositions of user interfaces for creating a complete application handling multiple use cases is beyond the scope of this paper.

Besides structuring the graphical user interface statically, specialized *Visuals*, such as *ViewerVisual* or *ListerVisual*, encapsulate insights about the envisaged usage of the *Layout*. While the first one ensures that all fields are read-only, the second holds additional properties to specify, e.g., if the list is represented as table or list as well as if in-line editing is supported. By separating the static layout information from the anticipated usage, *Layouts* can be reused and the additional semantics of visuals can be utilized by transformation and generation processes to increase the proportion of generated code.

Besides *Visuals*, *Interactions* encapsulate service calls and make their input and output data types available in the user interface model. Both share the parent class *UI-Part*. The dynamics of the user interface are encapsulated in so called *Events* that are consumed or emitted by *UI-Parts*. An *Event* is emitted by a user interaction, such as a click on a button, or by a service returning from processing data. Every event holds a reference to a data type. While the emitted relation between *UIPart* and *Event* describes those *Events* created by the *UI-Part*, the consumed relation describes the *Events* on which occurrence the *UI-Part* reacts.

An *Interaction* calling a service for example consumes an *Event* emitted by a *Visual* representing a button on the GUI. The emitted *Event* holds a reference to the input data types of the called service. In addition, the same interaction emits an *Event* referencing the data type returned by the called service. In order to describe the dynamics of the graphical user interface components, *EventLinks* are modeled that connect *Events*. For a single service call there can be multiple *EventLinks*, e.g., if the service to be called when a button is clicked takes three different parameters that are all represented by different Visuals, there are three *EventLinks* connecting each

consumed *Event* of the service call *Interaction* with each emitted *Event* of the three *Visuals* on the respective GUI. By providing fine-grained connection between *Visuals* and *Interactions* flexibility to support various use cases for visualizing and processing data is ensured.

As Figure 6 shows *EventLinks* are held by the UIContainer. Consequently, the UIContainer as a representation of one graphical user interface, specifies the internal dynamics as well as the *EventLinks* required to switch to other graphical user interfaces. In addition, the *UI-Container* bundles an arbitrary number of *Visuals* and *Interactions* that form one graphical user interface.

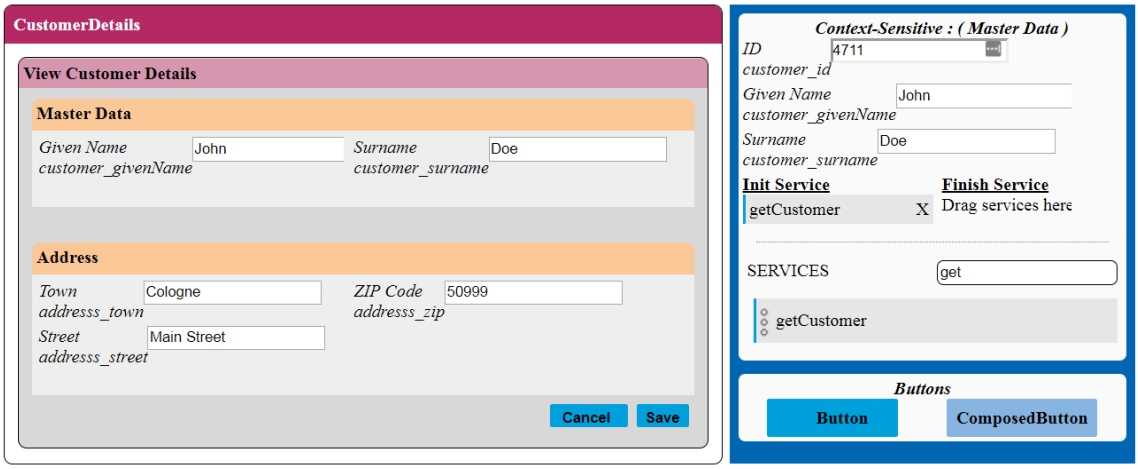

Figure 7: WYSIWYG Editor for Graphical User Interface Modeling

Running example. Figure 7 shows the *What you see is what you get* (WYSIWYG) editor for specifying the graphical user interface that is implemented as browser based application. The left part represents the static parts of a single-page graphical user interface. It shows the user interface for viewing customers master data that consist of one page called ”View Customer Details” that itself consists of two panels ”Master Data” and ”Address”. Within the first panel two widgets are shown, that are representations of the `name` and `surname` attributes as modeled within the `CustomerWithAddress` data type above. The layout in the second panel represents the static layout of the address related attributes, namely `town`, `zipCode`, and `street`. On the right hand side the blue panel shows in the top area the attributes from the data type used as basis for the selected layout. In this case the first panel is selected, therefore the attributes of the customer master data are shown.

