UNIVERSITÉ DU LUXEMBOURG

Laboratory for Advanced Software Systems

Technical Report

A Bridging Mechanism for Adapting Abstract Models to Platforms

Sam Schmit-van Werweke

Faculty of Science, Technology and Communication
Computer Science and Communications Group
University of Luxembourg
6, rue Coudenhove-Kalergi
L-1359 Luxembourg

TR-LASSY-12-10

To Sandy Schomer, for her patience,
understanding and general support.

Abstract

In the model-centric view of model-driven software development the model is the code and the actual software can be fully generated from abstract models. Due to the complexity of currently used platforms, this vision has not been fully realized yet. In this master thesis we present an alternative approach in which only the application logic is modelled abstractly while platform-specific details are defined using platform-specific tools.

Two platform-independent Domain Specific Modelling Languages (DSMLs) have been developed which allow to create a bridge between abstract models and concrete platforms using an event-driven communication approach. The *Binding Core DSML* is used to describe a targeted language structurally as well as idiomatically. The *Mapping DSML* uses language specific extensions of the *Binding Core DSML* to create connections between event sources and event targets. Model editors have been implemented with which to create DSML instances and a code generator has been created which is able to produce source code from these instances. Finally, the DSMLs have been tested and validated using a case study. The study is based on an invoicing system and has been implemented using EP and Java Swing.

Acknowledgements

"Time flies like an arrow, fruit flies like a banana."

— Unknown[1]

This thesis is a result of my research in the Computer Science and Communications Research Unit of the University of Luxembourg. Over the last 6 months I have worked and learned more than ever before in my life, but I could not have achieved my work without the help of others and I am indebted to all of them.

First of all I would like to thank my thesis supervisor Prof. Pierre Kelsen for giving me the opportunity to work with him and a group of extraordinary people on a fascinating research topic. I would also like to thank him for his guidance and supervision of this project, always reminding me to think more... *abstractly.*

Next, I would like to thank my advisor Christian Glodt for all the time and effort he sacrificed for my thesis. We spent endless hours in discussions while developing the components of my thesis and he spent even more time implementing the Eclipse plugins and code generator.

I would also like to thank Moussa Amrani for willingly and eagerly sharing his expert knowledge with me.

Finally, a big thank you to all my friends, my family, and especially my girlfriend, for their patience, understanding and support during this time.

Working on this thesis has been a wonderfully frustrating experience... and I would not have missed it for the world.

Sam Schmit-van Werweke

Luxembourg, August 2012

[1]Generally attributed to Groucho Marx.

Contents

List of Figures

Listings

CHAPTER 1

Introduction

1.1 Motivation

Advances in IT in the last decades have created a plethora of different tools, languages, frameworks, libraries and platforms, with a vast set of features which are difficult to master. To overcome this complexity Model Driven Software Development (MDSD) has seen a rising interest as well. Models abstractly represent a target domain and thus allow to reason about that domain without having to consider implementation details. The ultimate goal of MDSD is to be able to model every single aspect of a software system, from the requirements to the end product(s). The models are the code and the documentation, they are validated and verified, and they are either executable or the application can be generated entirely from them.

This concept is a vision which has not been fully realised yet. Full support for a given platform is often only achieved by including platform specific concepts in the modelled system. This pollutes the models and renders multi-platform support very difficult, if not impossible. For example, no platform independent Graphical User Interface (GUI) modelling language exists yet and existing solutions have limited functionality or result in GUIs with an unnatural feel to them.

1.2 Objective

The goal of this master thesis is to explore an alternative approach in which only the application logic is modelled abstractly while platform-specific details are defined using platform-specific tools. An approach is to be developed which allows to connect the abstractly defined models with the platform specific code.

A particular scenario that will be considered is the creation of a GUI using a Visual Editor, while a bridging mechanism is to be developed to interface the platform-specific image of the abstract model with the GUI specification to obtain a fully functional application.

1.3 Contribution

The main contribution of our work consists of two Domain Specific Modelling Languages (DSMLs) with which to create a bridge between abstract models and concrete platform code. The bridging mechanism is realised using event-based communication. Additionally a code generator was created with which to generate the source code implementing the modelled bridge.

The *Binding Core DSML* holds the role of "communication translator". With it, the targeted language or platform can be described structurally - i.e. the language syntax - as well as idiomatically - i.e. how the language elements may be used. The *Mapping DSML* holds the role of "communication manager". It uses the idiomatic descriptions in a bridging model to allow navigating to event sources and targets as well as creating a connection between them.

The implemented code generator uses the structural definitions in the bridging models in order to generate the source code which implements the behaviour designed in a *Mapping DSML* instance.

1.4 Structure

The remainder of the thesis is structured as follows: *Chapter 2. Problem Description* contains a definition of the problem to be solved by our contribution. *Chapter 3. State of the Art* gives a summary of the state of the art in MDSD and the work related to our thesis. *Chapter 4. DSMLs* defines the main contribution of our work, including the DSMLs with their abstract syntax and dynamic behaviour. *Chapter 5. Code Generation* describes how the created code generator generates the desired source code. *Chapter 6. Case Study* shows how to use the two DSMLs by implementing a case study, which focuses on an invoicing system. *Chapter 7. Evaluation* evaluates our contribution and *Chapter 8. Conclusion* summarises the work done for this thesis and concludes by mentioning some of the possible additions, alterations and extensions to the work done for this thesis.

CHAPTER 2

Problem Description

In this chapter we give a precise definition of the problem addressed by the thesis. In section 2.1 we define the main constituents of the problem which is to be solved, as well as the overall goal of the thesis. In section 2.2 we sketch how we plan to achieve this goal using Domain Specific Modelling Languages.

2.1 The goal

The problem input is a set of abstract models on the one hand and platform specific code on the other (c.f. figure 2.1). For the sake of this thesis topic, the models represent the business logic of the application and the platform-code is a Graphical User Interface (GUI) for the abstractly modelled system.

The environment in which the abstract models have been created shall provide a code generator. This generator shall be able to generate source code targeting the same platform as the GUI code. Furthermore, the target platform shall be object-oriented in nature.

The goal of this thesis, i.e. the desired output, is to create a bridge between these two elements, i.e. the models and the code, which allows them to communicate with each other. The bridge should contain the information necessary to allow a code-generator to create the source code necessary to create a fully functional application.

The GUI and the business logic will be communicating with each other using a simple event driven communication

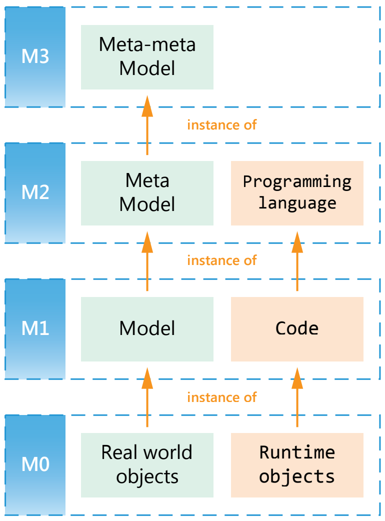


Figure 2.1: Four-layered Metamodel Hierarchy, based on the UML metamodel hierarchy [35]

model. It shall contain all the relevant information needed in order to realise the communication in a manner independent of a specific programming platform. In our case, the GUI needs to be able to interact with the business side, i.e. defined behaviour on the business model, as well as to update its own internal state as a reaction to state changes in the business models. This type of information exchange is well researched, for example see [67].

2.2 Domain Specific Modelling Languages

The communication between the abstract models and the code is realised using two Domain Specific Modelling Languages (DSMLs). To that extend, the languages used by the two elements need to be translated, in order for them to be used in a common communication model.

This task is achieved by the **Binding Core DSML**. Language specific extensions of this DSML will bind a specific language and thus its types and instances. The bound language is described first structurally using object-oriented notations and secondly idiomatically. The structural information is necessary for a code generator to be able to generate the desired source code.

The **Mapping DSML** allows to create connections between event sources and multiple event targets. It uses the idiomatic description of a language as provided by its Binding Core DSML extensions, in order to know which of the language elements can be interpreted as event source or target, as well as how to navigate from a "root instance" to the instance which provides the source or target.

We will introduce the state of the art concerning Model Driven Software Development and the work related to this thesis in the next chapter.

CHAPTER 3

State of the art

"Any intelligent fool can make things bigger, more complex and more violent. It takes a touch of genius and a lot of courage to move in the opposite direction."

— Albert Einstein

Models play an ever increasing role in software development. *"[They] provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on the relevant ones."* [10]

In section 3.1 we give a short introduction into current practices of Model Driven Development (MDD), including Model Driven Architecture (MDA) and Domain Specific Modelling Languages (DSMLs). In section 3.2 we define the concepts of Platform and Platform Model and in section 3.3 we discuss more directly work related to our thesis topic.

3.1 Model Driven Software Development

Modelling approaches have been around for a long time in many areas in science and art. For example, architectural drawings are models of constructions and the notes on a sheet of music are a model for the sound a composer wishes to produce. The main advantage of a model is that it allows to abstractly represent the relevant information of a system in a precise and concise manner, allowing to reason about highly complex systems without having to deal with distractions. Although modelling approaches have been used in software development since the 1950s, interest has been rising only in the last few years [73].

3.1.1 Modelling Spectrum

Brown et al. categorised the spectrum of modelling approaches used by software developers today as illustrated by figure 3.1 on the next page [9, 10]. According to them, the **code only** approach is still used by most programmers. The programmers rely

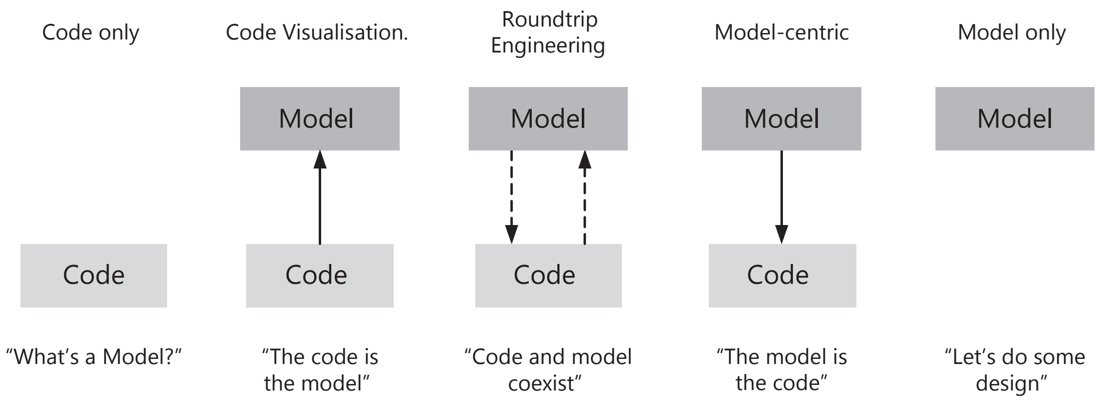


Figure 3.1: Modelling Spectrum [10]

solely on the source code created by them to represent the system they are developing. In this approach the complexities of the created system are intertwined with those of the chosen execution platform and programming language.

In **code visualisation**, models are created from the source code. These models may be used to verify and document the code, but the code is still the main artefact to create and maintain. In the **model only** approach, models are used for design purposes. This means that they are used to understand and define the domain of the modelled system, but they are not necessarily connected to its implementation.

With **roundtrip engineering** the models become more important. With this approach, an implementation of a system is based on design models. Its main disadvantage is that the source code and the set of models are often disconnected and need to be maintained separately. In the **model-centric** approach, the focus shifts entirely onto models. They are the artefacts to be created and maintained and any source code is entirely generated from them. A disadvantage of this approach is that platform details may also need to be included in the models in order for them to contain the information necessary to generate fully functioning source code.

3.1.2 General Purpose Modelling Languages

Modelling approaches generally fall into one of the two categories of General Purpose Modelling Language (GPML) and Domain Specific Modelling Language (DSML). The former allows to model a wide variety of different applications using the modelling languages it provides. An arguably well known example is Model Driven Architecture (MDA) [32] with its set of GPMLs known as Unified Modelling Language[1] [35] or its executable counterpart Foundational Subset for Executable UML (fUML) [34]. fUML allows to model the structure, the state as well as the static and dynamic behaviour of a system.[2] On the other hand, to support modelling a wide variety of applications, general purpose modelling systems such as fUML have a large set of features which can make them complex and difficult to use.

[1]For more details on MDA we would like to refer you to the books by David S. Frankel [23], Kleppe et al. [54] and Mellor et al. [59]. For more details on UML we would like to refer you to the books by Ahmed et al. [2], Booch et al. [8], Miles et al. [62], Martin Fowler [22], Dennis et al [15] and Hassan Gomaa [27] to name just a few.

[2]For more details on fUML we would like to refer you to the books by Mellor et al. [58], Dragan Milicev [63] and Raistrick et al. [70].

Furthermore, due to the complexity and diversity of current platforms, it is often very difficult, if not impossible, to capture in a single model the information necessary to target multiple platforms during code generation. One example is the current smartphone technology with its two major camps: Android and iOS. They use different hardware, different operating systems, different libraries and frameworks and different programming languages. Most importantly, they have a very different feature set - e.g. compare Quartz 2D [45] against Java 2D [48] - and current modelling approaches have difficulty providing a native experience. Modern cross platform development tools are usually code-centric and they are either targeted towards creating games using purpose build libraries [56] or they use programming languages different from the platform specific ones, such as LUA [46], Flash [42], JavaScript [44, 43] or Ruby [65]. Actually, the Android camp is fragmented to such a degree [49], that some developers have decided to drop support for the Android platform entirely [64, 72].

3.1.3 Domain Specific Modelling Languages

On the opposite side of GPMLs are Domain Specific Modelling Languages (DSMLs). The following definition is based on [16]: a DSML is a modelling language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain. In other words, and this is considered to be their main advantage, DSMLs use concepts of the targeted domains in their own make-up.

One possibility to create DSMLs is to use UML as the modelling environment (see for example [25] and [74]). Other possibilities include the Eclipse Modelling Framework (EMF) [77], General Modelling Environment (GMA) [76] and MetaEdit+ [61] to name just a few examples.

The initial investment when using DSMLs is considered to be generally higher than with conventional methodologies, but according to Hudak et al., their advantages can quickly compensate (c.f. figure 3.2).

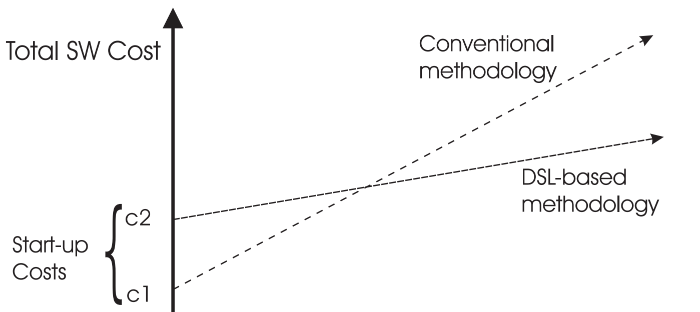


Figure 3.2: DSL Payoff [40]

DSMLs have been used successfully for a long time in many different areas, as for example in Embedded and Real Time System development [60, 17, 20, 57] and Business Process Modelling [1, 5, 66]. More examples of Domain Specific (Modelling) Languages can be found in [16].

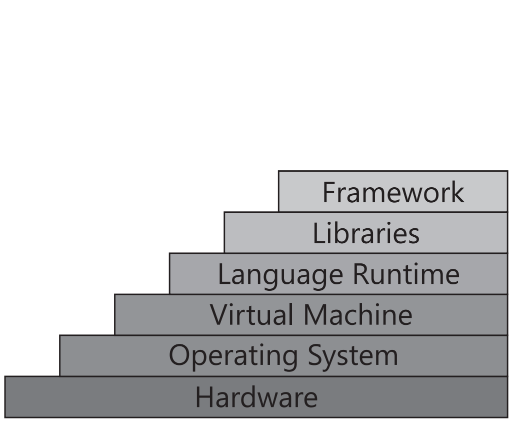
3.2 Platform (Model)

According to Atkinson et al., the most explicit definition of the concept of a platform in the MDA context is probably to be found in the MDA Guide [4], which states [32]:

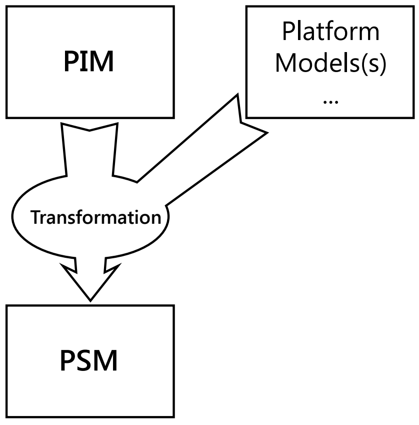
> *"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented."*

Atkinson et al. further state that the most concrete definition of "platform model" available today comes from the school of thought that characterises MDA in terms of transformations between DSLs [4]:

> *"According to this school of thought, the essential difference between the input and output models in the MDA transformation illustrated in [figure 3.3b] is that they are written in different languages [...]. Therefore, although it is not stated explicitly, language definitions essentially play the role of platform models in the DSL view of MDA."*



(a) Typical Platform Layers



(b) MDA Principle

Figure 3.3: Stereotypical platform views as per Atkinson et al. [4]

The former defines *platform* as the execution environment of a software system and the latter defines a *platform model* to be equal to a language definition. Atkinson et al. challenge these definitions, as they do not capture the entire nature of platform (models). The former definition tends to ignore that a platform is not always an execution environment, while the latter ignores key components of what should constitute a platform model and is thus incomplete.

Hence, Atkinson et al. propose a generalised notion of platform which they named *General Platform Model (GPM)*. Based on the GPM, Atkinson et al. finally propose a full platform description (c.f. figure 3.4 on the facing page).

A GPM groups *"four basic facets through which information about the capabilities and rules of an object-oriented platform is conveyed"* [4]. The following is a short summary:

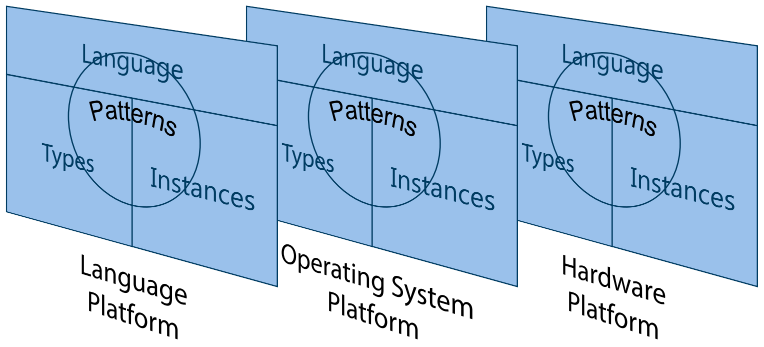


Figure 3.4: Full Platform Description [4]. Each square represents a GPM describing the designated platform.

