

# Multi Front-End Engineering

Goetz Botterweck

**Abstract.** Multi Front-End Engineering (MFE) deals with the design of multiple consistent user interfaces (UI) for one application. One of the main challenges is the conflict between commonality (all front-ends access the same application core) and variability (multiple front-ends on different platforms). This can be overcome by extending techniques from model-driven user interface engineering. We present the MANTRA approach, where the common structure of all interfaces of an application is modelled in an abstract UI model (AUI) annotated with temporal constraints on interaction tasks. Based on these constraints we adapt the AUI, e.g., to tailor presentation units and dialogue structures for a particular platform. We use model transformations to derive concrete, platform-specific UI models (CUI) and implementation code. The presented approach generates working prototypes for three platforms (GUI, web, mobile) integrated with an application core via web service protocols. In addition to static evaluation such prototypes facilitate early functional evaluations by practical use cases.

## 1 Introduction

Multi Front-End Engineering (MFE) deals with the systematic design and implementation of multiple consistent user interfaces for one application.

One of the main challenges in MFE is the inherent conflict between commonality and variability. On the one hand, all front-ends provide access to the same application core. Hence, they share a common structure and provide similar functionality. On the other hand, each front-end has to take into account the specifics of the particular user interface platform. Hence, each front-end has to be adapted to these specific characteristics, e.g., when grouping interaction elements into logical presentation units of varying sizes.

---

Goetz Botterweck

Lero – The Irish Software Engineering Research Centre  
University of Limerick, Limerick, Ireland  
e-mail: [goetz.botterweck@lero.ie](mailto:goetz.botterweck@lero.ie)

The challenges that arise from this conflict between commonality and variability can be overcome by adapting and extending techniques from model-driven user interface engineering. To support multiple user interfaces, however, we have to prepare by providing additional information that can guide an automatic or semi-automatic adaptation, for instance to take into account platform-dependent characteristics.

To illustrate how this can be done, we present the MANTRA<sup>1</sup> approach, where the abstract structure of all user interfaces of an application is modelled in an abstract UI model (AUI). This model is annotated with temporal constraints on the dialogue flow and the relative order of interaction tasks. Based on this information, we are able to adapt the user interface on an abstract level, for instance, by deriving and tailoring dialogue structures, which take into account constraints imposed by the particular user interface platform. The adaptation includes the clustering into presentation units and the insertion of control-oriented interaction elements. Based on this abstract model, we use model-to-model transformations to derive concrete, platform-specific UI models (CUI). Subsequently, we use model-to-text transformations to generate implementation code.

The presented approach is realised as a set of Eclipse-based tools and model transformations in ATL (Atlas Transformation Language). It generates working prototypes for three platforms: desktop GUI applications, dynamic web sites and mobile applications. These prototypes are integrated with an application core via web service protocols. Because of the *functional* integration with the application core, in addition to the evaluation of the static user interface (composition, layout and visual presentation of interaction elements), such prototypes also facilitate functional evaluations by performing and analysing practical use cases.

The remainder of this chapter is structured as follows: Section 2 summarises related work, Section 3 gives an overview of the presented approach, Section 4 explains the use of Abstract User Interface (AUI) models in the context of MFE, Section 5 deals with the adaptation of user interfaces on the AUI level, Section 6 describes the transformation from AUI to Concrete User Interface (CUI) Models and the generation of implementation artefacts, Section 7 explains the meta-models of the modelling languages used by our approach, and Section 8 concludes the chapter by discussing the presented approach and future work.

## 2 Related Work

Calvary et al. [1] define a reference framework for multi-target user interfaces. This is also known as the Cameleon reference framework after the European project of the same name. Calvary et al. define the challenges of “multi-targeting” and “plasticity”, which are related to the problem addressed in this chapter. Also, the processes and artefacts in our approach are structured on abstraction levels similar to the Cameleon framework (see Section 3 for an overview).

---

<sup>1</sup> MANTRA stands for Model-driven Approach to UI-Engineering with Transformations.

The *mapping problem* [2] is the problem of defining mappings between abstract and concrete elements is one of the fundamental challenges in model-based approaches to User Interface Engineering. This challenge can occur in various forms and can be dealt with by various types of approaches [3]. One instance of this is the question of how we can identify concrete interaction *elements* that match a given abstract element and other constraints [4].

A similar challenge is the derivation of *structures* in a new model based on information given in another existing model. Many task-oriented approaches use requirements given by the task model to determine UI structures; for example, temporal constraints similar to the ones in our approach have been used to derive the structure of an AUI [5] or dialogue model [6].

Florins et al. [7] take an interesting perspective on a similar problem by discussing rules for splitting existing presentations into smaller ones. That approach combines information from the abstract user interface and the underlying task model - similar to our approach using an AUI annotated with temporal constraints which are also derived from a task model.




Many model-driven approaches to UI engineering have proposed a hierarchical organisation of interaction elements, which are grouped together into logical units. For instance, Eisenstein et al. [8] use such a structure when they aim to support designers in building user interfaces across several platforms.


A number of approaches to multiple user interfaces has been collected in a book edited by Seffah and Javahery [9].