In addition to showing the available attributes, the blue panel also encapsulate the information of dynamics aspect, e.g., the *Interactions* required to call a service. In order to specify how the widgets shown on a panel or page are initialized with data, the service used to load the required data has to be referenced. Due do the fact that currently the “Master Data” editor *Visual* is selected (cf. Figure 7), there are two possible services shown, namely, “Init Service” and “Finish Service”. While the first should return the data in accordance to the data type used for the panel, the second should take the same data type to process the potentially changed data type. Next to the “Services” label a search field is provided that takes the entered name and searches the whole anti-corruption layer for services matching the given name. Afterwards the modeler can drag and drop a service listed underneath either to init or finish service area. Thereby, a variety of *Interactions*, *Events*, and *EventLinks* are created by the WYSIWYG editor in accordance to the metamodel. If the user needs to specify more complex meta model concepts, e.g., additional *EventLinks*, another finer grained user interface is available. However, for most of the cases the modeling features of the WYSIWYG editor are feasible for specifying data-driven graphical user interfaces. In general, the WYSIWYG editor hides the complexity of the underlying meta models from the user and enables the modeling of user interfaces by domain-experts that do not require

programming skills.

3.4 Transformation and Generation

Conceptual and technical foundations. Based on the combined legacy service, anti-corruption layer, and GUI model a transformation and generation process is implemented to automatically create the required source code. Instead of directly generating source code from the three models specified before, the technology-agnostic models are transformed into technology-specific models. The model-to-model transformation process as illustrated by Figure 8 automatically creates the instances of the technology-specific models based on parameters, such as a "technical project name". In contrast to MDA approaches, the technical models are read-only and must not be further edited. All customizing is encapsulated in the transformations that might be parameterized by configuration parameters. Technical models are persisted and a browser based GUI to explore these models is provided. The extra effort of introducing an additional layer of models is out weighted by the inherent benefits, namely having a model of the implementation and simplifying the implementation of generators. With a formal model of the technical implementation, the analysis can be executed on model level, e.g., to identify models using certain implementations that are about to be changed. More importantly, the technical models ease the implementation of model-to-text transformations. By relying on pre-calculated values within the technical model, generators are independent of the technology-agnostic models. The vertical separation of technical models enables changing only one aspect of the architecture, e.g., the service implementation for a certain technology-agnostic model. In addition, vertically separated models can evolve independently of each other. Thereby, the approach itself is maintainable and extensible.

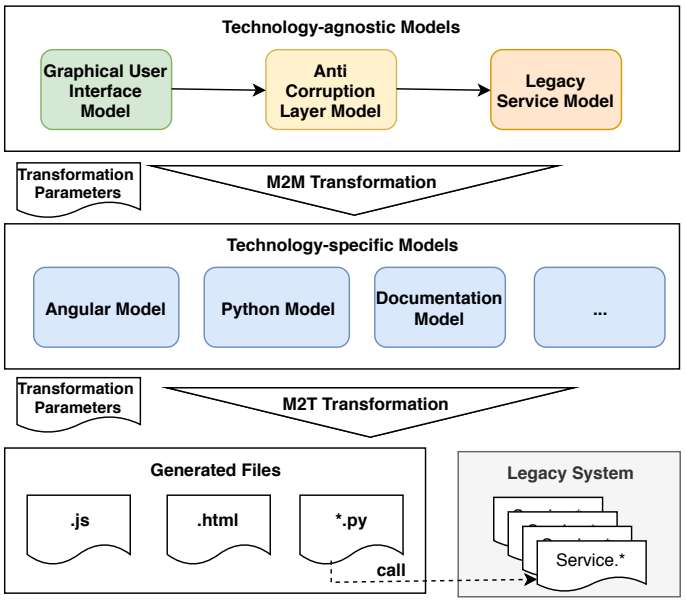

Figure 8: Graphical User Interface Modeling based on Legacy Services

Based on the technology-specific models generators for each technology are implemented that create the required source code. Like the model-to-model transformation, the model-to-text transformation can be adapted using transformation parameters as shown by Figure 8. While the graphical user interface component exemplified by the ".js" file in Figure 8 communicates with a specific backend written in python it does not require any knowledge about the legacy system. Subsequently, calls to services implemented by the legacy system are fully generated including mapping from data types known in the user interface (based on the ACL types) to the legacy

system (based on the legacy system model). The .html files constitute the system and user documentation that is deduced from information available in the legacy model as well as in the graphical user interface model.