Language describes how applications designed to use the platform can be constructed.

Predefined Types encompass the standard library provided with the platform as well as any framework libraries used during development.

Predefined Instances are objects pre-initiated by the language support layer, e.g. Java's standard I/O streams "in, out, err".

Patterns consists of the additional concepts and rules that are needed to use the capabilities found in the previous facets in a meaningful fashion.

We will use this notion of *platform* and *platform model* for the remainder of the thesis.

3.3 Related Work

Combining heterogeneous systems is an active field of research. Many different approaches, each with their own advantages and disadvantages, have been developed. These can be categorised into the following five categories:

3.3.1 Model Driven Architecture

One of the core principles of MDA is interoperability between different models [32]. Usually, this is achieved by providing a mapping between a Platform Independent Model (PIM) to a Platform Specific Model (PSM). By applying model transformation to combine both, a platform specific end-model is created. This allows to create cross-platform implementations based on a single PIM.

The main disadvantage of MDA is its complexity. The described methodology is difficult to master and thus prone to faults and errors.

3.3.2 Interoperability Frameworks

Interoperability frameworks have been widely used in an enterprise setting. Known as Enterprise Application Integration (EAI), these systems *"integrate heterogeneous applications and system via message-based communication"* [7]. Böhm et al. propose

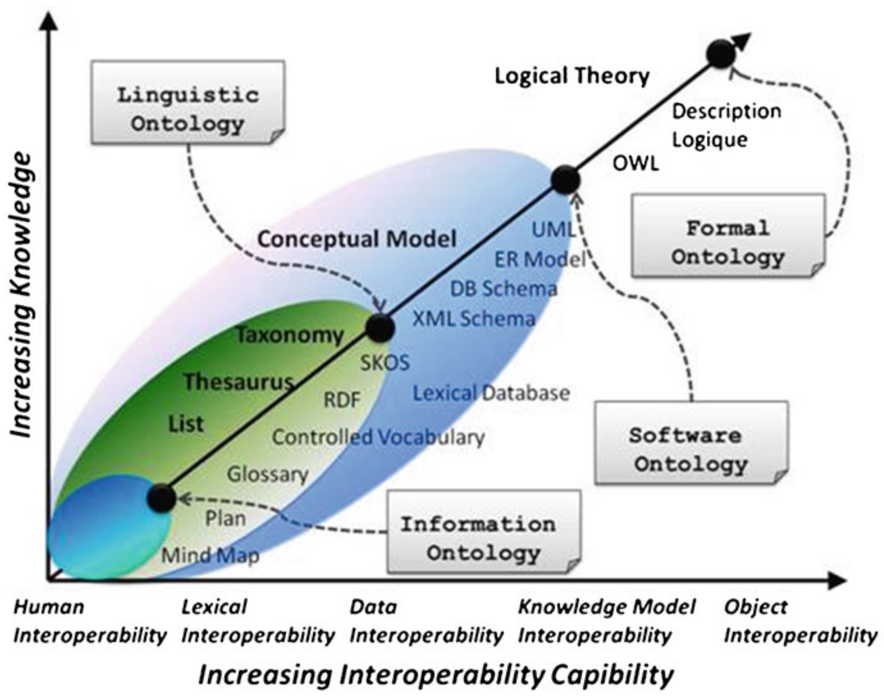


Figure 3.5: A schematic representation of the different kinds of interoperability [71]

a model-driven solution for generating dynamic adapters for Integration platforms in [7]. This methodology is supposed to overcome some of the limitations of current EAI systems, e.g. slow performance, functional restrictions and data independence. In an enterprise setting, Web Services play a very important role, which is why interoperability in this field constantly gains in importance. Therefore, Benatallah et al. propose a system with which adapters for web service integration can be created [6].

EAI systems are usually very complex and require dedicated servers to run. Our purpose is to combine abstract models with concrete code in order to generate a single end-application, which makes EAI unsuitable.

3.3.3 Ontology based Interoperability

Roussey et al. provide an overview over *"how ontologies can be used to improve interoperability between heterogeneous information systems"* [71]. Figure 3.5 shows an overview of their survey.

An example they cite is [30] by Grangel et al., in which they propose to use ontology based communication through model driven tools in order to create an interoperability framework. Ciocoiu et al. define in [13] an approach similar to ours, in that every component uses their own Interlingua ontology enabling them to communicate with each other using a shared ontology.

Although similar, ontologies are meant to be used to facilitate information exchange, which requires direct support from the concerned systems. Event-based communication can still be realised, e.g. using a message passing system, but we do not want to impose the use of such a system, since this could pollute the design of the abstract models and the concrete code with elements unrelated to their original purpose.

3.3.4 Component based Design

Component based modelling and software development is a well researched problem. In the context of interoperability for example, Gossler et al. define a methodology for composition in the context of component-based modelling [29]. They provide a purely abstract modelling framework, with which to achieve communication between heterogeneous components. Cao et al. propose in [11] a more practical approach, where the wrapper or glue code between the different models is generated using the domain specific knowledge provided by the different components.

Component based design simplifies the reusability of a software or modelling artefact, an advantage shared with our approach, but we do not want to impose such a design on neither the abstract models nor the concrete code. Our goal is to support any meta-model or programming language instance through the “API” they provide.

3.3.5 Model Transformation

By using model transformations, the goal is to combine the models which are to interact with each other, such that they may use their respective concepts directly. In other words, the different models are combined into a common model, or a means to reference each other is supplied, which utilises the concepts of all of the original models, ideally duplicate free. According to [84] this is usually achieved using one of the following approaches:

Metamodel Extensions
In this approach, a base metamodel is extended using the concepts of another one, either by inclusion or by referencing existing concepts. This task can be achieved by using e.g. Aspect Oriented Modelling techniques. The main advantages are that both metamodels remain reusable and the combined syntax and semantics can easily be build upon the original ones.

On the other hand, the semantics need to be “compatible”, or in other words, attention needs to be paid that the semantics of one metamodel are not broken by the inclusion of another metamodel. Furthermore, this approach is only applicable if both metamodels have been created using a common language, or meta-metamodel. This is often not the case, for example, if a bridge is to be created between abstract models and concrete code, since source code is an instance of a programming language for which usually no well-defined metamodel exists.

Metamodel Merge
Metamodel Merge is an operation in which the merged models actually fuse to form a new metamodel. This operation is provided on the level of packages by UML [36] and on the level of classes using MetaGME [19]. A general approach is proposed by Pottinger and Bernstein in [69] in which the correspondence between metamodel elements is defined by the user.

Again, the semantics of the merged metamodels need to be compatible in order for Metamodel Merge to be applicable, which makes this approach unsuitable for our purposes. Furthermore, this approach does not allow for creating a bridging mechanism with support for multiple platforms. Either a new metamodel has to be created for every combination of metamodel and platform, which considerably reduces reusability, or a single metamodel with support for multiple platforms has to be generated, which increases the model's complexity.

Embedding and extending
Alternatively, a metamodel may be constructed by *"[inheriting] the infrastructure of some other language, tailoring it in special ways to the domain of interest. This is called language embedding."* [39, 38]. This method allows the embedded language to reuse the tools, libraries and syntax of the host language. Another approach is to use a host language with support for extensions, thus facilitating language embedding, e.g. UML Profiles [36].

The main disadvantage of these methodologies is that the target language needs to be re-defined using the host language. Additionally, the host language needs to be expressive enough to support modelling the target language's concepts and semantics. On the other hand, these disadvantages are mitigated if the targeted language is to be supported only partly, as it is the case for our work.

In the next chapter, we will define the two Domain Specific Modelling Languages, introduced earlier in this thesis.

CHAPTER 4

DSMLs

"Most of the things in life are simple and only the wise understand them."

— Paulo Coelho

This chapter contains the definitions of the two DSMLs introduced in chapter 2. The DSMLs were developed using the Eclipse Modelling Framework (EMF) [77] and written in the Ecore modelling language [77, 41]. The Object Constraint Language (OCL) is used as an expression language in the *Mapping DSML* [33]. Its task is to query the different DSML instances and manipulate values.

First the core concepts modelled in the languages are explained in section 4.1. In section 4.2 we introduce the Ecore Modelling Language. Sections 4.3 up to 4.5 contain the definitions of the abstract syntaxes of the two languages. The dynamic semantics are described in section 4.6.

4.1 Core concepts

An OO based model or program has, at the very basic level, a state - the application's properties - and associated behaviour - operations and events [3]. While an operation can have an impact on the application's state, an event is often considered to be a notification "that something has happened", e.g. an operation was executed or the value of a property was changed. The properties, operations and events are considered to be provided by a "class". In other words, the value of a language element instance has a type, namely a "class", which defines the properties, operations and events accessible through the instance. This relation between an instance, its type and the contained properties, operations and events describe the structure of the language.

A second approach to defining a language is to use an idiomatic description of its elements. Such a description defines how different language elements may be used. For example, the event element mentioned previously can be regarded as an event source. An operation can be used as the target of an event, similarly to event handlers in Java, and lastly, properties are value providers. A specific language may allow to use its

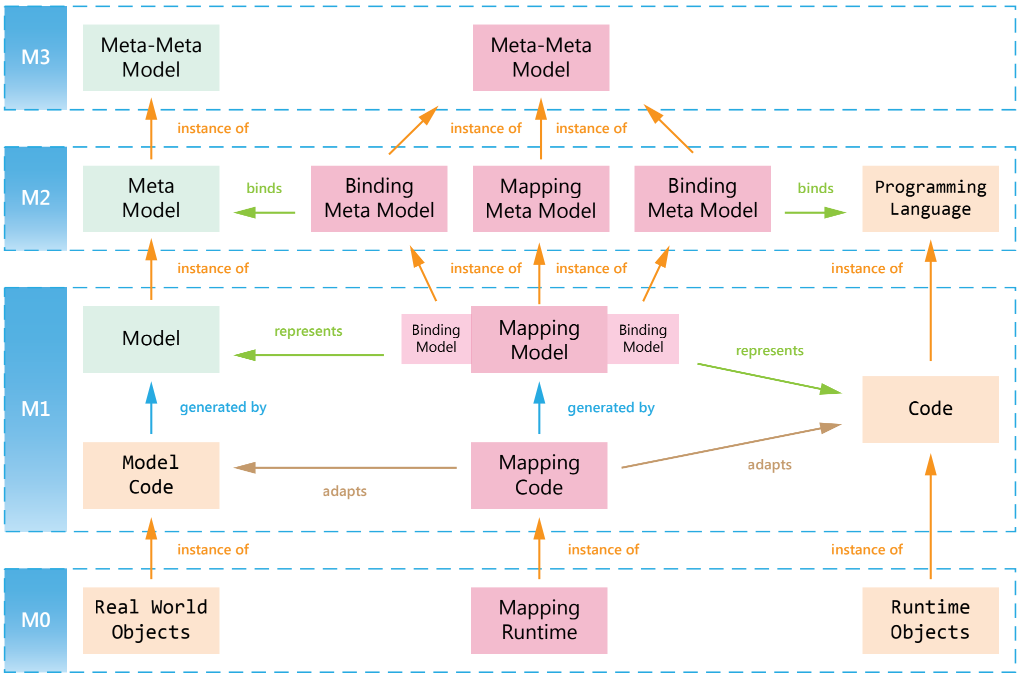


Figure 4.1: Metamodel Hierarchy including the DSMLs

elements in other ways as well, for example, if a Java class method returns a value, it too can be used as a value provider.

The structural and idiomatic language descriptions are represented by the *Binding Core DSML*. Since the concepts are abstract in nature, platform specific extensions are needed for every language which is to be supported. Level **M2** in figure 4.1 shows how two of these extensions bind to their respective languages. Level **M1** shows how an instance of each is created to represent language instances.

We have already discussed in chapter 2 on page 3 that the communication between the two targeted platforms is to be event-driven. An event has a source as well as one or more targets which are executed in response to the source triggering. In other words, a connection between an event source and multiple event targets of the two platforms is needed in order for there to be communication between the targeted platforms. Additionally, the source and the targets need to be located in the language instances. This entails a means to navigate from one language element instance to another.

The concepts of connections and navigation are represented by the *Mapping DSML*. It is situated on level **M2** in figure 4.1 alongside of the *Binding Core DSML* language specific extensions. Level **M1** shows how an instance of it uses the Binding DSML instances in order to create a bridge between the target platforms.

4.2 Ecore Modelling Language

The Ecore modelling language was developed to combine Java, UML and XML in a single modelling environment [77]. Since its introduction, and probably due to its simplicity and highly integrated tool support, it has been used as the modelling language for a multitude of different DSMLs [31]. Figure 4.2 on the next page shows part of Ecore's metamodel.

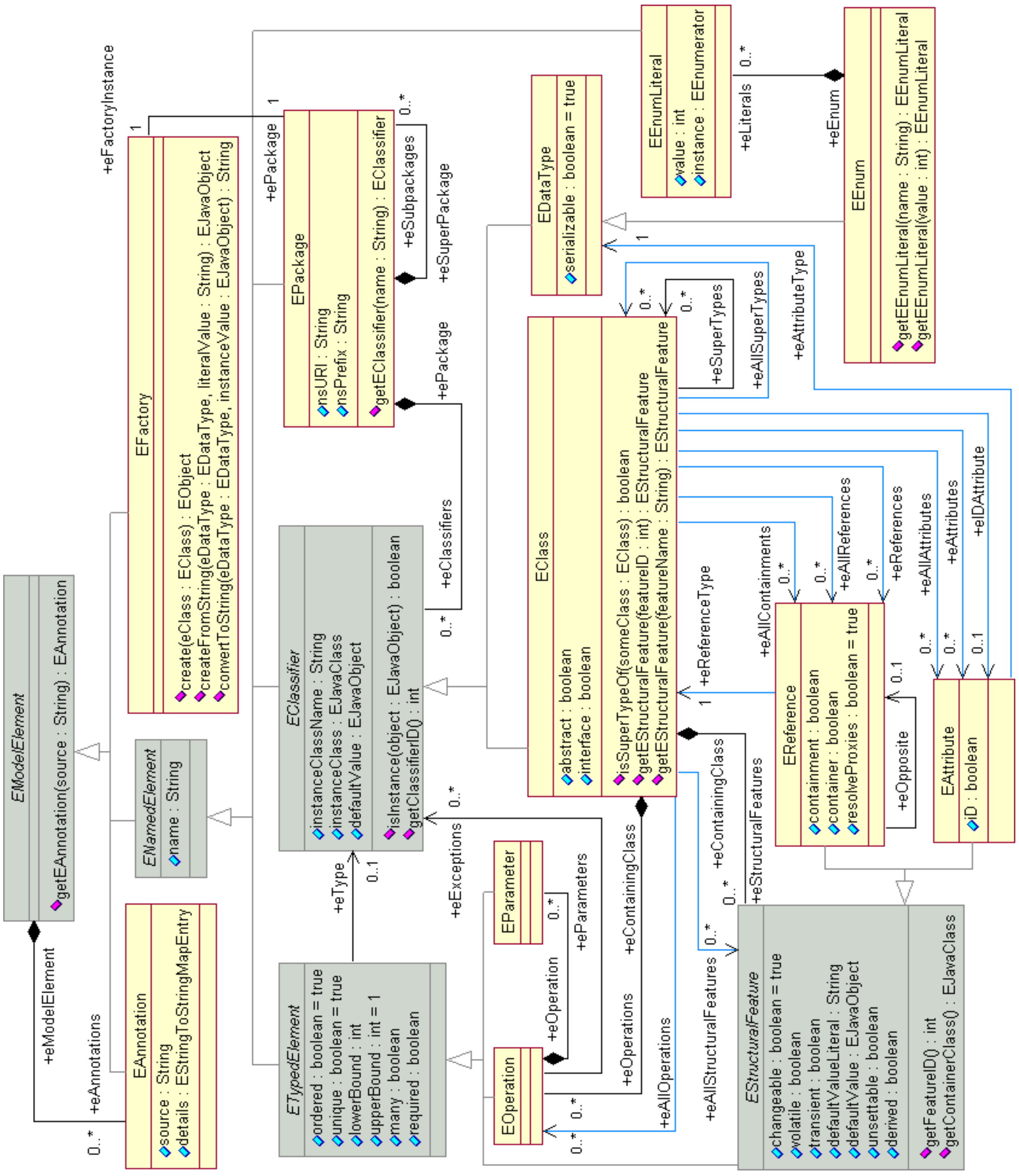


Figure 4.2: Relations, attributes and operations of the Ecore components [41]

The main element of the Ecore modelling language is *EClass*, which represents the concept of a class - or an interface, if the *interface* attribute is set to `true` - in Ecore. By convention, we will start the names of interfaces with an uppercase i.

An *EClass* may inherit from other EClasses - c.f. the *eSuperTypes* reference in the metamodel and it may reference other EClasses - c.f. the *eReferences* reference in the metamodel. Lastly, we use an EClass's ability to contain attributes - c.f. the *eAttributes* reference in the metamodel.

4.3 Binding Core DSML

The *Binding Core DSML* has been developed to express the different OO based concepts mentioned above. As its name already suggests, it only represents the **core** features of the binding language. In order to support the targeted modelling or programming languages, platform specific extensions of the core need to be created. Figure 4.3 shows the language metamodel and chapter 4.4 on page 18 introduces two platform specific extensions.

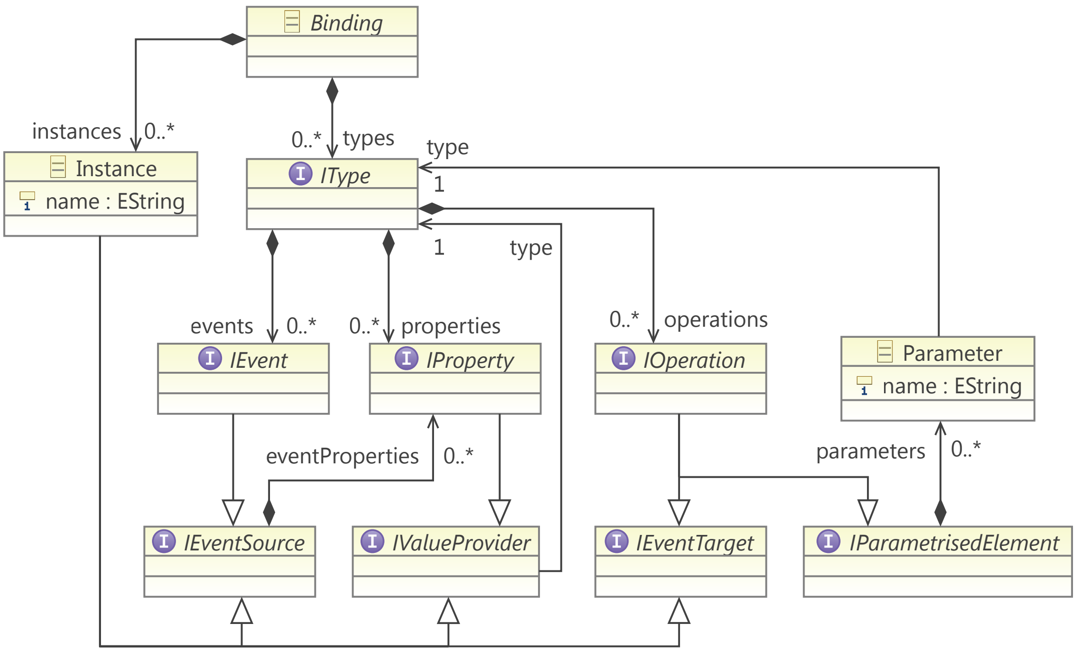


Figure 4.3: Binding Core DSML

Following is a natural language description of the abstract syntax of the *Binding Core DSML*. Where appropriate, the description includes examples based on the Java programming language.