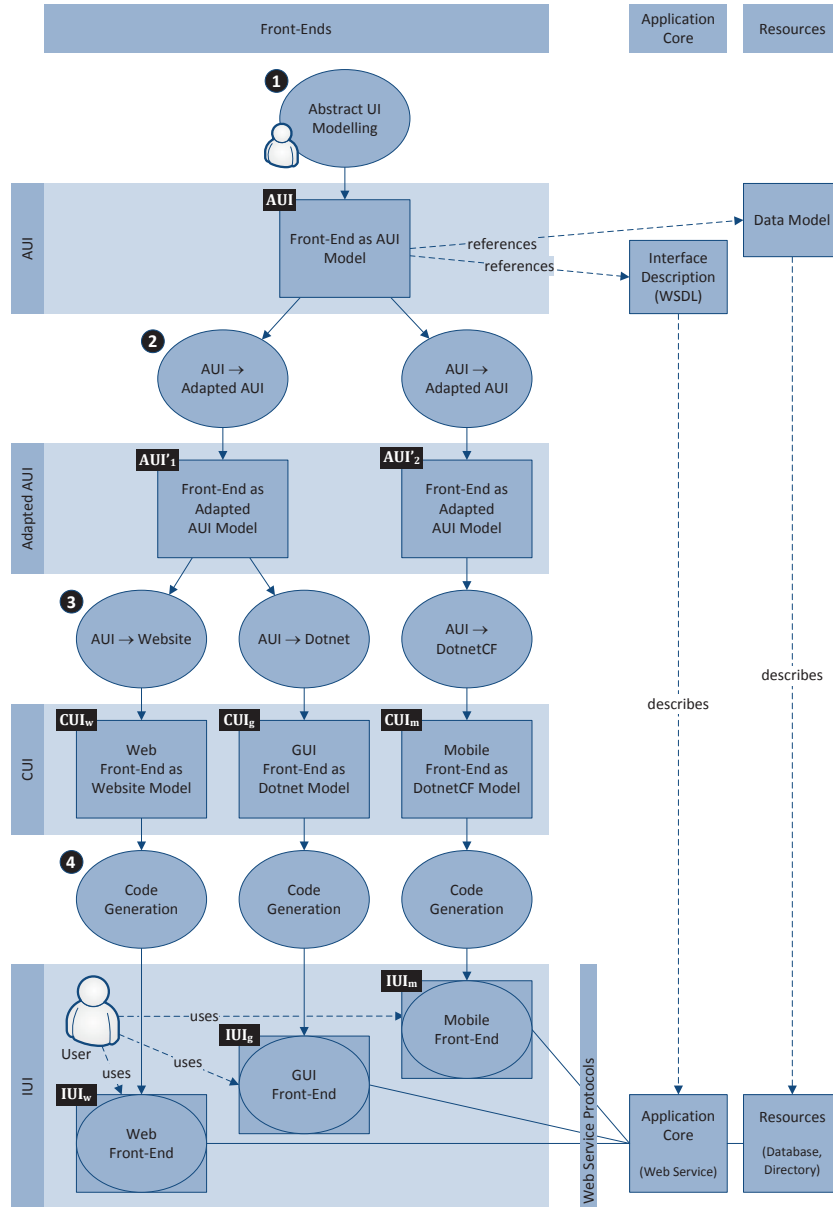
Earlier work on MANTRA, the approach discussed in this chapter, has been presented in [10]. In current work [11] we adapt and specialise techniques taken from the MANTRA approach to support the configuration, derivation and tailoring of user interfaces for products in a product line.

### 3 Overview of the MANTRA Approach

Horizontally, the MANTRA approach (see Fig. 1) is structured by three tiers, *Front-Ends*, *Application Core*, and *Resources*. Vertically, the MANTRA activities and created artefacts are structured by abstraction levels similar to the CAMELEON framework [1]. They include Abstract User Interface (AUI), Adapted Abstract User Interface (Adapted AUI), Concrete User Interface (CUI), and Implemented User Interface (IUI).

The ultimate goal of MANTRA is to create multiple implemented front-ends of the application (see the , , and  at the bottom of Fig. 1). These front-ends are based on different platforms, but provide access to the same *Application Core* and indirectly to *Resources*, which are used by the application.

The MANTRA approach starts with the activity of *Abstract UI Modelling* ❶, which creates an abstract user interface . Then, the interface is adapted on an abstract level ❷. The resulting adapted AUI models are then transformed ❸ into concrete platform-specific UI models (CUI) for three different platforms (Web,



**Fig. 1** Overview of the artefacts and data flow between them in the MANTRA approach

GUI, Mobile). Finally, the desired front-ends (implemented UI, IUI) are created by generating platform-specific code ④.

The following sections will explain these activities and the processed artefacts in more detail.

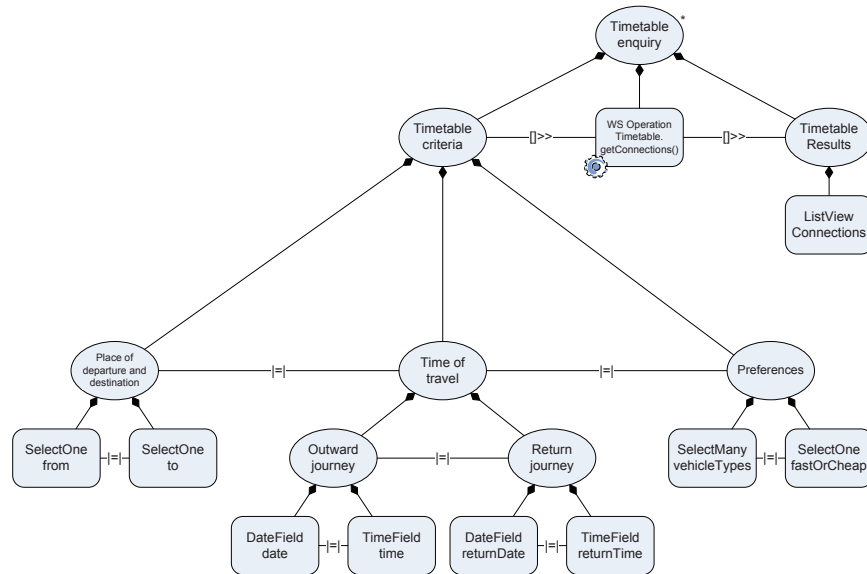
## 4 Abstract Modelling of User Interfaces

We will now explain the subsequent activities of the MANTRA approach in more detail. For the illustration and discussion we will use a simple time table application.