Running example. Based on the graphical user interface model a web application utilizing the Angular framework is generated. The layout of the single page application is in accordance with the Layout as specified in Figure 7. The data model of the Angular frontend components is generated based on the data types underlying the *Visuals*, such as the "CustomerWithAddress" data type. In addition, a server side REST endpoint returning the data type by internally calling the legacy service and mapping the returned value is generated.

```
def getCustomer(customerKey)
    serviceResult = callJavaService('getCustomer',{'id':customerKey})
    customer = Customer()
    customer.id=serviceResult['customer']['id']
    customer.givenName=serviceResult['customer']['name']
    ...
    customer.address.town=serviceResult['customer']['address']['town']
    return customer
```

Listing 4: Legacy Service Call Implemented in Python

Listing 4 exemplifies the source code generated behind the REST endpoint on the server side at the boundary between newly generated python component and the legacy Java service. With the function `getCustomer` a framework function is called that takes the name of the legacy Java Service and the parameters as inputs. The parameters of the called Java service are given as dictionary with the key being the name of the parameter and the value being the newly modeled data type. Within this function, the call to the Java service as well as the data type mapping from new model to legacy model is encapsulated.

The return value of the framework method contains the data as returned by the Java service in JavaScript Object Notation (JSON) format. Based on the mapping information as specified in Listing 3 the result of that function is mapped to the object of type "Customer" as expected by the frontend. The implemented generator approach in accordance with the basic idea of MDSE as introduced by Stahl and Völter [24] creates schematic repetitive code, e.g., "callJavaService" that refers to the platform-specific functionality completed by individual code "getCustomer" to specify the individual service to be called or the generated mapping code as shown by Figure 4.

In addition to the JavaScript and python sources, the model-to-text transformation also creates user and system documentation partially based on the documentation of the legacy system. Thereby, additional efficiency improvements are realized by automatically providing documentation for the new graphical user interface model

4 Conclusion and Outlook

The introduced approach provides an automatic mechanisms to instantiate models of legacy services and their data types, a semi-automatic mechanism to model an anti-corruption layer encapsulating design flaws, and a sophisticated WYSIWYG editor for forward modeling of static and dynamic parts of a graphical user interface. The conceptual approach for instantiating service models based on legacy source code was exemplified by a model-to-model transformation starting from existing Java source code. The anti-corruption layer build on-top of the legacy model

ensures integration of existing services and data types without polluting the new model. In addition, the separation of mapping and usage of imported legacy types fosters modelers to provide high quality models tailored for the envisaged usage and not with a prematurely optimized mapping. Finally, the graphical user interface model provides state-of-the-art concepts to specify static and dynamic parts of the user interface building upon existing data types and services.

The configurable service discovery based on model-to-model transformation enables the dynamic adaption to arbitrary service descriptions. First practical experience has been gained with transforming existing services described by their Java source code. Due to sophisticated tooling, namely the Eclipse JDT, the implementation of transformation algorithms for automatically instantiating legacy service models could be realized in reasonable time. However, additional research is required to analyze the effects of larger input models on the overall performance of the AST instantiation and the model creation.

One key concept of the anti-corruption layer modeling is the separation of mapping attributes aggregated by *Entity* objects and specifying data types referencing those attributes to be used in services and graphical user interfaces. First practical experience has shown that this separation leads to mapped attributes being used in six different new data types on average. This strongly indicates, that the new data types are modeled based on their envisaged usage instead of the required effort to implement a mapping. In addition, the introduced anti-corruption layer model enables the creation of data types fit for the anticipated purpose. Thereby, communication payload is reduced positively affecting the response times and subsequently improving user experience. At the same time, attributes are reused and mapping information has to be specified only once to automatically provide mapping for data, data format and additional use case, such as mapping of exception messages for providing mature usability in graphical user interfaces.