Binding is the root *EClass* of this metamodel. It is abstract in order to prevent the creation of a model instance. The platform specific extension mentioned previously, needs to provide a concrete *EClass* inheriting from *Binding*.

The following six elements are used to describe the languages to be bridged conceptually and structurally. By this we mean that these elements provide the *Mapping DSML* with the knowledge of **where** to find **which** type of language element.

IType is the basic building block used to describe the targeted language. It defines the rules of the *Type System* used by the targeted language. For example: a Java Class would be represented as inheriting from *IType* in a Java specific extension to the *Binding Core DSML*. As such, it defines which *IEvent*, *IProperty* and *IOperation* elements are provided by this class.

IProperty is an artefact with which to define the elements the system created using the targeted language, which make up its state. For example, in the Java programming language class variables would be modelled as *IProperties*.

IEvent is a notification "that something has happened", i.e. a notification about an event. It represents the elements of the language capable of or responsible for this task. Which element of Java can be represented as *IEvent* depends on which library is used. For example, in order to be notified about an event in a GUI written using the Java Swing library, an event listener is required. Furthermore, an event listener may be responsible for different types of events. Hence, *IEvent* would represent the combination of a listener as well as the specific event type out of the collection of event types the listener can be notified about.

IOperation is an executable part of the behaviour of the target system model. It represents those elements of the language which may have an impact on the modelled system's state. For Java this would be the class methods.

IParametrisedElement describes an element inheriting from it as "possibly having parameters", such as *IOperation*. The list of parameter references by *IParametrisedElement* is ordered. For example, a Java class method may, but is not required to have parameters; they form part of the method signature. While the method parameters have a name, this information is only relevant during development of the method body. When the method is called by another part of the application, the type and the order of the parameters is important.

Parameter represents a parameter in the target language. It has a name and a type, both of which are required.

The following three interfaces and the *Instance EClass* describe a targeted language semantically. In other words, these tell the *Mapping DSML* **how** the elements of the language may be used.

IEventSource represents an element of the language which constitutes the source of an event. It may provide information on the event in form of an unordered list of *eventProperties*. An example of the latter is the `Event Object` which is delivered with every Java Swing event.

IValueProvider represents an element of the language which may be used by the *Mapping DSML* to *a)* determine the value to provide to a parameter as part of a *ParameterMapping*'s expressions or *b)* provide the value of a *PathElement* (c.f. section 4.5 on page 22). A value has a type associated with it as represented by the *IValueProvider*'s *type* reference.

IEventTarget represents an element of the language which may act as the target of an event. An *IOperation* such as a Java method is inherently an *IEventTarget*.

Instance represents the idea of an object instance during runtime. It allows to create new object instances, by using it as an *IEventTarget*, or to monitor its instantiation as an *IEventSource* and react accordingly. Furthermore, an *Instance* is the root for navigating to other instances in a *Mapping DSML* path (c.f. section 4.5 on page 22).

No further constraints are applied to the *Binding Core DSML* or its elements.

4.4 Platform specific Binding Core extensions

The languages developed during this thesis will later be validated by implementing a case study. To that extend, platform specific *Binding Core DSML* extensions are needed for the languages used for the study. These languages are EP and Java (Swing) and we will present the corresponding DSMLs in this section. For more details on the two languages and how they are used, please see chapter 6 on page 33.

4.4.1 EP Binding

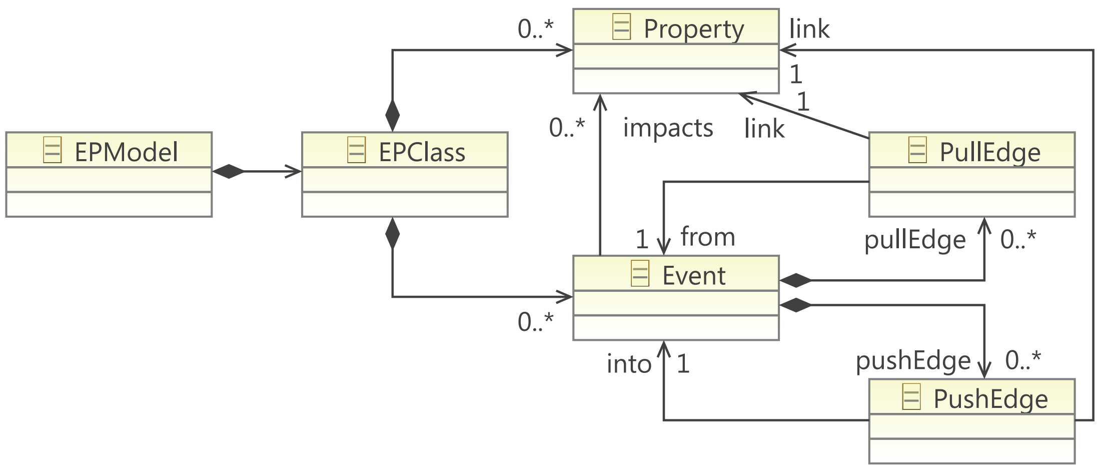


Figure 4.4: Simplified EP Metamodel [50]

EP is a language to create *"declarative executable models for object-based systems based on functional decomposition"* [50, 51] (see figure 4.4 for a simplified metamodel of EP). Its name is derived from its two main entities *Events* and *Properties*.

The static structure of an EP model is made up of *EPClasses*, created using the EP language, which contain properties [50] and which are organised in domains and bridges [52]. The former describe the state of the EP system while the latter provides a means for creating domain hierarchies.

The dynamic behaviour of an EP model is created using events and query properties with locally annotated OCL code snippets, both of which are contained within an EP model [50]. An event may directly affect the system state by changing the value of a property, in which case an impact link between the event and the target property is set. Such a link has an OCL code snippet assigned to it and when the link source event is triggered, the code snippet is evaluated and the returned value is assigned to the link target property. Events may also be forwarded to or respectively monitor other events in a model instance, i.e. they can act as the source for another event.

Last but not least, query properties are also part of a models behaviour. They are transient properties, the value of which are determined dynamically using OCL code snippets.

The concepts relevant for our purposes are EP classes, (query) properties and events. They are represented in the platform specific *Binding Core* extension named *EP Binding*. Figure 4.5 on the facing page shows the corresponding metamodel. Following is a natural language description of the abstract syntax of this model.

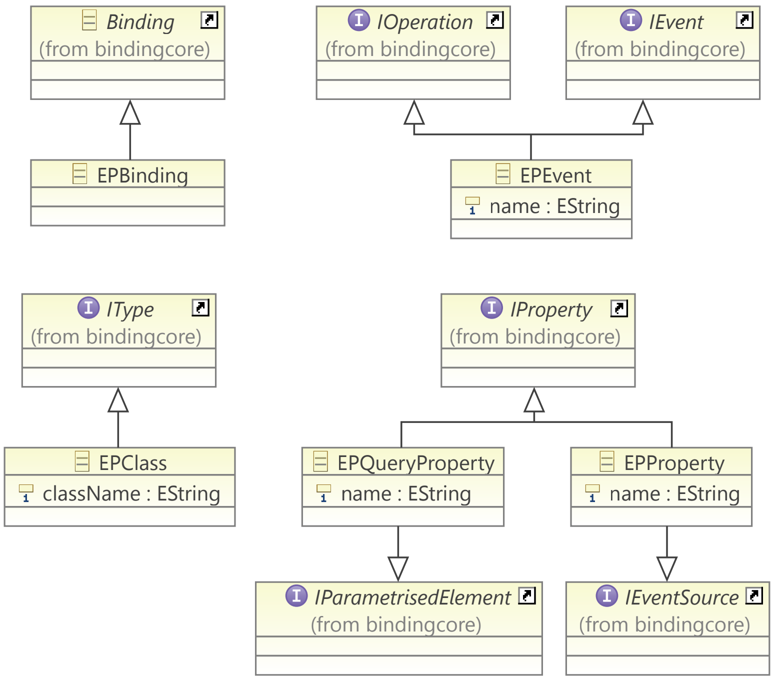


Figure 4.5: EP Binding Metamodel

EPBinding is the required concrete binding model root *EClass*.

EPClass represents the basis of the "EP type system". As seen in the EP metamodel on the preceding page, this element contains the components which make up an EP model.

EPEvent has the task of both *IOperation*, since it may affect the system state, and *IEvent*, since it may trigger another EP event.

EPProperty is by definition an *IProperty*. The set of EP properties in an EP system defines its state through the values associated to them. An *EPProperty* may also act as an *IEventSource*, even though this behaviour cannot be modelled in an EP class. It is however accessible through the EP runtime support.

EPQueryProperty is modelled as an *IProperty*. It technically is not part of the state of an EP system, since the value it returns is transient. Nonetheless, this value still may depend on the system state.

For the purpose of this thesis, the UML Object Diagram modelling language [35] was chosen as the language to represent instances of the DSMLs. This language is used purely for expositional purposes. The concrete DSML instances in the case study in chapter 6 on page 33 were created using their respective abstract languages. Furthermore, two alterations have been made to the UML Object Diagram notation. First of all, the order of the elements in an ordered list is shown using an increasing number in parentheses, starting at 1. Additionally, an *IType* element, if it is the target of a *type* relation, may also be shown as a property of the element the type relation belongs to.

Lastly, the different DSML object instances are colour coded. **Blue** means it is an instance of an *Java Binding DSML* element instance, **green** means *EP Binding DSML* element instance and **orange** means *Mapping DSML* element instance.

An exemplary model written in the concrete language for the *EP Binding DSML* can be seen in figure 4.6. The figure is an excerpt of the EP binding model created for the case study in chapter 6 on page 33.

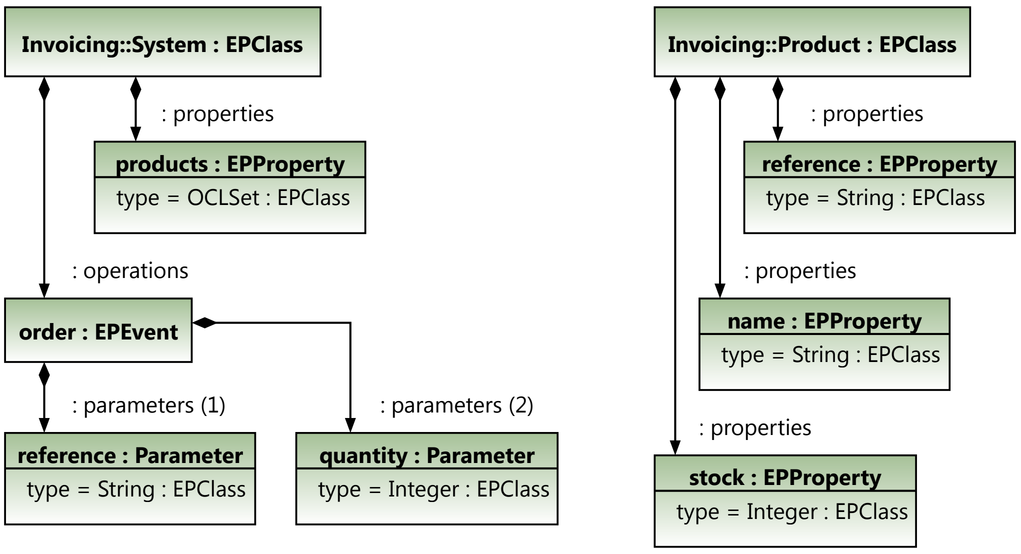


Figure 4.6: EP Binding instance example

The EP class named *System* in the *Invoicing* domain, has, amongst others, a property named *products*. This property is a set, as shown by its *type* property *OCLSet*. It also contains the EP event named *order* which has two parameters *reference* and *quantity*. Their order is shown as a number in parentheses in the *parameter* relations and their types are of *String* and *Integer* respectively.

Of the EP class named *Product* in the *Invoicing* domain, the three EP properties *reference*, *name* and *stock* are modelled. Their respective types are types provided by the EP language, but in order to be able to use them, they too have been modelled as *EPClasses* (shown here as class properties).

4.4.2 Java Bean Binding

The Java programming language was developed by James Gosling at Sun Microsystems and released in 1995 as one part of the Java Platform - the other being the Java Virtual Machine (JVM) [28, 83]. Its source code is compiled into byte-code, a language which is understood, interpreted and executed by the JVM. The JVM is also the reason for Java to be platform independent, that is, it is independent of the used operating system and hardware environment as long as a JVM for the target platform exists.

Unlike for example C/C++, Java is a strongly typed OO language and features a Garbage Collector (GC) which takes care of memory management. This reduces the risk for programming errors which according to Gosling et al. plagued application written in C/C++. For more details on the language and its constructs we would like to refer you to the specification [28] and the books by Christian Ullenboom [82, 81].

The Java language concepts are represented in the platform specific *Binding Core* extensions *Java Bean Binding*. Figure 4.7 on the next page shows the corresponding metamodel. Following is a natural language description of the abstract syntax of this model.

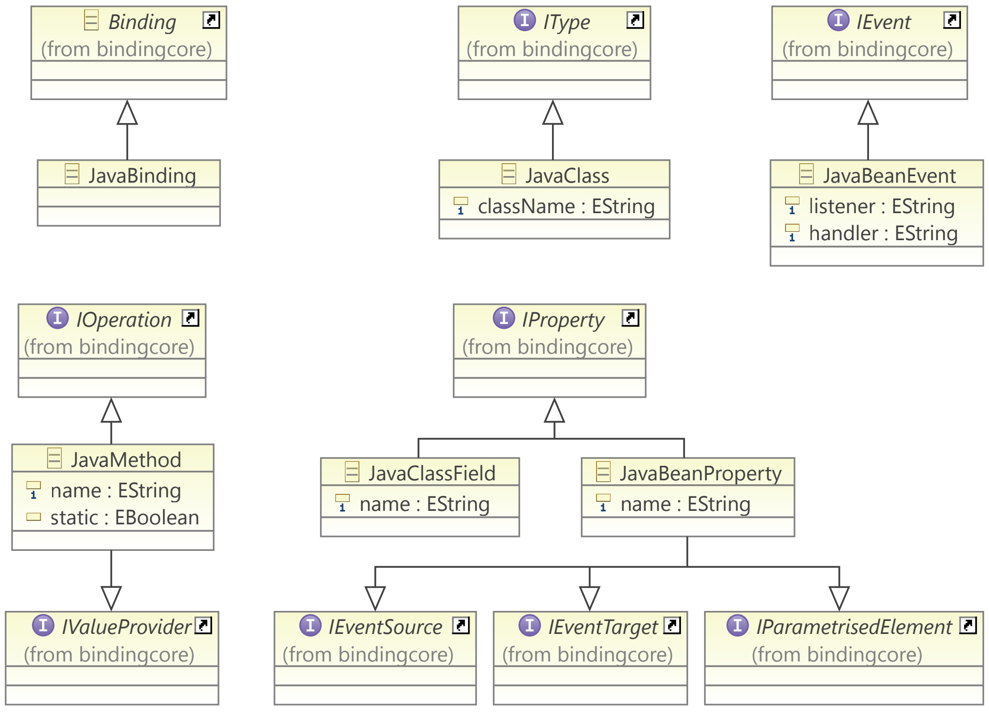


Figure 4.7: Java Bean Binding Metamodel

JavaBinding is the required concrete binding model root *EClass*.

JavaClass represents the basis of the "Java type system" and thus inherits from *IType*. A *JavaClass* defines the identity and the behaviour of an element of the represented type. It is itself identified by the *className* property, which represents the fully qualified class name of the *JavaClass*.

JavaBeanEvent constitutes an event notification in Java (Swing). Although in Java, the event source is the element which creates the notification, it is the notification itself which defines how to listen to which type of event. The former is modelled by the containment relation between *JavaClass* and *JavaBeanEvent*, as inherited from *IType*, and the latter is the reason for having *JavaBeanEvent* inherit from *IEvent*. The type of the listener required as well as the type of event triggered is defined by the two properties *listener* and *handler*.

JavaMethod represents a specific method identified by its *name* in a given *JavaClass*. Only public methods are to be modelled in the DSML and the means of how to execute it depend on whether the method is static or not. Furthermore, a Java method may return a value, which is why it may also be used as an *IValueProvider*.

JavaBeanProperty represents a Java class property which follows the Java Bean Specification. This means that a getter and a setter method is defined for this property and that the setter in turn creates an event when it is called, i.e. when the value of the property is changed. This is why it inherits from *IEventSource* as well as *IEventTarget* in addition to being an *IProperty*. Furthermore, a setter method of a *JavaBeanProperty* has a single parameter which holds the value the *JavaBeanProperty* is to be set to, hence it also inherits from *IParametrisedElement*.

JavaClassField represents a Java class field or class variable.

An exemplary model written in the concrete language for the *Java Binding DSML* can be seen in figure 4.8. The figure is an excerpt of the Java binding model created for the case study in chapter 6 on page 33.