We start of with the activity of *Abstract UI Modelling* ❶, which creates an abstract user interface **AUI**. Fig. 2 shows the corresponding AUI model of our sample timetable application. The user can search for train and bus connections by specifying several search criteria like departure and destination locations, time of travel or the preferred means of transportation (lower part of Fig. 2).

Please note that in the context of the overall MANTRA approach the AUI model, which describes the front-ends of the application, references concepts in models describing other parts or aspects of the system (see AUI layer in the overview in Fig. 1). For instance, abstract UI elements can refer to web service operations to describe integration with the application core or data types defined in a data model.

In our sample shown in Fig. 2 one of the nodes refers to a web service operation *Timetable.getConnections()*, which retrieves connections from the *Timetable* web services and provides them to the subsequent presentation *Timetable Results* for display.



**Fig. 2** AUI model of the sample application annotated with temporal constraints (horizontal lines)

At first, the AUI model only contains *UI Elements* (  ) and *UI Composites* (  ) organised in a simple composition hierarchy (indicated by  relations) and the web service operation necessary to retrieve the results. This model is the starting point of our approach (see **AUI** in Fig. 1) and captures the common


essence of the multiple user interfaces of the application in one abstract UI. This AUI contains platform-independent interaction concepts like “Select one element from a list” or “Enter a date”.

As an input for later adaptation techniques, the AUI is then further annotated by dialogue flow constraints based on the temporal relationships of the ConcurTaskTree approach [12]. For instance we can describe that two interaction elements have to be processed sequentially ( $\gg$ ) or can be processed in any order ( $\mid$ ).

## 5 Adaptation of Abstract User Interfaces


As a next step ( ② in Fig. 1) we augment the AUI by deriving dialogue and presentation structures. These structures are still platform-independent. However, they can be adapted and tailored to take into account constraints imposed, for instance, by platforms with limited display size or by inexperienced users. The result of this process step, the adapted AUI model, is shown in Fig. 3.

### 5.1 Clustering Interaction Elements to Generate Presentation Units



To derive this adapted AUI model we cluster UI elements by identifying suitable UI composites. The subtrees starting at these nodes will become coherent presentations in the user interface (  ). For instance we decided that “Time of Travel” and all UI elements below it will be presented coherently. This first automatic clustering is done by heuristics based on metrics like the number of UI elements in each presentation or the nesting level of grouping elements.

To further optimise the results the clustering can be refined by the human designer by an interactive editors that operates on the adapted AUI model.

### 5.2 Inserting Control-Oriented Interaction Elements

Secondly, we generate the navigation elements necessary to traverse between the presentations identified in the preceding step. For this we create triggers (  ). These are abstract interaction elements which can start an operation (*OperationTrigger*) or the transition to a different presentation (*NavigationTrigger*). In graphical interfaces these can be represented as buttons, menus, or hyperlinks. In other front-ends they could for instance be implemented as speech commands.

To generate *NavigationTriggers* in a presentation  $p$  we calculate *dialogueSuccessors*( $p$ ) which is the set of all presentations which can “come next” if we observe the temporal constraints. We can then create *NavigationTriggers* (and related Transitions) so that the user can reach all presentations in *dialogueSuccessors*( $p$ ). In addition to this we have to generate *OperationTriggers* for all presentations which will trigger a web service operation, e.g., “Search” to retrieve matching train connections (see the lower right corner of Fig. 3).

These two adaptation steps (derivation of presentations, insertion of triggers) are implemented as ATL model transformations. These transformations augment the AUI with dialogue structures (e.g., presentations  and transitions ) which determine the paths a user can take through our application.

It is important to note that the dialogue structures are *not* fully determined by the AUI. Instead, we can adapt the AUI according to the requirements and create different variants of it (see the two adapted AUI models resulting from step 2 in Fig. 1). For instance, we could create more (but smaller) presentations to facilitate viewing on a mobile device – or we could decide to have large coherent presentations, taking the risk that the user has to do lots of scrolling if restricted to a small screen.

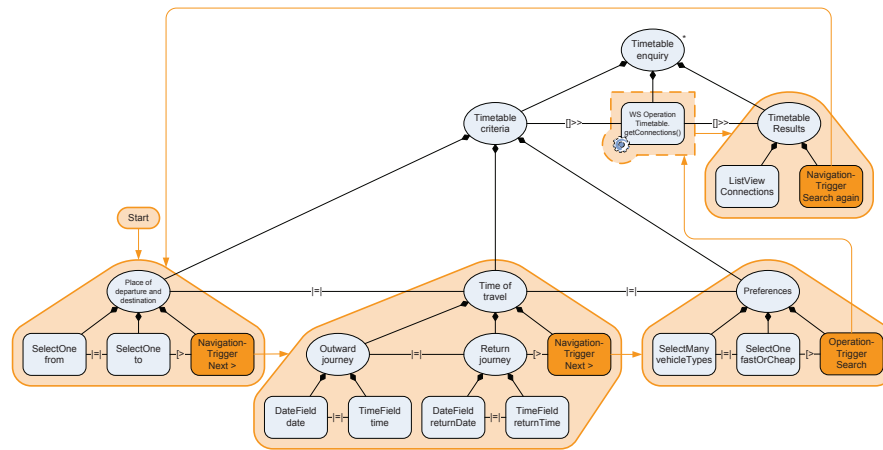



Fig. 3 Adopted AUI model with generated presentations and triggers

### 5.3 Selecting Content

To reflect limitations of the platform, e.g., limited screen estate, which only allows to show a limited number of interaction elements, can apply an additional adaptation step that filters content retrieved from the web service based on priorities.