The meta model for modeling graphical user interfaces provides means to specify static and dynamic parts of a data driven business applications. By providing an event based paradigm for connecting *Visuals* and *Interactions*, the approach is sufficiently flexible to model GUIs for data driven business applications. The WYSIWYG editor hides the complexity of the meta model by providing GUI designer that looks like typical sketching tools on first site. Yet, the integration of dynamic aspects in format of buttons and service calls enables holistic modeling of graphical user interfaces.

The transformation and generation process enables the automatic creation of technology-specific models that are vertically separated. Thereby, technology-specific models can evolve separately and additional models can be added at any time. Applying the approach in real world projects has shown that large portions of GUI code can be generated based on the abstract models. Nevertheless, additional case studies are required to further investigate the potential of the generative approach.

To ensure high quality of the modeled service calls a concept is required to validate that the legacy services called by the newly modeled services from the anti-corruption layer provide all information required for newly modeled data types. This becomes especially complex when the parameters have deeply nested data types with partially different attributes. Although, the approach has been adapted in three real-world projects, additional case studies are required to examine the benefits of the approach.

In summary, the introduced approach, the WYSIWYG editor and the transformation and generation process enable effective creation of new user interfaces based on existing legacy services due to consequently reducing redundant modeling by following the don't repeat yourself principle. The positive results from the first real world projects indicate the economic advantage that has to be investigated by further case studies. The introduced approach leverages the combination of MDSE and MBRE to quickly exploit new frontend technologies and channels by utilizing on model-based engineering techniques.

References

[1] Angular. Angular - One framework. Mobile and desktop.

[2] Hugo Brunelière, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012 – 1032, 2014.

[3] Mickael Clavreul, Olivier Barais, and Jean-Marc Jézéquel. Integrating legacy systems with mde. *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering and ICSE Workshops*, 2, 01 2010.

[4] Valerio Cosentino, Jordi Cabot, Patrick Albert, Philippe Bauquel, and Jacques Perronnet. A model driven reverse engineering framework for extracting business rules out of a java application. In *Proceedings of the 6th International Conference on Rules on the Web: Research and Applications*, RuleML'12, pages 17–31, Berlin, Heidelberg, 2012. Springer-Verlag.

[5] Eclipse. Eclipse java development tools.

[6] Eclipse. Windowbuilder - is a powerful and easy to use bi-directional java gui designer.

[7] Omar El Beggar, Bousetta Brahim, and Taoufiq Gadi. Getting relational database from legacy data-mdre approach. *Computer Engineering and Intelligent Systems*, 4:10–32, 01 2013.

[8] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. Model-driven engineering for software migration in a large industrial context. In *International Conference on Model Driven Engineering Languages and Systems*, pages 482–497. Springer, 2007.

[9] Object Management Group. Mda - the architecture of choice for a changing world.

[10] Ignacio Guzmán, Macario Polo, and Mario Piattini. Obtaining web services from relational databases. pages 304–310, 01 2006.

[11] Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and java. volume 5969, pages 374–383, 01 2009.

[12] Bernhard Jung, Matthias Lenk, and Arnd Vitzthum. Structured development of 3d applications: round-trip engineering in interdisciplinary teams. *Computer Science - Research and Development*, 30(3):285–301, Aug 2015.

[13] Robert C. Martin and Micah Martin. *Agile principles, patterns, and practices in C#*. Robert C. Martin series. Prentice Hall, Upper Saddle River, NJ and Munich, 10. print edition, 2015.

[14] Thomas Memmel and Harald Reiterer. Inspector: Interactive ui specification tool. *Presented at: 7th International Conference On Computer Aided Design of User Interfaces (CADUI) 2008, June 11-13, 2008, Albacete. Spain*, 01 2009.

[15] Rahul Mohan and Vinay Kulkarni. Model driven development of graphical user interfaces for enterprise business applications – experience, lessons learnt and a way forward. volume 5795, pages 307–321, 10 2009.

[16] Netbeans. Swing gui builder.

[17] André Reis and Alberto Silva. Xis-reverse: A model-driven reverse engineering approach for legacy information systems. pages 196–207, 01 2017.

[18] Sarra Roubi, Mohammed Erramdani, and Samir Mbarki. A model driven approach for generating graphical user interface for mvc rich internet application. *Computer and Information Science*, 9, 04 2016.

[19] Swagger.io. What is openapi.

[20] David Sward. User experience design: A strategy for competitive advantage. volume 1, page 163, 01 2007.