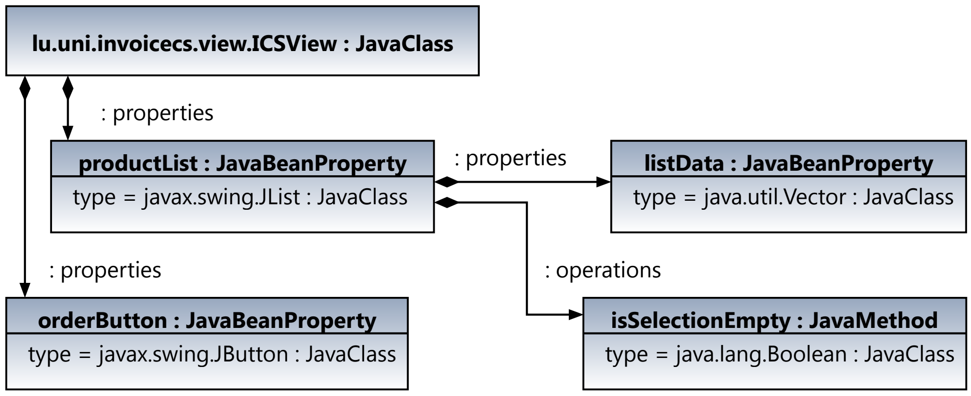

lu.uni.invoicecs.view.ICSView : JavaClass
: properties
productList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: properties
listData : JavaBeanProperty
type = java.util.Vector : JavaClass
: properties
: operations
orderButton : JavaBeanProperty
type = javax.swing.JButton : JavaClass
isSelectionEmpty : JavaMethod
type = java.lang.Boolean : JavaClass

Figure 4.8: Java Binding instance example

The Java class named *ICSView* in the package *lu.uni.invoicecs.view*, has, amongst others, the two shown properties named *productList* amd *orderButton*. Their *types* are *JList* and *JButton* in the package *javax.swing* respectively.

In order for the features of the *JList* type to be usable by the *Mapping DSML*, they are also included in the Java binding model. They are modelled as inheriting from *JavaClass*, here shown as a tree in the concrete language. Two exemplary elements are the *listData* Java Bean property of type *java.util.Vector* and the *isSelectionEmpty* Java method, which returns a value of type *java.lang.Boolean*.

4.5 Mapping DSML

The *Mapping DSML* has been developed to express the connection and navigation concepts mentioned in section 4.1 on page 13. Figure 4.9 on the next page shows the language metamodel; it is followed by a natural language description of the abstract syntax of this model.

Connection represents the multipoint connection described previously. It has exactly one source and an ordered list of targets (at least one). The *IEventSource* associated to the *EventSourcePath* referenced by the *source* relation can provide a list of *eventProperties* as defined by the *Binding Core DSML*. Since these properties may differ from event to event, a connection only has a single source.

Path is a means to identify the location of an instance. In other words, a path defines the navigation from one instance to another. It has a required *root* and an ordered list of path *elements*. If the root is the element to be either monitored or the intended target of the connection, then the list of path elements has to be empty.

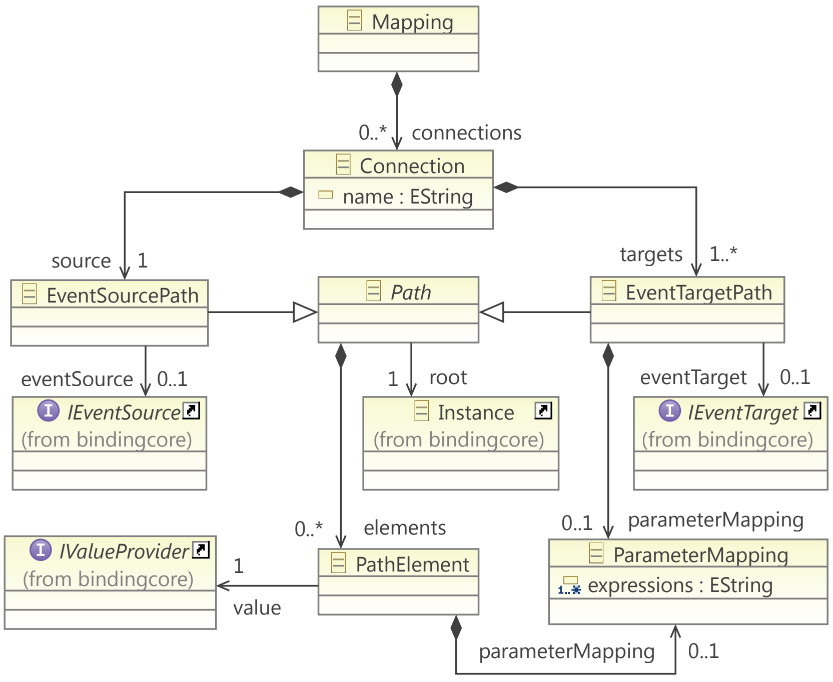


Figure 4.9: Mapping DSL

PathElement represents a single step, in the navigation from a root instance to the target instance of the targeted platform language. It references the value of the navigation step and contains a *parameterMapping* if it is needed by the *IValueProvider* referenced with *value*.

EventSourcePath is a *Path* which can be used as the source of the *Connection*. As such it references the intended *IEventSource* through the *eventSource* relation. If the root *Instance* inherited by *Path* is the element to be either monitored or the intended target of the connection, then the *eventSource* relation may not be set.

EventTargetPath is a *Path* which can be used as one of the targets of the *Connection*. As such it references the intended *IEventTarget* through the *eventTarget* relation. If the root *Instance* inherited by *Path* is the element to be either monitored or the intended target of the connection, then the *eventTarget* relation may not be set. Additionally, it references a *ParameterMapping* if it is needed by the *IEventTarget*.

ParameterMapping creates a mapping between the parameters of the element referenced by the class which contains the *ParameterMapping* and the values returned by its *expressions*, in the order the expressions are specified.

An exemplary model written in the concrete language for the *Mapping DSML* can be seen in figure 4.10 on the following page. The figure is an excerpt of the EP mapping model created for the case study in chapter 6 on page 33.

This particular model combines the Mapping DSML instance alongside with the Java and EP Binding DSML instances. It shows the connection between the source event of pressing a button named *okButton* in the *Instance* named *orderDialog* of the

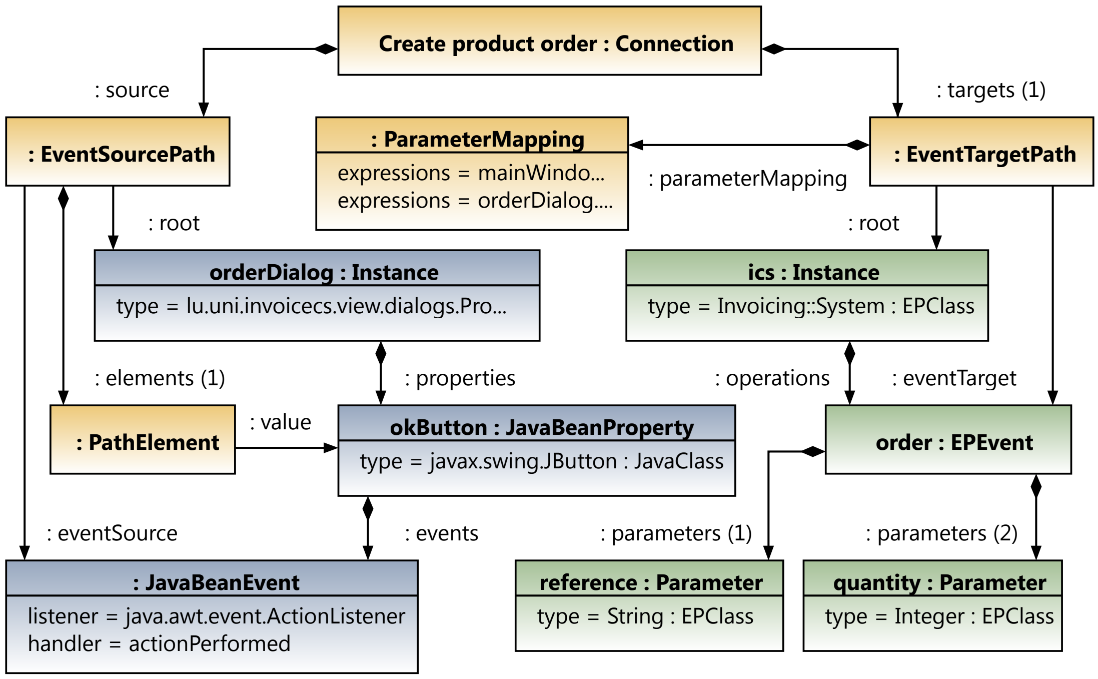


Figure 4.10: Mapping DSML instance example (with Mapping DSML elements in orange)

GUI domain and the event target named *order* in the *Instance* named *ics* in the business domain.

The event source and event target are represented as *EventSourcePath* and *EventTargetPath* respectively, following the definition of the *Mapping DSML*. The relation named *eventSource* of the *EventSourcePath* models an *ActionListener* event which, associated with a *JButton*, represents the event of pressing the represented button. The *EventTargetPath* uses a *ParameterMapping* in order to provide values to the two *Parameters* of the *order* EP event named *reference* and *quantity*.

4.6 Semantics

This section contains a natural language description of the semantics of the DSMLs. The semantics can be summarised using the following procedure:

Step 1 - Instantiation
A *Mapping DSML* instance contains a set of different *Instance* elements which are monitored for instantiation. As soon as such an instantiation event occurs, either from outside of the model or because the same *Instance* was the target of another connection, the value assigned to it is stored and step 2 follows. For example see the Connection in diagram 6.4 on page 40 in our case study. The instantiation of the *Instance* named *ics* is the entry point of the application. This event is monitored by the *Connection* in the shown diagram.

Step 2 - Attaching connections
After an *Instance* is instantiated, the *Mapping DSML* instance is queried for the set of *Connections* which use said *Instance* as the root in an *EventSourcePath*. These paths are then evaluated (path evaluation is defined later on) and listeners

are attached to the associated *IEventSources*. Diagram 4.10 on the preceding page shows an example of such an *EventSourcePath*.

Step 3 - Reacting to events
When a monitored *IEventSource* is triggered, the *EventTargetPaths* of the corresponding *Connection* are evaluated in their specified order and their associated *IEventTargets* are executed. If an *EventTargetPath* has a *ParameterMapping* assigned to it, the OCL expressions contained therein are evaluated and the resulting values are assigned to the *Parameters* of the *IEventTarget* in the order specified. Diagram 4.10 on the facing page shows an example of such an *EventSourcePath*.

In the case that an *EventTargetPath* does not have any *PathElements* set, its root *Instance* is instantiated. This in turn triggers this procedure for this particular *Instance*. For an example, see diagam 6.6 on page 42.

When a *Path* is evaluated, the values associated to the *IValueProviders* of its different *PathElements* are accessed in the order they are specified. In the case that an *IValueProvider* is not contained in the *IType* of the previously accessed *IValueProvider* or the root *Instance*, the previous *IValueProvider* or the root is cast to the expected *IType*. Lastly, if a *PathElement* has a *ParameterMapping* assigned to it, the OCL expressions contained therein are evaluated and the resulting values are assigned to the *Parameters* of the *IValueProvider* in the order specified.

CHAPTER 5

Code Generation

This chapter introduces the code generation framework. Section 5.1 explains the modelling environment used to create DSML instances. Section 5.2 illustrates the functioning of the code generator, including the template engine and the OCL translator.

5.1 Modelling Environment

The modelling environment for the *Binding DSML* and *Mapping DSML* has been created using EMF [77], making use of its ability to generate model editors for languages created using Ecore [78, 79]. The work-flow to be followed when generating a model editor with EMF is shown in figure 5.1.

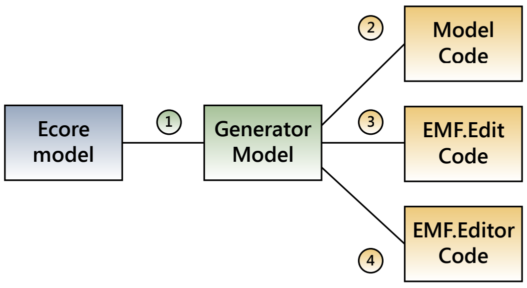


Figure 5.1: EMF Workflow

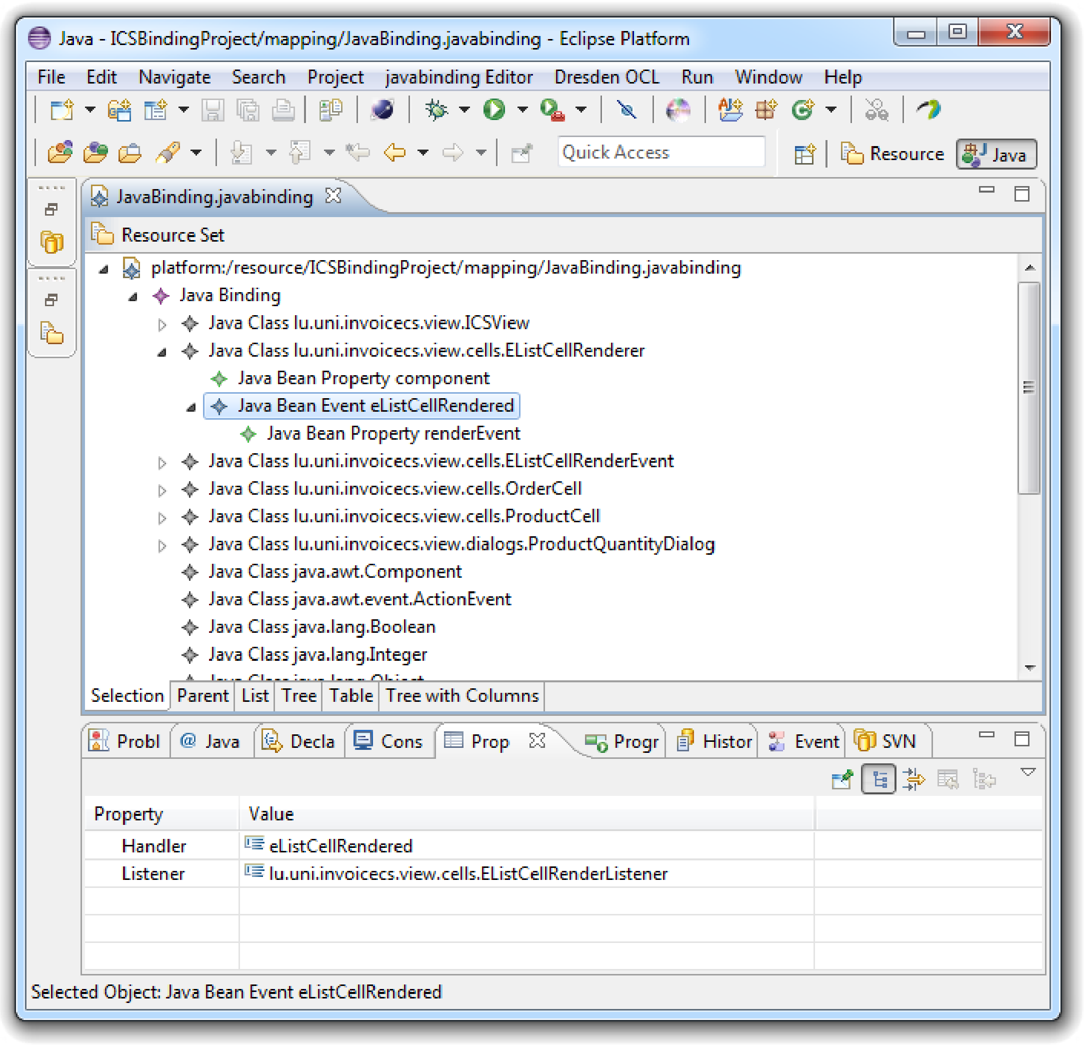
Step 1: A *Generator Model* is generated from the original Ecore model. It contains additional configurable information and properties needed for the generation of the model code and the editors.

Step 2: The *Model Code* is generated from the Generator Model. It is a Java source code representation of the original Ecore model. Every element in the model is converted to an interface with getters and setters for the elements' relations and properties as well as a class which implements this interface.

Step 3: EMF.Edit Code is generated from the Generator Model. This code is an Eclipse plugin which contains the UI-independent part of the model editor, i.e. it implements the functionality needed for Eclipse to be able to work with the generated model elements.

Step 4: EMF.Editor code is generated from the Generator Model. It represents a fully functional Eclipse editor plugin which uses and extends the EMF.Edit Code in order to allow viewing and editing model instances.

Figure 5.2 shows the generated EMF editor for the Java Bean Binding DSML. The loaded model contains the bindings used in the case study in chapter 6 on page 33.


Java - ICSBindingProject/mapping/JavaBinding.javabinding - Eclipse Platform
File Edit Navigate Search Project javabinding Editor Dresden OCL Run Window Help
Quick Access
Resource
Java
JavaBinding.javabinding
Resource Set
Selection Parent List Tree Table Tree with Columns
Probl Java Decla Cons Prop Progr Histor Event SVN

platform:/resource/ICSBindingProject/mapping/JavaBinding.javabinding
Java Binding
Java Class lu.uni.invoicecs.view.ICSView
Java Class lu.uni.invoicecs.view.cells.EListCellRenderer
Java Bean Property component
Java Bean Event eListCellRendered
Java Bean Property renderEvent
Java Class lu.uni.invoicecs.view.cells.EListCellRenderEvent
Java Class lu.uni.invoicecs.view.cells.OrderCell
Java Class lu.uni.invoicecs.view.cells.ProductCell
Java Class lu.uni.invoicecs.view.dialogs.ProductQuantityDialog
Java Class java.awt.Component
Java Class java.awt.event.ActionEvent
Java Class java.lang.Boolean
Java Class java.lang.Integer

Property	Value
Handler	eListCellRendered
Listener	lu.uni.invoicecs.view.cells.EListCellRenderListener

Selected Object: Java Bean Event eListCellRendered

Figure 5.2: EMF Editor generated for the Java Binding DSML. Containment relations are shown as a tree in the top part of the window. Non-containment relations and properties are shown in the property list in the bottom of the Window.

For more details on the generated source code as well as the edit and editor plugins, we would like to refer you to the book *EMF: Eclipse Modeling Framework* by Steinberg et al. [77].

5.2 Code Generator

The code generator created for the purposes of this thesis is made up of two main parts. First of all, the template engine FreeMarker is used to generate the source code files [24]. Secondly, an OCL translator converts any expression language code snippet encountered by the templates to source code as well.

5.2.1 Generated Source Code

The generator creates code for the Java platform, which includes elements of Aspect Oriented Programming (AOP) [21, 85]. The idea behind AOP is to achieve a higher level of source code modularisation than possible with a typical programming language by allowing the separation of cross-cutting concerns. These concerns are divided into so called *Aspects* which are combined with the base source code during compilation. The combination process is called *Weaving* and results in and application as if the Aspects' content were part of the original source code.

The role of the aspect is to monitor the instantiation of "root" *Instances*. We call root *Instance* all those instances which are *a)* the root of an *EventSourcePath* with an empty set of *PathElements* and *b)* never the root of an *EventTargetPath* with an empty set of *PathElements*. This means that the generated code is not in control of the instantiation of root instances. The issue is that Java does not create event notifications for instantiating a class, for example. Furthermore, proper instantiation depends on the target platform. Hence we use AOP to introduce the missing notification behaviour into the existing source, by generating an Aspect responsible for this task.