For instance, if a user has a choice, higher priority is given to knowing when the train is leaving and where it is going before discovering whether it has a restaurant. This optional information can be factored out to separate “more details” presentations.

A similar concept are substitution rules which provide alternative representations for reoccurring content. A train, for example, might be designated as InterCityExpress, ICE, or by a graphical symbol based on the train category (for instance,  to indicate a luxury train) depending on how much display space is available. These priorities and substitution rules are domain knowledge which cannot be inferred from other models. The necessary information can, for instance be modelled as annotations to the underlying data model.

## 6 Generating Concrete Interface Models and Their Implementation

Subsequently, we transform the adapted AUI models into several CUIs using a specialised model transformation (③ in Fig. 1) for each target platform. These transformations encapsulate the knowledge of how the abstract interaction elements are best transformed into platform-specific concepts. Hence, they can be reused for other applications over and over again.

As a result we get platform-specific CUI models. These artefacts are still represented and handled as models, but now use platform-specific concepts like “HTML-Submit-Button” or “.NET GroupBox”. This makes them more suitable to use them as a basis for the code generation (④ in Fig. 1), which produces the implementations of the desired user interfaces in platform-typical programming or markup languages.

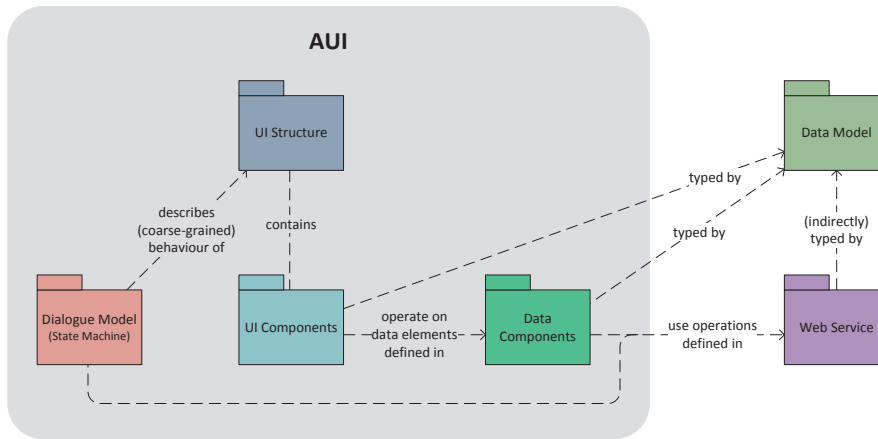


Fig. 4 Overview of the AUI metamodel

## 7 Modelling Languages

In this section we will present the modelling languages (i.e., meta-models) used by the MANTRA approach. We will focus on the AUI level, where the common aspects of all front-ends are represented and differences between platforms are abstracted away. Fig. 4 shows an overview of the involved packages.

First, we will now introduce modelling elements that describe the overall user interface structure. Then, we will then focus on modelling elements for describing the dialogue structure. Finally, we will show how user interface elements can be bound to data components and how these data components are bound to the application core.



## 7.1 User Interface Structure

Fig. 5 shows a simplified excerpt from the AUI metamodel with metaclasses to describe UI structure (on the left) and metaclasses to describe dialogue structures (on the right). Please note that the sections “UI Structure” and “Dialogue” correspond to packages in the metamodel, which was shown earlier overview in Fig. 4.

The core structure of a user interface is given by the composition hierarchy of the various user interface components. In the AUI metamodel this is modelled by a “Composite” design pattern [13] consisting of the classes *UIComponent*, *UIElement*, and *UIComposite* (see ❶ in Fig. 4).

There are two types of *UIComponents*: The first subtype are *UIElements* (see ❷ in the metamodel in Fig. 5) which cannot contain further *UIComponents*. Hence, they become the “leaves” of the hierarchy tree (see the □ symbols in the Timetable sample in Fig. 2). Subclasses of *UIElement* can be used to describe various abstract interaction tasks, such as the editing of a simple string value (*InputField*) or the selection of one value from a list (*SelectOne*). A special case of *UIElements* are Triggers which can start the transition to another presentation (*NavigationTrigger*) or start a (potential data modifying) transaction (*TransactionTrigger*). Please note that the AUI modelling language contains many more *UIElement* subclasses, but they have been omitted here to simplify the illustration.