[21] Feliu Trias, Valeria de Castro, Marcos Lopez-Sanz, and Esperanza Marcos. Migrating traditional web applications to cms-based web applications. *Electronic Notes in Theoretical Computer Science*, 314:23 – 44, 2015. CLEI 2014, the XL Latin American Conference in Informatic.

[22] Jean Vanderdonckt. Model-driven engineering of user interfaces: Promises, successes, failures, and challenges. *Proceedings of ROCHI*, 8:32, 2008.

[23] Vaughn Vernon. *Domain-driven design distilled*. Addison-Wesley, Boston, 2016.

[24] Markus Völter and Thomas Stahl. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Series in Software Design Patterns. John Wiley & Sons, New York, NY, 2013.

[25] W3C. Javascript tutorial.

[26] Xtend. Xtend.

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004.

Nr. 2 Teubner, R. A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evaluation Practice. 2005.

Nr. 3 Teubner, R. A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. 2007.

Nr. 6 Teubner, R.; Mocker, M.: A Literature Overview on Strategic Information Management. 2007.

Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.

Nr. 8 Hagemann, S.; Vossen, G.: Web-Wide Application Customization: The Case of Mashups. 2010.

Nr. 9 Majchrzak, T.; Jakubiec, A.; Lablans, M.; Ükert, F.: Evaluating Mobile Ambient Assisted Living Devices and Web 2.0 Technology for a Better Social Integration. 2010.

Nr. 10 Majchrzak, T.; Kuchen, H: Muggl: The Muenster Generator of Glass-box Test Cases. 2011.

Nr. 11 Becker, J.; Beverungen, D.; Delfmann, P.; Räckers, M.: Network e-Volution. 2011.

Nr. 12 Teubner, A.; Pellengahr, A.; Mocker, M.: The IT Strategy Divide: Professional Practice and Academic Debate. 2012.

Nr. 13 Niehaves, B.; Köffer, S.; Ortbach, K.; Katschewitz, S.: Towards an IT consumerization theory: A theory and practice review. 2012

Nr. 14 Stahl, F., Schomm, F., Vossen, G.: Marketplaces for Data: An initial Survey. 2012.

Nr. 15 Becker, J.; Matzner, M. (Eds.).: Promoting Business Process Management Excellence in Russia. 2012.

Nr. 16 Teubner, R.; Pellengahr, A.: State of and Perspectives for IS Strategy Research. 2013.

Nr. 18 Stahl, F.; Schomm, F.; Vossen, G.: The Data Marketplace Survey Revisited.2014.

Nr. 19 Dillon, S.; Vossen, G.: SaaS Cloud Computing in Small and Medium Enterprises: A Comparison between Germany and New Zealand. 2015.

Nr. 20 Stahl, F.; Godde, A.; Hagedorn, B.; Köpcke, B.; Rehberger, M.; Vossen, G.: Implementing the WiPo Architecture. 2014.

Nr. 21 Pflanzl, N.; Bergener, K.; Stein, A.; Vossen, G.: Information Systems Freshmen Teaching: Case Experience from Day One (Pre-Version of the publication in the International Journal of Information and Operations Management Education (IJIOME)). 2014.

Nr. 22 Teubner, A.; Diederich, S.: Managerial Challenges in IT Programmes: Evidence from Multiple Case Study Research. 2015.

Nr. 23 Vomfell, L.; Stahl, F.; Schomm, F.; Vossen, G.: A Classification Framework for Data Marketplaces. 2015.

Nr. 24 Stahl, F.; Schomm, F.; Vomfell, L.; Vossen, G.: Marketplaces for Digital Data: Quo Vadis?. 2015.

Nr. 25 Caballero, R.; von Hof, V.; Montenegro, M.; Kuchen, H.: A Program Transformation for Converting Java Assertions into Controlflow Statements. 2016.

Nr. 26 Foegen, K.; von Hof, V.; Kuchen, H.: Attributed Grammars for Detecting Spring Configuration Errors. 2015.

Nr. 27 Lehmann, D.; Fekete, D.; Vossen, G.: Technology Selection for Big Data and Analytical Applications. 2016.

Nr. 28 Trautmann, H; Vossen, G.; Homann, L.; Carnein, M; Kraume, K; Challenges of Data Management and Analytics in Omni-Channel CRM. 2017

Nr. 29 Rieger, C.: A Data Model Inference Algorithm for Schemaless Process Modeling. 2016.

Nr. 30 Bünder, H.: A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services. 2019.