The role of the generated class is to set up the different connections modelled using the *Mapping DSML*. The class is instantiated by the aspect and uses a dispatcher pattern in order to connect the sources of the connections at runtime. This ensures that the source *Instances* of the different connections have been properly set up, before the connections are made.

The generator creates a single file, which contains the described aspect and class, for every root instance it detects.

5.2.2 Template engine

In a first step, the generator loads a mapping model and passes it to the template engine. The engine delegates code generation to one of a set of templates, based on the type of the element it encounters when parsing the mapping model. For example, the generation of the aspect code to monitor the creation of an EP class is handled by the `EPClass.tpl` template. The following pseudo algorithm shows how the template engine creates the source code:

```
createAspectForRootInstance();

public className() {
   createVariablesForCreatedInstances();

   public constructor() {
      for every connection
         addConnection(new Connection(
            createSourcePathDefinition();
```

```
                for every connection.target
                   createTargetPathDefinition();
                end for
             ));
          end for
       }
    }
```

Listing 5.1: Template engine pseudo algorithm

The generation of a path follows the following algorithm:

```
createRootElement(),
createEventTargetSpecification(),
for every path.element
   new PathElement(
      createValueProviderDefinition();
      createParameterMapping();
   );
end for
createParameterMapping();
```

Listing 5.2: Path definition pseudo algorithm

The difference between source and target path definitions is that the source path uses `createEventSourceSpecification()` instead of the contents of line 2 and a source path does not have a parameter mapping assigned to it.

Listing 5.3 shows the template for an EP Property. Depending on the context, the template produces different output. For example, if the template engine is currently evaluating a path and encounters an EP based value provider, it sets the context to `valueProviderInstantiation` and delegates the code generation process to the shown template.

```
<#if context == "propertyFieldDefinition">
      private ${typeMap.javaType(property.type)} ${property.name} =
          null;
<#elseif context == "eventSourceInstantiation">
      new EPProperty("${object.name}", "${object.eContainer().className
          }")
<#elseif context == "valueProviderInstantiation">
      new EPProperty("${vp.name}", "${vp.eContainer().className}")
<#else>
<#stop "Unknown context " + context>
</#if>
```

Listing 5.3: EP Property Template

5.2.3 OCL Translator

The second step of the generator is to convert OCL expressions into Java source code. This in itself is a two-step process:

1. an expression is parsed using an OCL plugin provided by the Eclipse IDE and the parsed elements are adapted to be compatible with this plugin.
2. the expression elements are converted to Java source code using platform specific mappers. For example a *JavaBeanProperty* with the name `text`, which is used as an *IValueProvider* for a *PathElement*, is converted to the string `getText()`.

5.3 Example

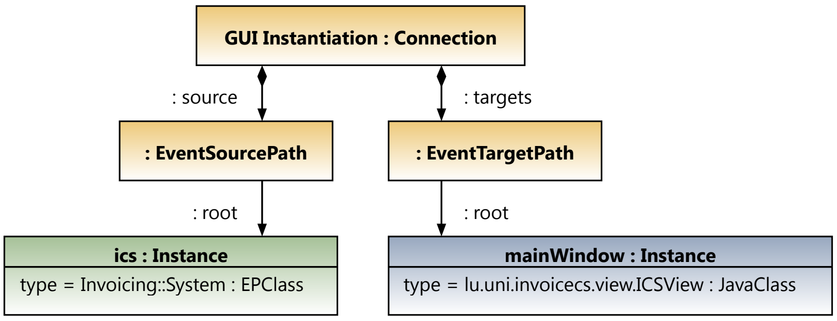

GUI Instantiation : Connection
: source
: targets
: EventSourcePath
: EventTargetPath
: root
: root
ics : Instance
type = Invoicing::System : EPClass
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass

Figure 5.3: GUI Instantiation Connection

Listing 5.4 is a small example of the generated source code. It is an excerpt from the case study in chapter 6 on page 33. The connection for which the source code is generated is shown in figure 5.3.

```
public aspect icsAspect {
   after(): call(* *.initProps()) && target(generated.Invoicing.System) {
      final generated.Invoicing.System object = (generated.Invoicing.System)
          thisJoinPoint.getTarget();
      new icsClass(object);
   }

   public class icsClass extends mapping.runtime.Mapping {
      private final generated.Invoicing.System ics;

      private lu.uni.invoicecs.view.ICSView mainWindow = null;

      public icsClass(generated.Invoicing.System ics_param) {
         this.ics = ics_param;

         /* GUI Instantiation ============================================== */
         this.addConnection(
            new Connection(
               new EventSourcePath(
                  /* root */
                  new EPClass.EventSource("ics", "generated.Invoicing.System")
                      {
                     public Object getValue(Object input, List parameterValues)
                          {
```

```
                        return ics;
                     }
                  }
               ),

               new EventTargetPath(
                  /* root */
                  new JavaClass.EventTarget("lu.uni.invoicecs.view.ICSView") {
                     public Object getValue(Object input, List parameterValues)
                          {
                        return mainWindow;
                     }

                     public void trigger(Object source, List parameterValues) {
                        Object instance = this.createInstance(parameterValues);
                        mainWindow = (lu.uni.invoicecs.view.ICSView) instance;
                        Dispatcher.getInstance().fireJavaClassInstantiated("
                            mainWindow", this.className, instance);
                     }
                  }
               )
            )
         );
      }
   }
}
```

Listing 5.4: Generated source code for the *GUI Instantiation* connection

CHAPTER 6

Case study

In this chapter we will illustrate a proof of concept implementation based on a case study developed by Henri Habrias [37]. The study defines a rudimentary invoicing system for which we created:

- abstract models representing the business logic of the invoicing system. The "back-end" source code is generated entirely from these models.
- a Graphical user Interface (GUI) which allows to interact visually with the invoicing system. The "front-end" GUI was modelled using a platform specific tool, which in turn created the corresponding source code.

Afterwards we will show how the Mapping and Bridging DSMLs are applied in order to interface the abstract business logic with the concrete GUI.

This particular study was chosen due to its size and complexity. The invoicing system described by the study is rather small, hence it does not introduce too many repetitious tasks. Additionally, such a system is well suited to be interacted with using a GUI.

6.1 Case Study Definition

Henri Habrias's case study is defined by the following text [37, 86]:

1. The subject is to invoice orders.
2. To invoice is to change the state of an order (to change it from the state "pending" to "invoiced").
3. On an order, we have one and one only reference to an ordered product of a certain quantity. The quantity can be different to other orders.
4. The same reference can be ordered on several different orders.

5. The state of the order will be changed into "invoiced" if the ordered quantity is either less or equal to the quantity which is in stock according to the reference of the ordered product.
6. You have to consider the two following cases:
   (a) Case 1
   All the ordered references are references in stock. The stock or the set of the orders may vary:
      - due to the entry of new orders or cancelled orders;
      - due to having a new entry of quantities of products in stock at the warehouse.

   However, we do not have to take these entries into account. This means that you will not receive two entry flows (orders, entries in stock). The stock and the set of orders are always given to you in an up-to-date state.

   (b) Case 2
   You do have to take into account the entries of:
      - new orders;
      - cancellations of orders;
      - entries of quantities in the stock.

For the purposes of the proof of concept implementation, we skipped case 1 of the case study and immediately implemented case 2. The remainder of the design choices will be described in section 6.3 on the next page.

6.2 Target language and platform

The modelling language chosen for the case study is EP. The reason behind this choice is that EP is executable while still being a small language compared to others, e.g. Executable UML [34]. Hence the entire business logic of the case study is designed using a single language and only a few models. Furthermore, the Democles tool with which to create the EP system, readily provides a code generator targeting the Java programming language. The latter is also the reason for choosing Java Swing, a flexible and easy to use GUI framework for Java, to create the case study's GUI.

6.2.1 Democles & EP

We introduced the EP modelling language in section 4.4 on page 18. EP based models are created using the Democles tool [55], a plugin for the Eclipse IDE. The plugin also provides a code generator which generates source code targeting the Java language from said EP models [26]. Furthermore, Democles allows to interactively execute an EP system and visually represent the system's state for debugging purposes.

6.2.2 Window Builder & Java Swing

Swing, part of the Java Foundation Classes, is based on the Abstract Window Toolkit and provides Widgets and an API for creating GUIs for Java programs. Apart from a few components, Swing is platform independent as it makes use of Java 2D to draw its graphical components and it uses its own typed event system [82, 81].

The de-facto standard for creating GUIs based on Swing using the Eclipse IDE is the Window Builder plugin [82, 47]. Similarly to other if not most GUI builders, WindowBuilder allows to create GUIs visually using drag and drop and a properties view for the different GUI elements and their respective layouts. The visual model created this way is used by WindowBuilder to generate the corresponding Java code.

6.3 Implementation

The abstract EP models and the concrete Java Swing GUI have been developed independently of each other. The EP models are self-contained and provide only the functionality necessary to implement the case study. The GUI provides the means to visually represent the data relevant to the study, but it does not know where or how this data is stored. For more details on the implementation of the business models, please see appendix A on page 59.

6.3.1 Business logic

The business logic of the case study is made up of three different EP classes, which we will explain in this section. These models are part of the domain named *Invoicing* while a fourth model, named *Main* in the *Application* bridge, instantiates the invoicing system.[1]

The static structure

The EP model named *System* handles the collections of products and orders of the invoicing system. These are typed as OCL sets, meaning that the order of the different items they contain has no importance and they do not contain the same object more than once.

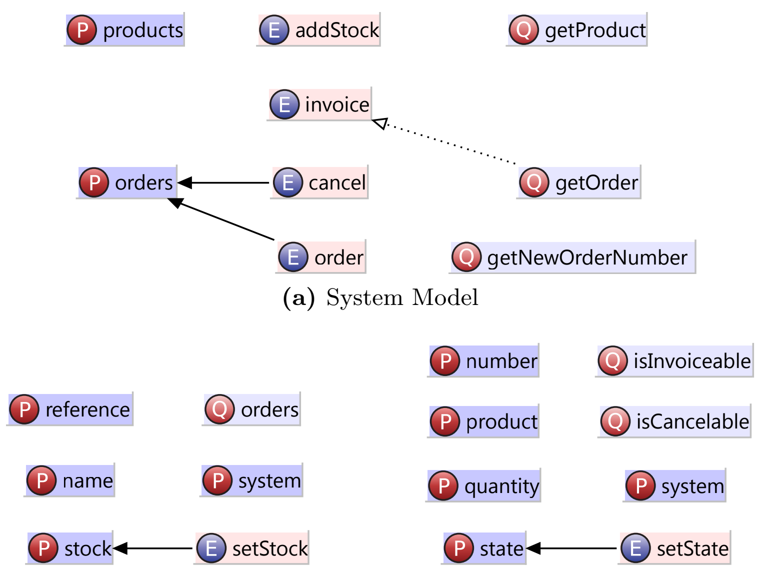
As already mentioned, the case study defines two EP models *Product* and *Order*. A product has three properties, namely *reference*, *name* and *stock*. The *reference* property is typed as *String* and should be unique for any given model instance. It is used by an order to “reference” the ordered product. Since proper stock management - product and stock management, (re)stocking, etc. - is outside of the scope of the case study, we simply included the information on how much stock of a product is available in the *Integer* typed property *stock*.

An order has as properties *number*, *product*, *quantity* and *state*. The property *number* is typed as *Integer* and should, like the *Product reference* property, be unique for a given model instance. *Product* is the reference to the ordered product and *quantity*, an Integer, is the number of ordered products.

The state property is a special case. The problem is that an order’s state should only be “pending” or “invoiced”, but EP does not support enumerations. Therefore state is of type String and proper model execution ensures that an order’s state will only take on one of the two allowed values.

Only a single instance of the *System* model is created at runtime. The *Order* and the *Product* models contain query properties which require access to the *products*

[1]The entry point of every EP system is a model named *Main* in the *Application* bridge



(a) System Model

(b) Product Model **(c)** Order Model

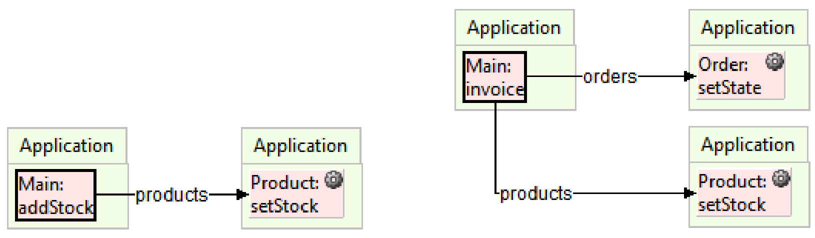
Figure 6.1: The case study business models. The static structure is represented as properties P and the dynamic behaviour as query properties Q and events E.

and *orders* sets of the *System* model, hence they each have a reference to the system instance, stored in the properties named *system* (c.f. figures 6.1b and 6.1c).

The dynamic behaviour

The system behaviour is defined in the four events seen in figure 6.1a. Creating or cancelling an order affects the set of orders managed by the system by adding or removing an item to or from the set named *orders*. Hence, both *cancel* and *order* have an impact link to said set.

addStock as well as *invoice* are examples of events which are forwarded into an OCL collection (c.f. figure 6.2). By using a link guard, it is ensured that the *addStock* event will in the end only trigger the *setStock* event of the targeted product instance.



(a) Add stock **(b)** Invoice

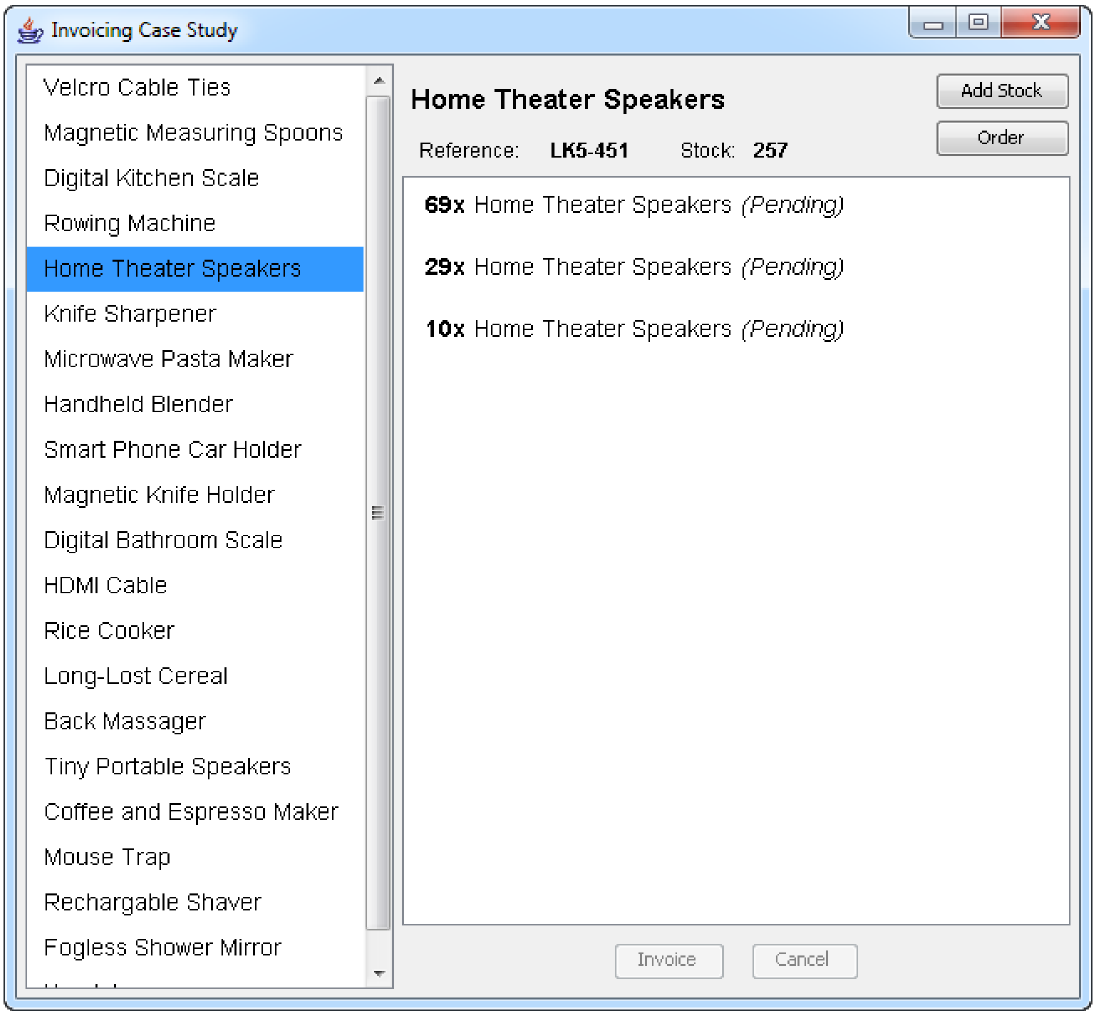
Figure 6.2: Adding stock and invoicing event trees

getProduct and *getOrder* are query properties which return the respective model instance with the given product reference or order number. *getNewOrderNumber* is a query property used by the *order* event. It returns a hitherto unused number which is to be used as an identifier for a new *Order* model instance.

The *orders* query property of the *Product* model simply returns a set containing the orders, which reference the given product. The *isInvoicable* and *isCancelable* query properties of the *Order* model return values stating whether the given order can be invoiced, i.e. it is in a pending state and the ordered product has enough stock available, or whether it can be cancelled, i.e. it is in a pending state, that is, it has not been invoiced already.

6.3.2 Graphical User Interface

As mentioned previously, the case study's GUI has been created using the Window Builder Eclipse plugin. It is composed of multiple custom panels, which have been combined into the window you can see in figure 6.3. The excerpt in listing 6.1 is an example of the source code generated by Window Builder. It shows part of the code for the product details panel in the upper right of the window.[2]



Velcro Cable Ties
Magnetic Measuring Spoons
Digital Kitchen Scale
Rowing Machine
Home Theater Speakers
Knife Sharpener
Microwave Pasta Maker
Handheld Blender
Smart Phone Car Holder
Magnetic Knife Holder
Digital Bathroom Scale
HDMI Cable
Rice Cooker
Long-Lost Cereal
Back Massager
Tiny Portable Speakers
Coffee and Espresso Maker
Mouse Trap
Rechargable Shaver
Fogless Shower Mirror

Home Theater Speakers

Reference: **LK5-451** Stock: **257**

69x Home Theater Speakers *(Pending)*

29x Home Theater Speakers *(Pending)*

10x Home Theater Speakers *(Pending)*

Figure 6.3: GUI Main Window

[2]The code for the GridBagLayout Layout Manager has been removed from the code. For more information on Swing Layout Managers, please see [47] and [82].

```
public class ProductDetailsPanel extends JPanel {
   private JLabel lblProductName = null;
   private final JButton btnOrder;

   public ProductDetailsPanel() {
      this.setPreferredSize(new Dimension(400, 62));

      this.lblProductName = new JLabel("");
      this.lblProductName.setFont(new Font("Arial", Font.BOLD, 17));
      this.add(this.lblProductName, ...);

      final JPanel buttons = new JPanel();
      this.add(buttons, ...);
      buttons.setLayout(new BoxLayout(buttons, BoxLayout.Y_AXIS));

      this.btnOrder = new JButton("Order");
      this.btnOrder.setMaximumSize(new Dimension(81, 23));
      buttons.add(this.btnOrder);
   }

   public JLabel getProductNameLabel() {
      return this.lblProductName;
   }

   public JButton getOrderButton() {
      return this.btnOrder;
   }
}
```

Listing 6.1: Excerpt from the source code generated by WindowBuilder for the product details panel. It contains only the code for one label and one button and the code for the used layout manager has been removed for brevity.