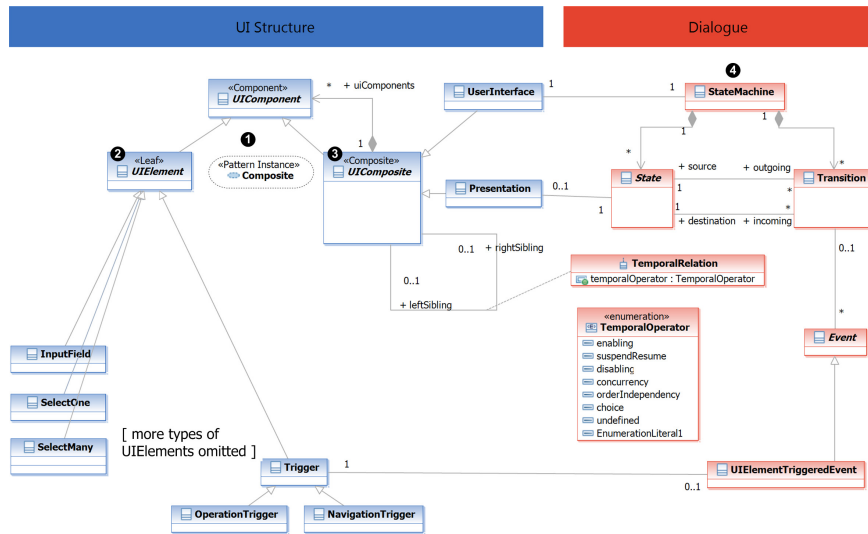


Fig. 5 Binding user interface elements to a data model

The second subtype of *UIComponents* are *UIComposites* (see ❸ in Fig. 5). *UIComposites* can contain other *UIComponents* via the association “uiComponents” and hence build up the “branches” of the hierarchy tree (see the ○ symbols in the

Timetable sample in Fig. 2). A *UIComposite* can be connected to its left and right sibling by temporal relations (see the horizontal lines  $\text{---}|=\text{---}$  in Fig. 2). In the metamodel this is described by an instance of the association class *TemporalRelation* which connects two *UIComposites* *leftSibling* and *rightSibling*. There are several kinds of temporal operators, such as *enabling*, *suspendResume* or *choice* (see the enumeration *TemporalOperator*).

There are two special cases of *UIComposites*: A *UserInterface* represents the whole user interface and is therefore the root of the hierarchy. In the Timetable sample this is the node “Timetable enquiry” (see Fig. 2).

Another special case of an *UIComposite* is a *Presentation*. A *Presentation* is a hierarchy node that was selected during the adaptation process, because all *UIElements* contained in the subtree below it should be presented coherently. For instance see the node “Time of travel” in the Timetable sample (Fig. 3): This node and the subtree below it are surrounded by a marked area to indicate that all *UIComponents* within that area will be presented in one coherent *Presentation*. Hence, this *UIComposite* will be converted into a *Presentation* in further transformation steps.

## 7.2 Dialogue Model

The dialogue model of an abstract user interface is described by a state machine (see ④ in Fig. 5) which is based on UML Statecharts [14]. It consists of *States*, which are linked to *Presentations* generated in the adaptation process. As long as the *UserInterface* is one particular state the related *Presentation* is displayed (or presented in different ways on non-visual interfaces).

When the *UserInterface* performs a *Transition* to a different *State* the next *Presentation* is displayed. *Transitions* can be started by *Events*, for instance by a *UIElementTriggeredEvent*, which fires as soon as the related *UIElement*, such as a *Trigger*, is triggered.

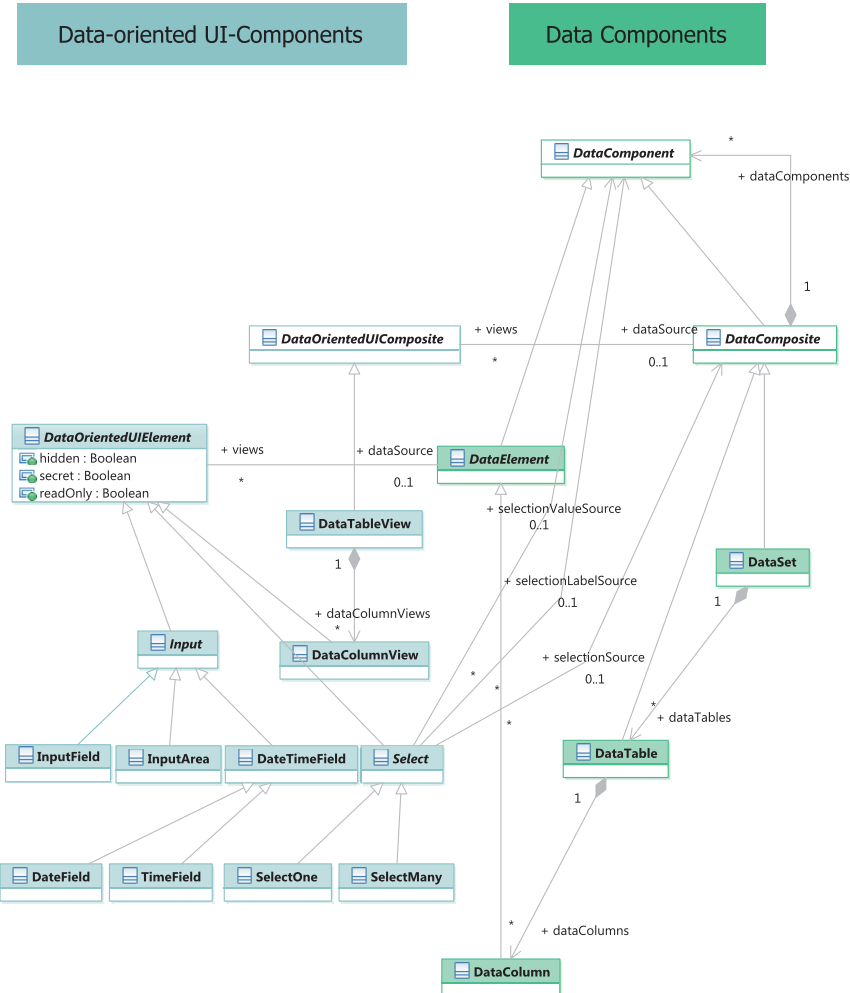
There are many other event types, which have been omitted here to simplify the metamodel illustration.

## 7.3 Binding UI Elements to Data Components

In the MANTRA metamodel user interface elements are grouped into different categories depending on their main function, for instance structure-oriented (e.g., *UIComposite*, *UIComponentGroup*, see Section 7.1), control-oriented (e.g., *Trigger*, *Hyperlink*, *MenuItem*, see Section 7.2), and data-oriented (e.g., various forms of *Input*, various forms of *Select*, and the composite *DataTableView*).

For the latter category, data-oriented user interface elements MANTRA allows to describe corresponding data components that will hold and organise the corresponding processed data, which is presented (and potentially edited) via the data-oriented UI elements.

Fig. 6 shows both data-oriented user interface elements (left) and data components (right) as well as some of the mappings between them. For instance, each

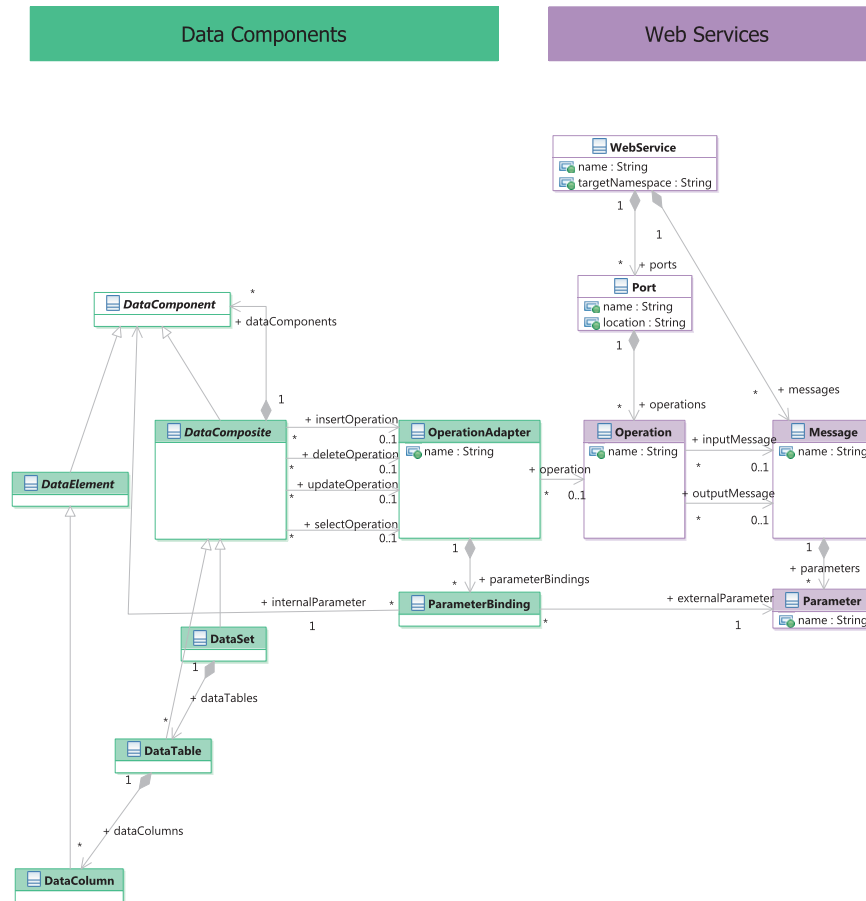


**Fig. 6** Binding data-oriented UI elements to data components

*DataOrientedUIElement* (including specialisations) has a *DataElement* as data source. Again the shown sections correspond to packages in the metamodel overview (Fig. 4).

#### 7.4 Binding Data Components to the Application Core

As mentioned earlier, the front-end is integrated with the application core via web service protocols. To allow this integration, the AUI model references concepts in a web service model, which is based on a WSDL description.



**Fig. 7** Binding user interface elements to a web service

As an example of how this integration works on a metamodel level, Fig. 7 shows how data-oriented components are referring to the corresponding web service operations, which are used to retrieve or store data from/to the web service.

## 8 Discussion and Outlook

We have shown how our MANTRA approach can be used to generate several consistent user interfaces for a multi tier application (see Fig. 8).

We discussed how the user interface can be adapted on the AUI level by tailoring dialogue and logical presentation structures which take into account requirements