Window Builder allows to *expose* components by creating getter methods, which return a reference to said component. We made use of this feature throughout the entire GUI and exposed every component used for displaying data from the business models (e.g. see the get... methods in listing 6.1 on the preceding page). This way, the mapping model created later on can access the components and more importantly its properties and methods.

Event-driven JList

While the creation of most of the GUI was straightforward, the two lists used, i.e. the list of products and the list of orders, required more attention. In order to understand the problem, first an explanation of how JList manages and renders its content is required.

The Swing JList follows the Model-View-Controller (MVC) pattern in that it manages the data to show and the item selection in two different models [82]. By default the content is rendered by calling `toString()` in every object in the data model. If the default behaviour does not yield the desired result, a custom `ListCellRenderer` is required.

A `ListCellRenderer` allows using any component to format the data of an item in the data model [82] for displaying. The case study GUI uses a custom panel with three different labels to display the data of an Order (c.f. figure 6.3 on page 37). Now the problem is that the custom renderer does not create multiple instances of the component, but uses a single one to render every model item. Each renderer is buffered and added to the overall list content. Although this results in high performance and low memory requirements, it also means that the different labels will not be accessible after they have been rendered. Hence we could not use the same approach with the two lists as with the rest of the GUI and simply expose the different components in order to set the labels' text property.

To overcome this problem, we implemented a class named `EListCellRenderer`, an event based custom `ListCellRenderer`. This custom renderer has been designed to work with any custom component used to render the list content and maintains a reference to this component. Instead of rendering the component itself when asked by the list the renderer is assigned to, it notifies a list of registered `EListCellRenderListener` event listeners and provides them with an `EListCellRenderEvent` event object, which contains all the information needed to actually render the custom component [82]. WindowBuilder does not provide a means to set the `cellRenderer` property of a list, which we therefore had to do manually by extending the source code generated by WindowBuilder.

The custom renderer being event based, allows the creation of a mapping using the Mapping DSL to set the values necessary for proper rendering. Note that this method works only because Java Swing GUIs are single-threaded. This means that a single thread is responsible for the entire GUI, including any event-handling routines. Hence it is assured that, by using the `EListCellRenderer`, the content of the custom component maintained by the renderer is set to the proper values, before it is finally rendered for a given list item. Please see the diagrams 6.6 on page 42 and 6.7 on page 43 for examples on how a mapping is used in combination with the `EListCellRenderer` and see A.2.2 on page 65 for implementation and design details.

6.4 Bridging EP and Java Swing

With both the EP based business domain and the Java Swing GUI implemented, the next step is to create binding models for both followed by the mapping model. In this section we will illustrate the different models using the concrete syntax presented in chapter 4 on page 13. Please note that the diagrams in this section each represent a single connection and include, of the two binding models, only the parts relevant to that particular connection. Due to the graphical nature of the chosen concrete syntax, diagrams of the binding models themselves would be too big to include in this thesis.

The different connections of the created mapping model instance are organised into the following categories:

GUI Instantiation
The connections in this category properly set up the application, including showing the main window and populating the two lists.

List selection
These connections handle what happens when an item is selected in either list.

Product actions
A product may be ordered or stocked up. This part of the mapping model shows how these actions are handled.

Order actions
Similarly, these connections show how the actions on an order are handled.

6.4.1 GUI Instantiation

The *GUI Instantiation* connection (c.f. diagram 6.4) has a single source and a single target, both of which only have their respective root references set. According to the mapping model semantics, this connection monitors the instantiation of the instance named *ics*, i.e. the creation of an instance of the *System* model in the business domain, and reacts to the event by creating the instance named *mainWindow*, i.e. the GUI of the application. Please note, that the entry point of the application is the model *Main* in the EP bridge named *Application* in the business domain and the GUI has been configured to show the main window immediately after instantiation.

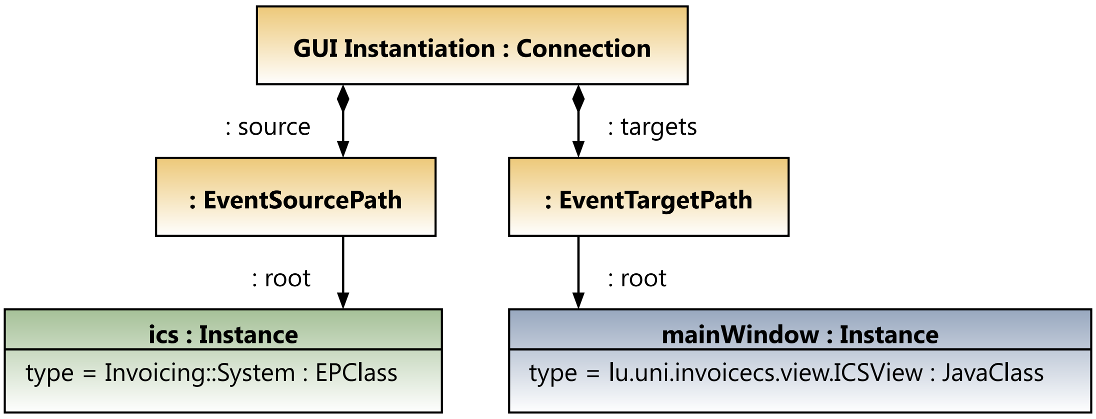

GUI Instantiation : Connection
: source
: targets
: EventSourcePath
: EventTargetPath
: root
: root
ics : Instance
type = Invoicing::System : EPClass
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass

Figure 6.4: Instantiation of the GUI

The *Populate Lists on GUI Instantiation* connection (c.f. diagram 6.5 on the next page), as the name suggests, handles the task of populating the two lists used. To that effect, it monitors the *mainWindow* instance for instantiation, just as the previous connection monitored the instance named *ics*. When *mainWindow* is created - for this particular application by the previous connection -, the connection assigns the parameter mapping values of its respective event target paths to the *listData* property of their respective lists.

The parameter mapping expressions used are `ics.products->asSequence()` and `ics.orders->asSequence()` respectively. `asSequence()` is called on both sets in the *ics* instance, since the type of the target property is `java.lang.Vector` which is logically incompatible with an OCL set.

Since the lists are being populated with data, the custom list cell renderers need to be adressed, as discussed in section 6.3.2 on page 38. The two connections seen in the diagrams 6.6 on page 42 and 6.7 on page 43 handle the `EListCellRender-Event` events triggered by the renderers. The source paths navigate to their respective lists and listen to the described event type using the custom event listener named `EListCellRenderListener`.

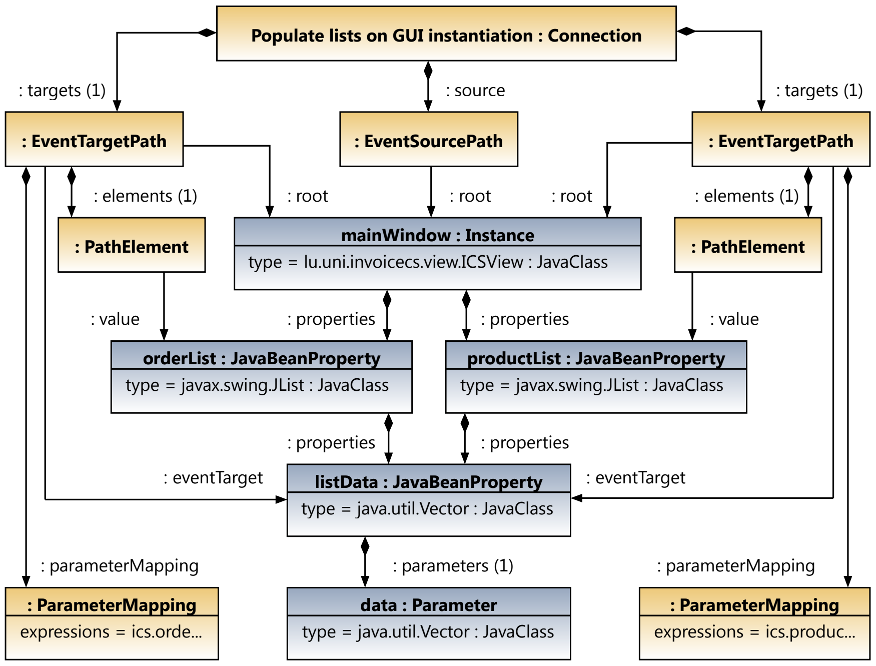

Populate lists on GUI instantiation : Connection
: targets (1)
: source
: targets (1)
: EventTargetPath
: EventSourcePath
: EventTargetPath
: elements (1)
: root
: root
: root
: elements (1)
: PathElement
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: PathElement
: value
: properties
: properties
: value
orderList : JavaBeanProperty
type = javax.swing.JList : JavaClass
productList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: properties
: properties
: eventTarget
listData : JavaBeanProperty
type = java.util.Vector : JavaClass
: eventTarget
: parameterMapping
: parameters (1)
: parameterMapping
: ParameterMapping
expressions = ics.orde...
data : Parameter
type = java.util.Vector : JavaClass
: ParameterMapping
expressions = ics.produc...

Figure 6.5: Populate Lists on GUI Instantiation

Of note are these connections, as their respective *EventSourcePaths* and some of the *value* references of the *PathElements* reference elements which do not necessarily belong on a path which starts at an *Instance*. Taking the *Handle product list cell rendering* connection as an example, the *eventSource* relation of the *EventSourcePath* expects the modelled *JavaBeanEvent* shown in the diagram. This type of event belongs to the *IType* *lu.uni.invoicecs.view.cells.EListCellRenderer*, different from the *EventSourcePath*'s last *PathElement*. As defined in chapter 4 on page 13, this represents a type cast.

The event target for both connections is a *Java method* named `render` which is implemented by both custom list cell components. The difference between them is that the render method for the order list has more parameters than the other one. These parameters are mapped to values using the *ParameterMappings* shown in the diagram which make use of data stored in the *renderEvent* event object, supplied by the *eListCellRendered* event.

The parameter expressions for *Handle product list cell rendering* are:

- `renderEvent.source`
- `renderEvent.value.oclAsType(Invoicing::Product).name`
- `renderEvent.selected`

And for the *Handle order list cell rendering* connection they are:

- `renderEvent.source`
- `ics.getProduct(renderEvent.value.oclAsType(Invoicing::Order).product).name`
- `renderEvent.value.oclAsType(Invoicing::Order).quantity`
- `renderEvent.value.oclAsType(Invoicing::Order).state`
- `renderEvent.selected`

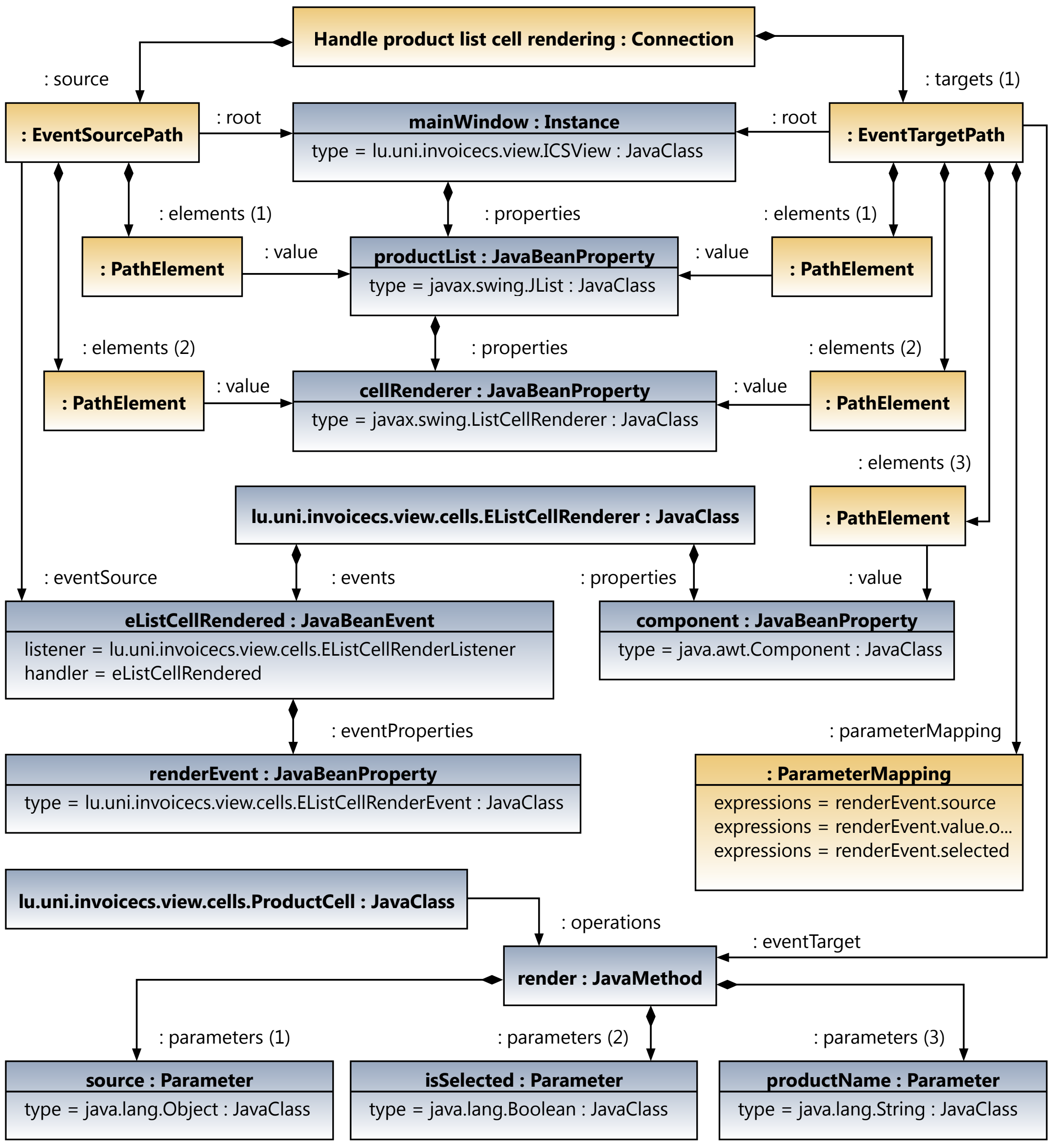

Handle product list cell rendering : Connection
: source
: targets (1)
: EventSourcePath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: root
: EventTargetPath
: elements (1)
: properties
: elements (1)
: PathElement
: value
productList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: value
: PathElement
: elements (2)
: properties
: elements (2)
: PathElement
: value
cellRenderer : JavaBeanProperty
type = javax.swing.ListCellRenderer : JavaClass
: value
: PathElement
: elements (3)
lu.uni.invoicecs.view.cells.EListCellRenderer : JavaClass
: PathElement
: eventSource
: events
: properties
: value
eListCellRendered : JavaBeanEvent
listener = lu.uni.invoicecs.view.cells.EListCellRenderListener
handler = eListCellRendered
component : JavaBeanProperty
type = java.awt.Component : JavaClass
: eventProperties
: parameterMapping
renderEvent : JavaBeanProperty
type = lu.uni.invoicecs.view.cells.EListCellRenderEvent : JavaClass
: ParameterMapping
expressions = renderEvent.source
expressions = renderEvent.value.o...
expressions = renderEvent.selected
lu.uni.invoicecs.view.cells.ProductCell : JavaClass
: operations
: eventTarget
render : JavaMethod
: parameters (1)
: parameters (2)
: parameters (3)
source : Parameter
type = java.lang.Object : JavaClass
isSelected : Parameter
type = java.lang.Boolean : JavaClass
productName : Parameter
type = java.lang.String : JavaClass

Figure 6.6: Handle Product List Cell Rendering

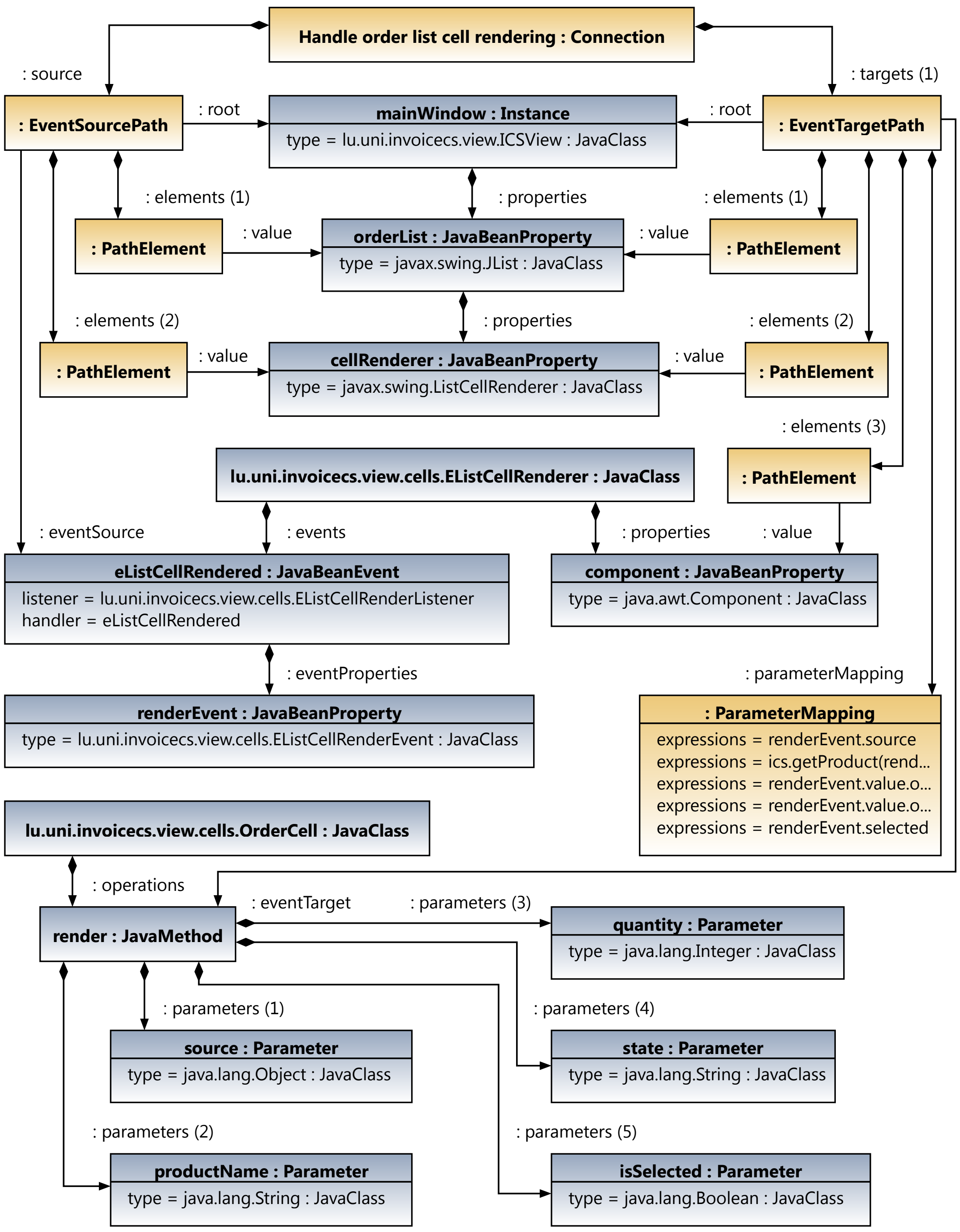

Handle order list cell rendering : Connection
: source
: targets (1)
: EventSourcePath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: root
: EventTargetPath
: elements (1)
: properties
: elements (1)
: PathElement
: value
orderList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: value
: PathElement
: elements (2)
: properties
: elements (2)
: PathElement
: value
cellRenderer : JavaBeanProperty
type = javax.swing.ListCellRenderer : JavaClass
: value
: PathElement
: elements (3)
lu.uni.invoicecs.view.cells.EListCellRenderer : JavaClass
: PathElement
: eventSource
: events
: properties
: value
eListCellRendered : JavaBeanEvent
listener = lu.uni.invoicecs.view.cells.EListCellRenderListener
handler = eListCellRendered
component : JavaBeanProperty
type = java.awt.Component : JavaClass
: eventProperties
: parameterMapping
renderEvent : JavaBeanProperty
type = lu.uni.invoicecs.view.cells.EListCellRenderEvent : JavaClass
: ParameterMapping
expressions = renderEvent.source
expressions = ics.getProduct(rend...
expressions = renderEvent.value.o...
expressions = renderEvent.value.o...
expressions = renderEvent.selected
lu.uni.invoicecs.view.cells.OrderCell : JavaClass
: operations
: eventTarget
: parameters (3)
render : JavaMethod
quantity : Parameter
type = java.lang.Integer : JavaClass
: parameters (1)
: parameters (4)
source : Parameter
type = java.lang.Object : JavaClass
state : Parameter
type = java.lang.String : JavaClass
: parameters (2)
: parameters (5)
productName : Parameter
type = java.lang.String : JavaClass
isSelected : Parameter
type = java.lang.Boolean : JavaClass

Figure 6.7: Handle Order List Cell Rendering

6.4.2 List selection

Selecting an item in the list of products has multiple consequences. First of all, as shown by diagram 6.8, the product details panel is populated with the data of the selected item. The *Populate product details on product selection* connection listens to the selection of an item in the list of products, i.e. the connection source, and, using an *EventTargetPath* for each of the three targeted labels, sets the labels' text property to the value if the respective parameter mappings.

The three parameter mapping expressions all are similar to the code in listing 6.2, respectively setting the name, reference and stock of the selected product.

```
if not mainWindow.productList.isSelectionEmpty() then
  mainWindow.productList.selectedValue.oclAsType(Invoicing::Product).
      name
else
  ''
endif
```

Listing 6.2: Exemplary parameter mapping used for populating product details

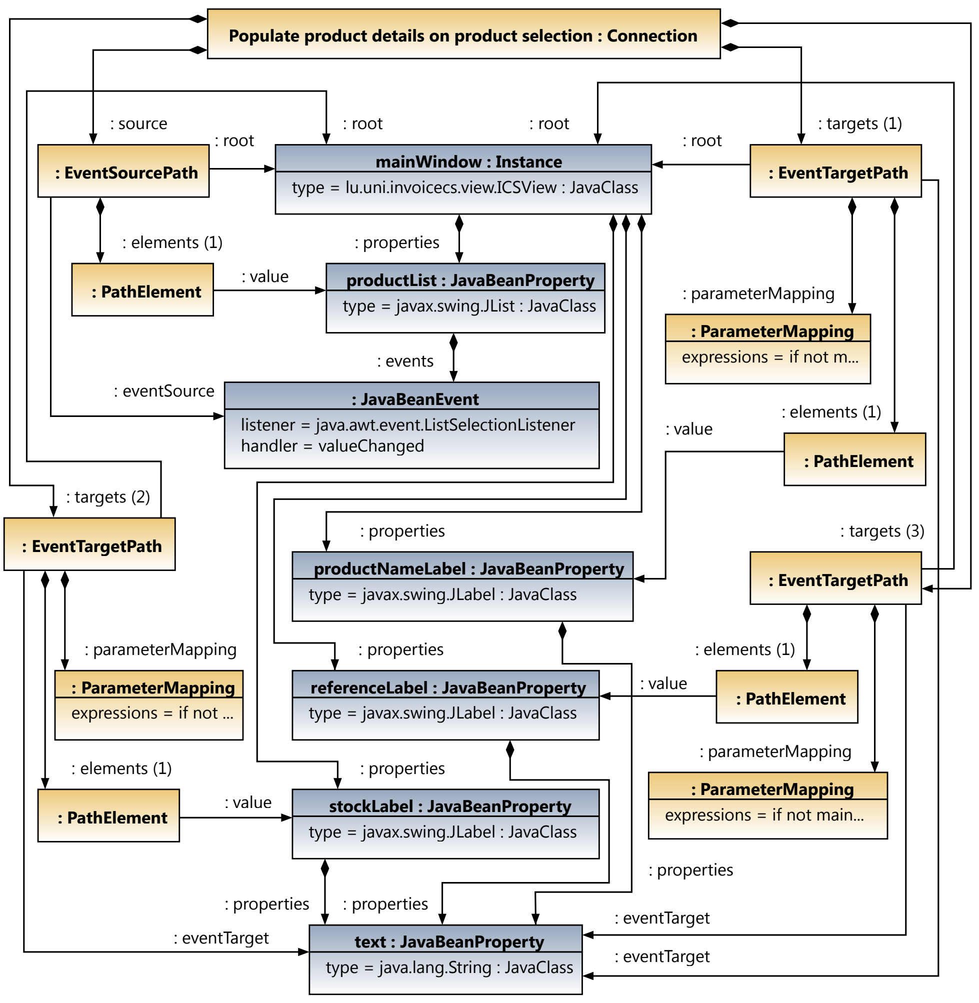


Figure 6.8: Populate product details on product selection

Secondly, the product details panel is shown to the user. The *Show/Hide product details on product selection* connection listens, exactly like the previous connection, to the selection of an item in the list of products. It reacts to this event by calling the method `displayProductDetails`. For more details on this method, please see section A.2.2 on page 65.

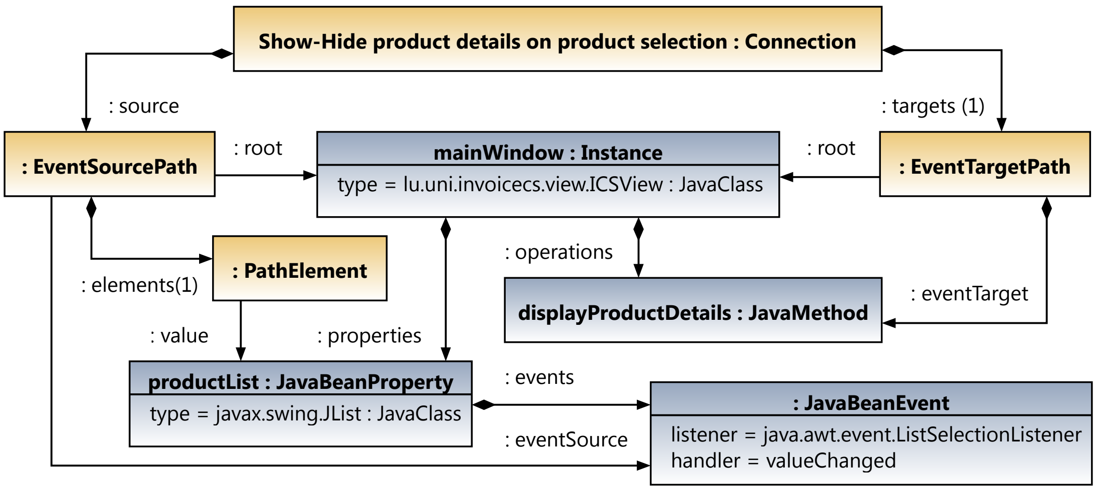

Show-Hide product details on product selection : Connection
: source
: targets (1)
: EventSourcePath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: root
: EventTargetPath
: PathElement
: elements(1)
: operations
displayProductDetails : JavaMethod
: eventTarget
: value
: properties
productList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: events
: JavaBeanEvent
listener = java.awt.event.ListSelectionListener
handler = valueChanged
: eventSource

Figure 6.9: Show/Hide product details on product selection

Lastly, the content of the list of orders is altered to only contain those orders related to the selected product. This connection, named *Populate order list on product selection*, also listens to the selection of an item in the list of products. After a product has been selected, the connection's target sets the `listData` property of the order list to the value in the *EventTargetPath*'s parameter mapping.

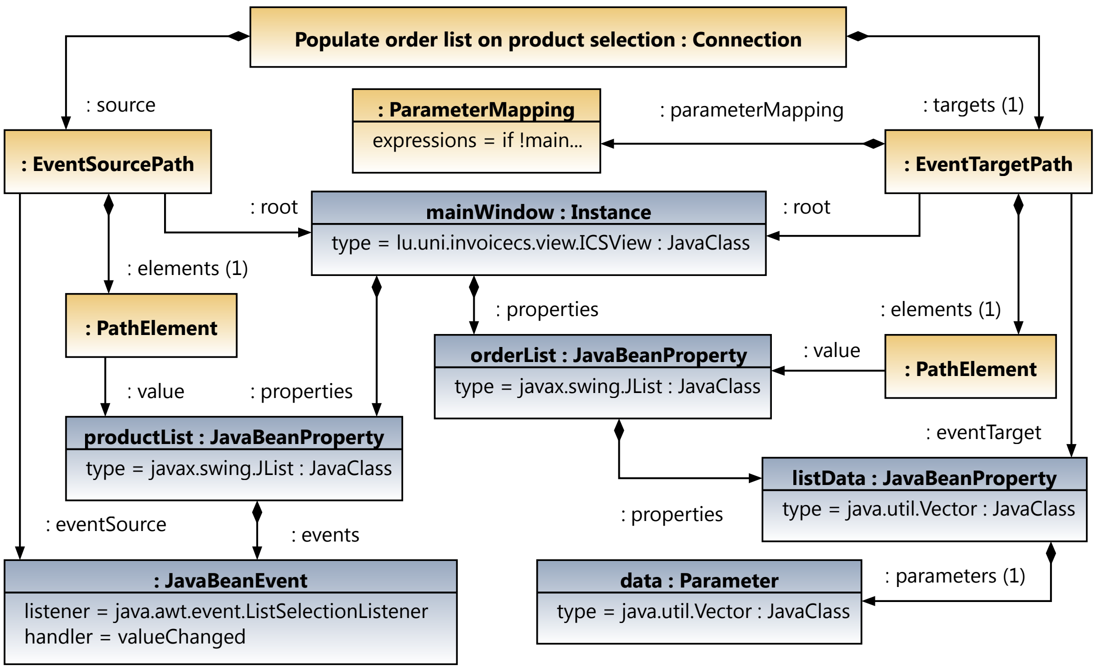

Populate order list on product selection : Connection
: source
: ParameterMapping
expressions = if !main...
: parameterMapping
: targets (1)
: EventSourcePath
: EventTargetPath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: root
: elements (1)
: PathElement
: properties
orderList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: elements (1)
: value
: PathElement
: value
: properties
productList : JavaBeanProperty
type = javax.swing.JList : JavaClass
: eventTarget
listData : JavaBeanProperty
type = java.util.Vector : JavaClass
: properties
: eventSource
: events
: JavaBeanEvent
listener = java.awt.event.ListSelectionListener
handler = valueChanged
data : Parameter
type = java.util.Vector : JavaClass
: parameters (1)

Figure 6.10: Populate order list on product selection

The parameter mapping expression used in the *Populate order list on product selection* connection is:

```
if mainWindow.productList.isSelectionEmpty() then
   ics.orders->asSequence()
else
   mainWindow.productList.selectedValue.oclAsType(Invoicing::Product).
       orders->asSequence()
endif
```

Listing 6.3: *Populate order list* connection parameter expressions

Selecting an item in the list of orders is the event source for the *Order button enablement on order selection* connection, as can be seen in diagram 6.11. As a result, two buttons, namely `invoiceButton` and `cancelButton`, are enabled or disabled depending on the values of the parameter expressions of their respective *EventTargetPaths*.

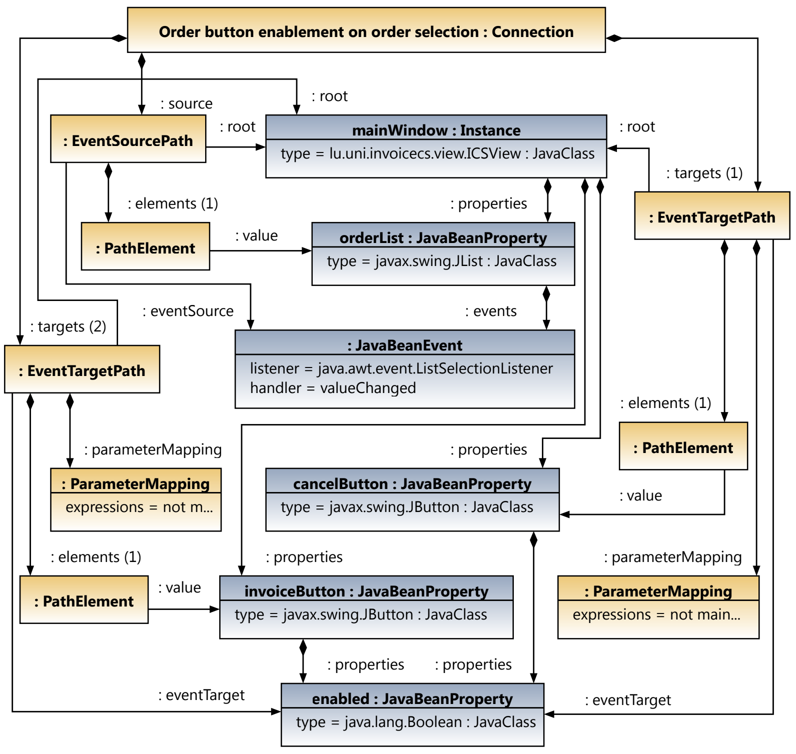


Figure 6.11: Order button enablement on order selection

The parameter mapping expressions used in the *Order button enablement on order selection* connection are:

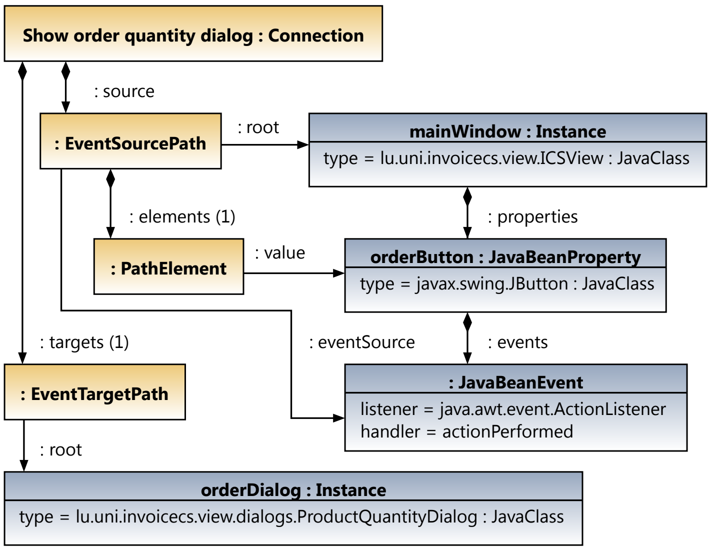
```
not mainWindow.orderList.isSelectionEmpty() and mainWindow.orderList.
    selectedValue.oclAsType(Invoicing::Order).isInvoicable
not mainWindow.orderList.isSelectionEmpty() and mainWindow.orderList.
    selectedValue.oclAsType(Invoicing::Order).isCancelable
```

Listing 6.4: *Order button enablement on order selection* connection parameter expressions for the invoice and the cancel button (in that order)

6.4.3 Product actions

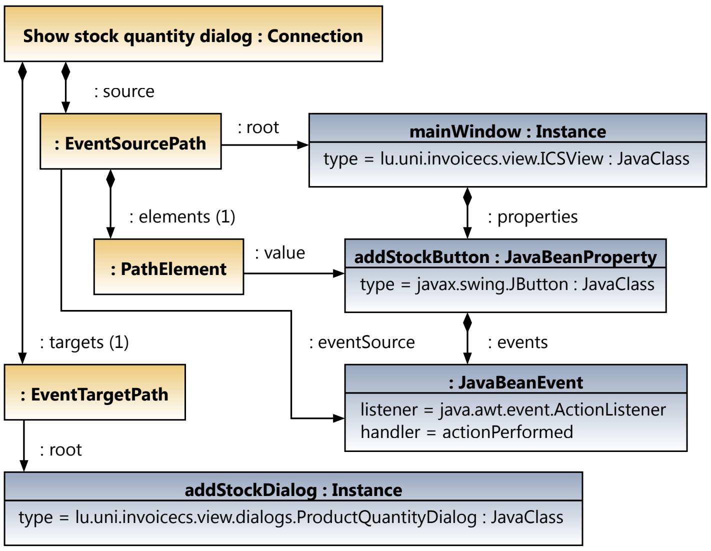
According to the case study definition, a given product may be ordered or restocked. In order to do so, the two buttons named `orderButton` and `addStockButton` will instantiate a dialog asking for the quantity of product to order or the amount to restock the selected product with. The former behaviour is shown in the diagrams 6.12 and 6.13.

The connections named *Show order quantity dialog* and *Show stock quantity dialog* react to one of the two buttons being pressed, i.e. the *EventSource-Paths*, by creating a new instance of the custom dialog `ProductQuantityDialog`, i.e. the *EventTargetPaths*. These instances are of the same type but have different names. The event handling of these dialogs is completely independent of this particular connection and the instance names allow the mapping system to differentiate between them.