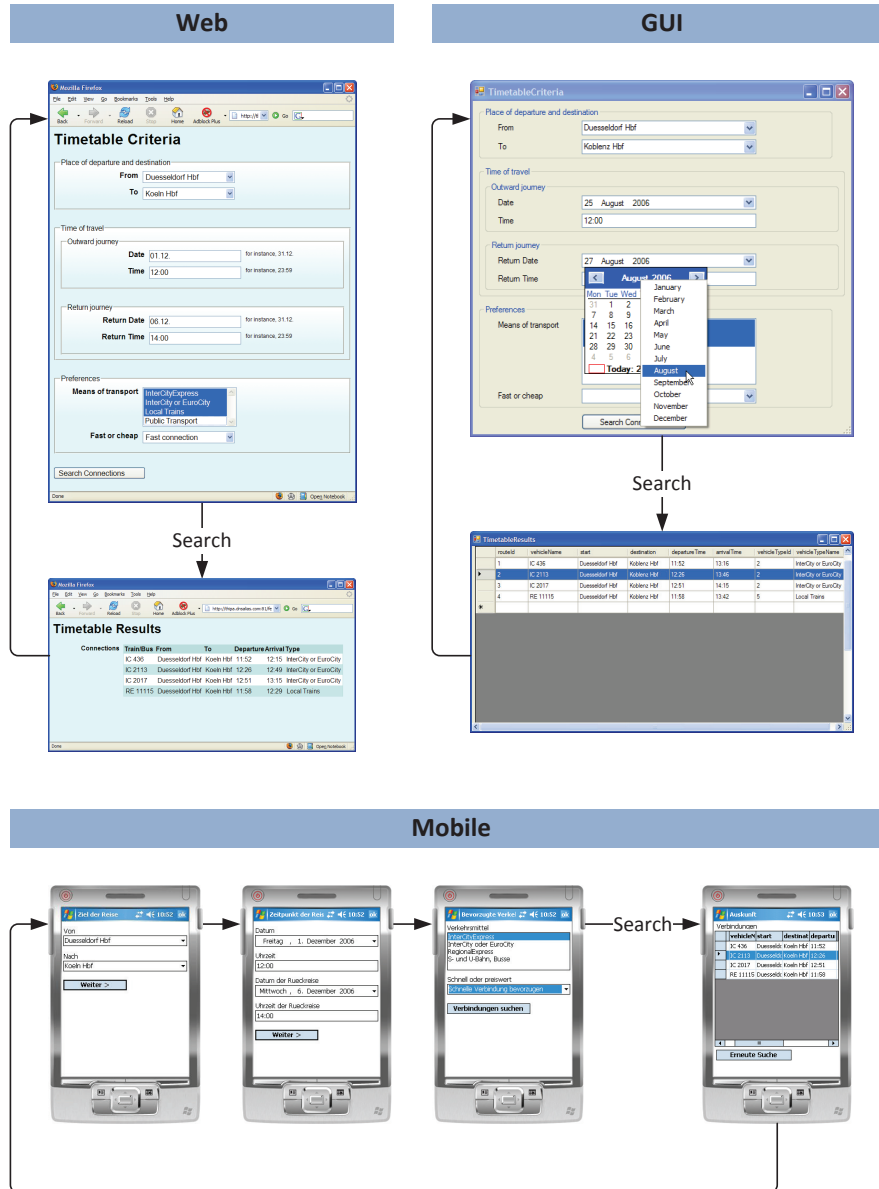


Fig. 8 The generated front-ends (Web, GUI, mobile)

imposed by front-end platforms or inexperienced users. For this we used the hierarchical structure of interaction elements and constraints on the dialogue flow which can be derived from a task model.

The approach generates fully working prototypes of user-interfaces on three target platforms (GUI, dynamic website, mobile device), which can serve as front-ends to arbitrary web services. Such generated prototypes are beneficial in rapid prototyping and early evaluation of user interfaces, since they facilitate functional evaluations by performing and analysing practical use cases.

The approach is geared towards interaction patterns that are commonly found in electronic forms and hypertext-like applications. It would, for instance, be very difficult to model and generate applications like an image processing tool or a spreadsheet calculation application.

### 8.1 *Applied Technologies*

When implementing MANTRA, we described the meta-models (including platform-specific concepts) in UML and then converted these to Ecore, since we use the Eclipse Modeling Framework (EMF) [15] to handle all processed models and the corresponding meta-models.

The various model transformations (e.g., for steps ② and ③ in Fig. 1) are described in ATL [16].

We use a combination of Java Emitter Templates and XSLT to generate ( ④ in Fig. 1) arbitrary text-oriented or XML-based implementation languages (e.g., C-Sharp or XHTML with embedded PHP).

The coordination of several steps in the model flow is automated by Apache Ant [17] (integrated into Eclipse) and custom-developed “Ant Tasks” to manage the chain of transformations and code generation.

We use web services as an interface between the UIs and the application core. Hence, the UI models reference a WSDL-based description of operations in the application core. The generated UIs then use web service operations, for instance, to retrieve results for a query specified by the user.

### 8.2 *Future Work*

In the discussion of the AUI adaptation it has been mentioned that the derived user interface structures (first the adapted AUI, then later the CUI) are not fully specified by the given input models (AUI with temporal constraints). In other words, when progressing down the MANTRA model workflow (see Fig. 2) and making decisions on how to *best* implement the given abstract specifications we can choose among different potential solutions. This provides the opportunity for various optimisation approaches. We intend to explore this further.

Another aspect for future work is the modelling and handling of variability and commonality among the multiple front-ends of one application. On an *element* level this can be addressed by abstraction, where one abstract element (e.g., “Select One” can describe multiple concrete solutions (e.g., “HTML Link List” and “MS Windows ListBox”). This however does not allow to describe variations among front-ends on a *structural* level. For instance, how do you represent that certain dialog

structures and navigation paths are only available in some front-ends, but not all of them? Here, approaches from product line engineering and variability modelling could be beneficial [18, 19].

Related to this, feature modelling [20] could be applied to describe variation and configuration options. Then, techniques from product configuration [21, 22] could be applied and extend to describe how configuration options are chosen – potentially over multiple stages [23]. Finally, techniques from feature mapping [24] and variability realisation [25] could be used to describe how the chosen options influence the artefacts that actually describe the user interface.

In current work, we have taken first steps towards such an integration of model-based user interface engineering and model-driven product line engineering approaches. In particular, we integrated feature models, which describe the variability and configuration choices of the product line, and abstract user interface models, which describe the common user interface structure of all products. More details are described in [11].

**Acknowledgements.** This work was partly supported, in part, by the Science Foundation Ireland (SFI) grant 03/CE2/I303\_1 to Lero – The Irish Software Engineering Research Centre, <http://www.lero.ie/>.

## References

1. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. *Interacting with Computers* 15(3), 289–308 (2003)
2. Puerta, A.R., Eisenstein, J.: Interactively mapping task models to interfaces in MOBI-D. In: DSV-IS 1998 (Design, Specification and Verification of Interactive Systems), Abingdon, UK, June 3–5, pp. 261–273 (1998)
3. Clerckx, T., Luyten, K., Coninx, K.: The mapping problem back and forth: customizing dynamic models while preserving consistency. In: TAMODIA 2004 (Third Annual Conference on Task Models and Diagrams), Prague, Czech Republic, November 15–16, pp. 33–42. ACM Press, New York (2004)
4. Vanderdonckt, J.: Advice-giving systems for selecting interaction objects. In: UIDIS 1999 (User Interfaces to Data Intensive Systems), Edinburgh, Scotland, September 5–6, pp. 152–157 (1999)
5. Paternò, F.: One model, many interfaces. In: CADUI 2002 (Fourth International Conference on Computer-Aided Design of User Interfaces), Valenciennes, France, May 15–17 (2002)
6. Forbrig, P., Dittmar, A., Reichart, D., Sinnig, D.: From models to interactive systems – tool support and XI ML. In: IUI/CADUI 2004 Workshop Making Model-Based User Interface Design Practical: Usable and Open Methods and Tools, Island of Madeira, Portugal (2004)
7. Florins, M., Simarro, F.M., Vanderdonckt, J., Michotte, B.: Splitting rules for graceful degradation of user interfaces. In: IUI 2006 (Intelligent User Interfaces 2006), Sydney, Australia, January 29 – February 1, pp. 264–266 (2006)

8. Eisenstein, J., Vanderdonckt, J., Puerta, A.R.: Applying model-based techniques to the development of UIs for mobile computers. In: *IUI 2001 (6th International Conference on Intelligent User Interfaces)*, Santa Fe, NM, USA, January 14-17, pp. 69–76 (2001)
9. Seffah, A., Javahery, H.: *Multiple user interfaces: cross-platform applications and context-aware interfaces*. John Wiley & Sons, New York (2004)
10. Botterweck, G.: A model-driven approach to the engineering of multiple user interfaces. In: Auletta, V. (ed.) *MoDELS 2006*. LNCS, vol. 4364, pp. 106–115. Springer, Heidelberg (2007), doi:10.1007/978-3-540-69489-2\_14
11. Pleuss, A., Botterweck, G., Dhungana, D.: Integrating automated product derivation and individual user interface design. In: *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2010)*, pp. 69–76 (January 2010)
12. Paternò, F., Mancini, C., Meniconi, S.: ConcurTaskTrees: A diagrammatic notation for specifying task models. In: Howard, S., Hammond, J., Lindgaard, G. (eds.) *Interact 1997 (Sixth IFIP International Conference on Human-Computer Interaction)*, Sydney, Australia, July 14-16, pp. 362–369. Chapman and Hall, Boca Raton (1997)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading (1995)
14. OMG: Uml 2.0 superstructure specification (formal/05-07-04). Object Management Group (2005)
15. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: *Eclipse modeling framework: a developer's guide*. In: *The Eclipse Series*. Addison-Wesley, Boston (2003)
16. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
17. Holzner, S., Tilly, J.: *Ant: the definitive guide*, 2nd edn. O'Reilly, Sebastopol (2005)
18. Clements, P., Northrop, L.M.: *Software Product Lines: Practices and Patterns*. In: *The SEI series in software engineering*. Addison-Wesley, Boston (2002)
19. Pohl, K., Boeckle, G., van der Linden, F.: *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, New York (2005)
20. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: *Feature oriented domain analysis (FODA) feasibility study*. SEI Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute (1990)
21. Botterweck, G., Thiel, S., Nestor, D., bin Abid, S., Cawley, C.: Visual tool support for configuring and understanding software product lines. In: *12th International Software Product Line Conference (SPLC 2008)*, Limerick, Ireland (September 2008); ISBN 978-7695-3303-2.
22. Botterweck, G., Janota, M., Schneeweiss, D.: A design of a configurable feature model configurator. In: *Proceedings of the 3rd International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS 2009)*, pp. 165–168 (January 2009)
23. Czarnecki, K., Antkiewicz, M., Kim, C.H.P.: Multi-level customization in application engineering. *Commun. ACM* 49(12), 60–65 (2006)
24. Heidenreich, F., Kopceck, J., Wende, C.: Featuremapper: Mapping features to models. In: *ICSE Companion 2008: Companion of the 13th international conference on Software engineering*, pp. 943–944. ACM, New York (2008)
25. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. *Software - Practice and Experience (SP&E)* 35, 705–754 (2005)



Model-Driven Development of Advanced User Interfaces

Hussmann, H.; Meixner, G.; Zuehlke, D. (Eds.)

2011, XX, 304 p., Hardcover

ISBN: 978-3-642-14561-2