Show order quantity dialog : Connection
: source
: EventSourcePath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: elements (1)
: properties
: PathElement
: value
orderButton : JavaBeanProperty
type = javax.swing.JButton : JavaClass
: targets (1)
: eventSource
: events
: EventTargetPath
: JavaBeanEvent
listener = java.awt.event.ActionListener
handler = actionPerformed
: root
orderDialog : Instance
type = lu.uni.invoicecs.view.dialogs.ProductQuantityDialog : JavaClass

Figure 6.12: Show order quantity dialog



Show stock quantity dialog : Connection
: source
: EventSourcePath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: elements (1)
: properties
: PathElement
: value
addStockButton : JavaBeanProperty
type = javax.swing.JButton : JavaClass
: targets (1)
: eventSource
: events
: EventTargetPath
: JavaBeanEvent
listener = java.awt.event.ActionListener
handler = actionPerformed
: root
addStockDialog : Instance
type = lu.uni.invoicecs.view.dialogs.ProductQuantityDialog : JavaClass

Figure 6.13: Show stock quantity dialog

The latter behaviour described above is shown in the diagrams 6.14 and 6.15 on the next page. A `ProductQuantityDialog` contains a button `okButton` which is the event source for both connections. The *Create product order* connection reacts to this event by calling the business domain operation *order* and filling its parameters with the values from the following parameter mapping expressions:

```
mainWindow.productList.selectedValue.oclAsType(Invoicing::Product).
    reference
orderDialog.amount
```

Listing 6.5: *Create product order* connection parameter expressions

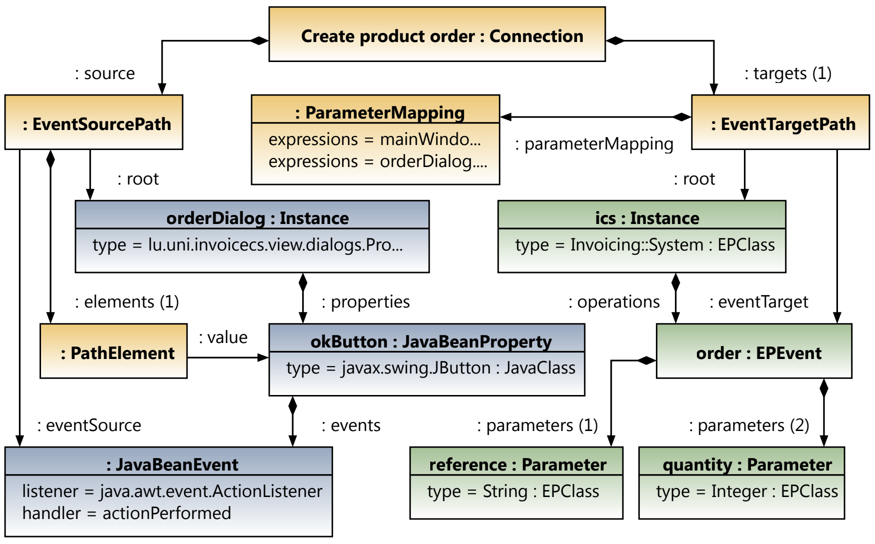


Figure 6.14: Create product order

Similarly, the *Add product stock* connection reacts to this event, by calling the business domain operation *addStock* and filling its parameters with the values from the following parameter mapping expressions:

```
mainWindow.productList.selectedValue.oclAsType(Invoicing::Product).
    reference
addStockDialog.amount
```

Listing 6.6: *Add product stock* connection parameter expressions

Since the stock of the currently selected product is shown using a label (c.f. diagram 6.8 on page 44), the label needs to be updated at this point. The bottom left *EventTargetPath* fulfils this function by setting the label's text property to the value of the following parameter mapping expression:

```
mainWindow.productList.selectedValue.oclAsType(Invoicing::Product).
    stock
```

Listing 6.7: *Add product stock* connection parameter expressions

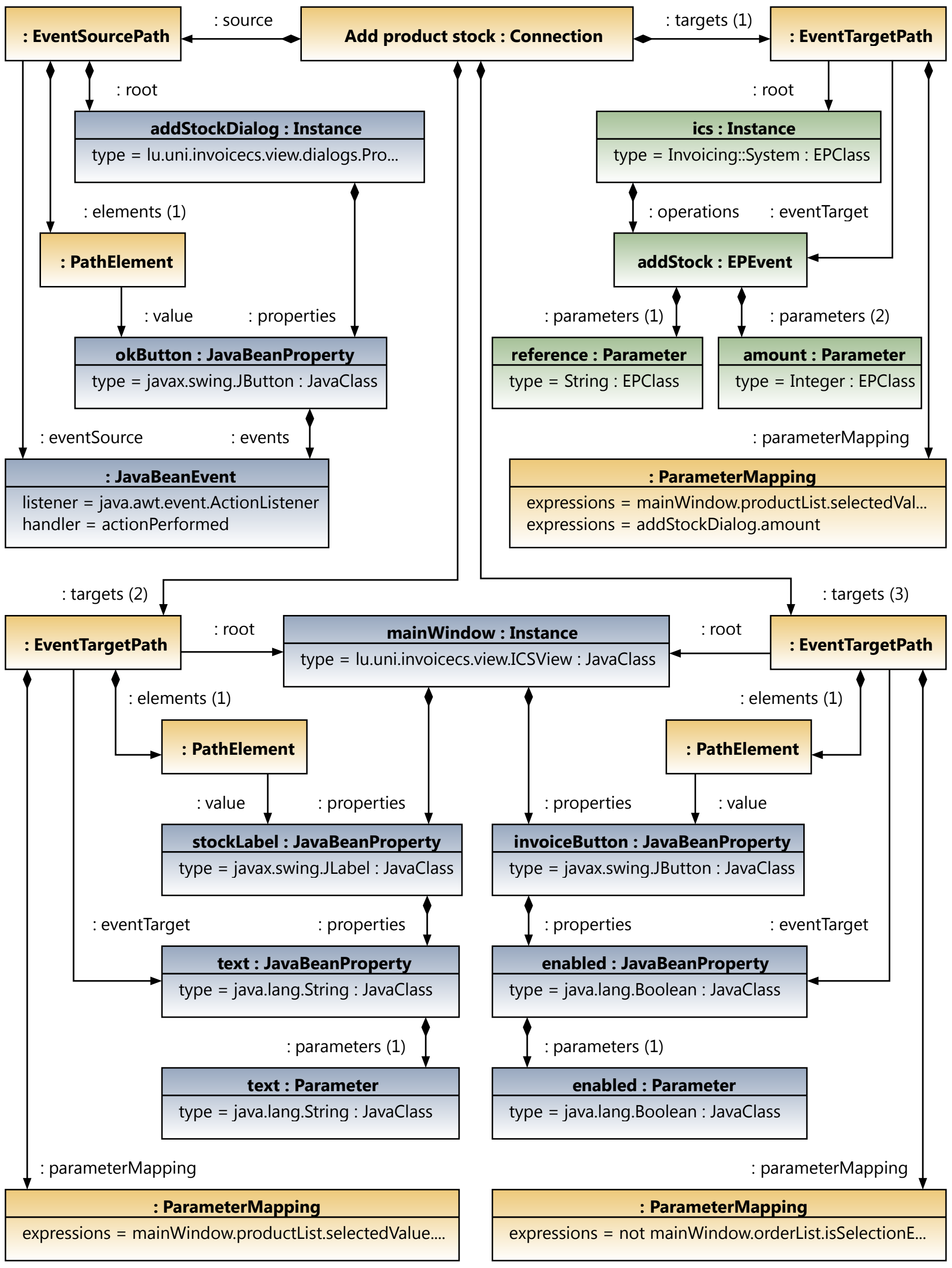

: EventSourcePath
: source
Add product stock : Connection
: targets (1)
: EventTargetPath
: root
addStockDialog : Instance
type = lu.uni.invoicecs.view.dialogs.Pro...
: root
ics : Instance
type = Invoicing::System : EPClass
: elements (1)
: PathElement
: operations
: eventTarget
addStock : EPEvent
: value
: properties
okButton : JavaBeanProperty
type = javax.swing.JButton : JavaClass
: parameters (1)
: parameters (2)
reference : Parameter
type = String : EPClass
amount : Parameter
type = Integer : EPClass
: eventSource
: events
: parameterMapping
: JavaBeanEvent
listener = java.awt.event.ActionListener
handler = actionPerformed
: ParameterMapping
expressions = mainWindow.productList.selectedVal...
expressions = addStockDialog.amount
: targets (2)
: targets (3)
: EventTargetPath
: root
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: root
: EventTargetPath
: elements (1)
: elements (1)
: PathElement
: PathElement
: value
: properties
: properties
: value
stockLabel : JavaBeanProperty
type = javax.swing.JLabel : JavaClass
invoiceButton : JavaBeanProperty
type = javax.swing.JButton : JavaClass
: eventTarget
: properties
: properties
: eventTarget
text : JavaBeanProperty
type = java.lang.String : JavaClass
enabled : JavaBeanProperty
type = java.lang.Boolean : JavaClass
: parameters (1)
: parameters (1)
text : Parameter
type = java.lang.String : JavaClass
enabled : Parameter
type = java.lang.Boolean : JavaClass
: parameterMapping
: parameterMapping
: ParameterMapping
expressions = mainWindow.productList.selectedValue....
: ParameterMapping
expressions = not mainWindow.orderList.isSelectionE...

Figure 6.15: Add product stock

Lastly, the invoice button enablement needs to be re-evaluated. It is possible that, before adding stock, an order has been selected, the quantity of which was higher than the available stock. After restocking the product, this may no longer be the case. The bottom right *EventTargetPath* fulfils this function by setting the invoice button's enabled property to the value of the following parameter mapping expression:

```
not mainWindow.orderList.isSelectionEmpty() and mainWindow.orderList.
    selectedValue.oclAsType(Invoicing::Order).isInvoicable
```

Listing 6.8: *Add product stock* connection parameter expressions

6.4.4 Order actions

An order, the main artefact of the case study, can be invoiced or cancelled. The first action is represented in the diagram 6.16 and the second one in 6.17 on the facing page. The two connections listen to the clicking of the *invoiceButton* or the *cancelButton* respectively and react accordingly by executing the *invoice* or *cancel* operations of the business domain.

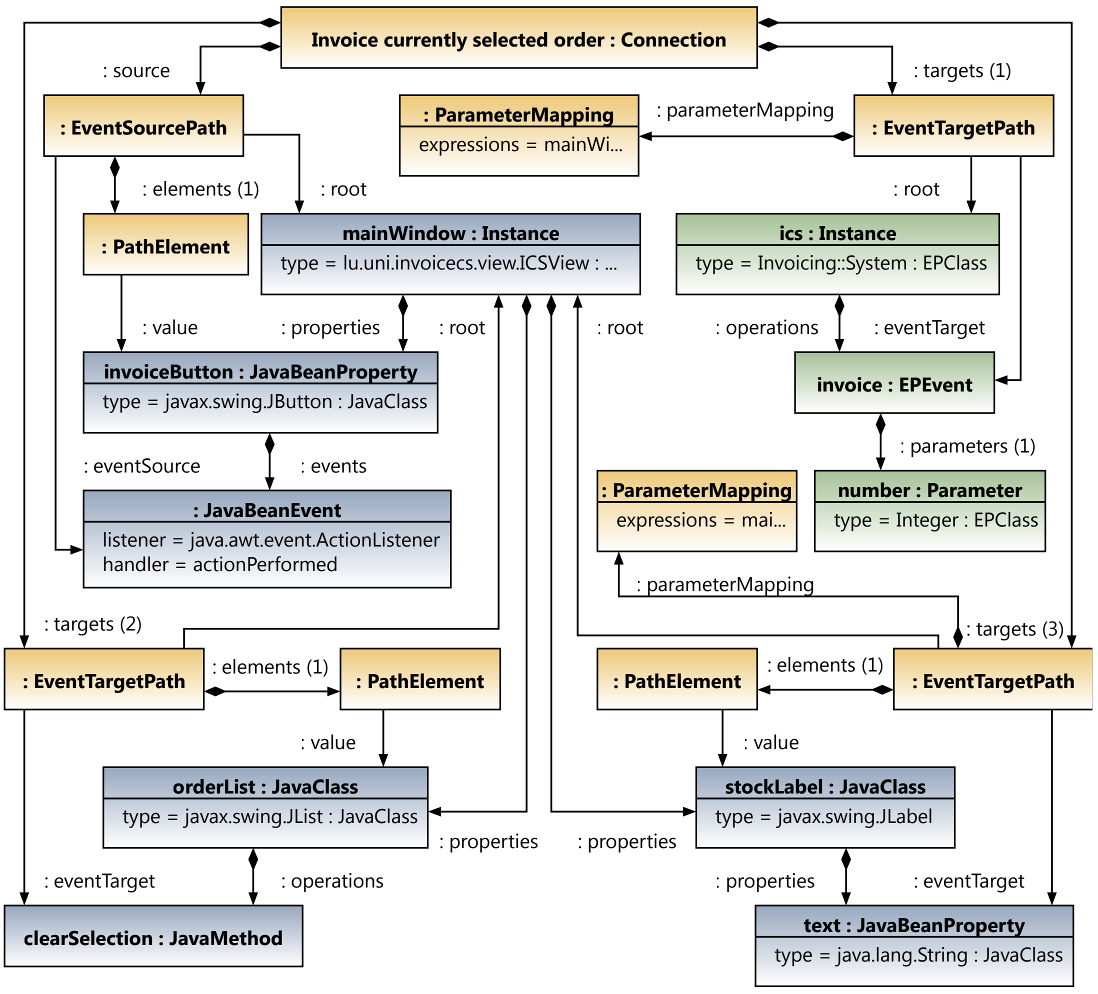


Figure 6.16: Invoice currently selected order

Both operations require as a single parameter the number of the order to handle, which they are supplied with using the following parameter mapping expression:

```
mainWindow.orderList.selectedValue.oclAsType(Invoicing::Order).number
```

Listing 6.9: Parameter expression for both invoicing and cancelling an order

Invoicing an order has an additional two consequences over cancelling an order. For one, it reduces the available stock of the ordered product, hence the *Invoice currently selected order* connection sets the text property of the *stockLabel* to the value of the parameter mapping expression 6.7 on page 48 using the *EventTargetPath* on the bottom right. Furthermore, the selection of the order list is cleared, as seen in the *EventTargetPath* on the bottom left, since no further action on the invoiced order is possible. Additionally, clearing the selection triggers a re-rendering of the previously selected item in the list, which properly updates the list content.

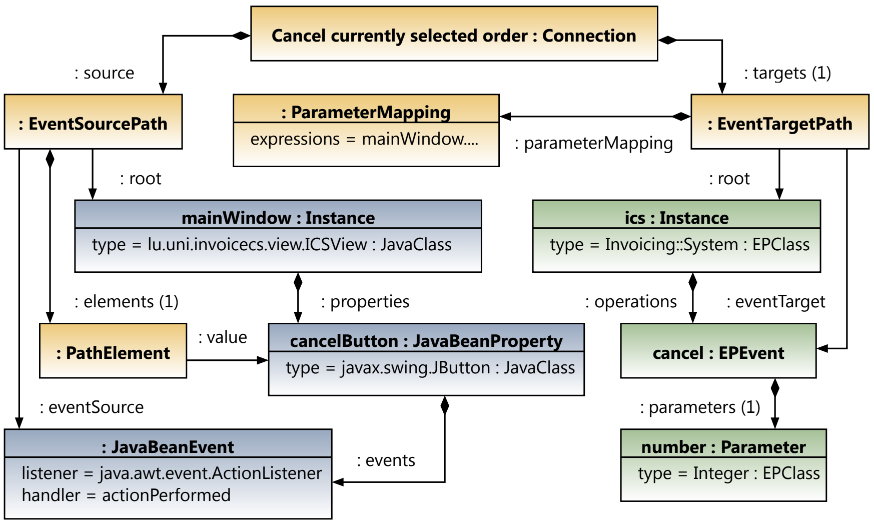


Figure 6.17: Cancel currently selected order

The task of the last connection, named *Monitor set of orders for changes*, is to update the list of orders whenever the content of the set of orders in the business domain changes, that is any time an order is created or cancelled.

The parameter mapping used to achieve this task is:

```
if mainWindow.productList.isSelectionEmpty() then
   ics.orders->asSequence()
else
   mainWindow.productList.selectedValue.oclAsType(Invoicing::Product).
       orders->asSequence()
endif
```

Listing 6.10: *Monitor set of orders for changes* connection parameter expression

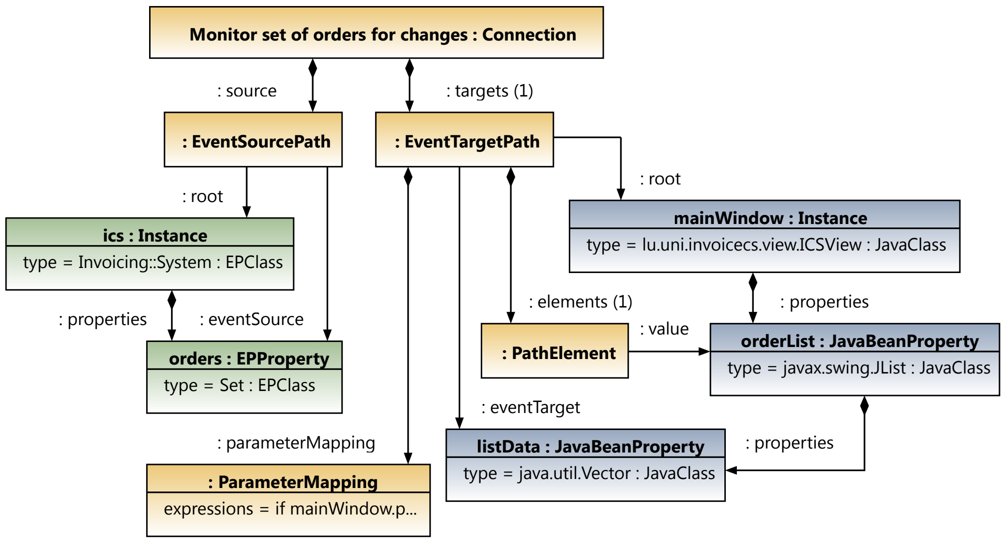

Monitor set of orders for changes : Connection
: source
: targets (1)
: EventSourcePath
: EventTargetPath
: root
: root
ics : Instance
type = Invoicing::System : EPClass
mainWindow : Instance
type = lu.uni.invoicecs.view.ICSView : JavaClass
: elements (1)
: properties
: properties
: eventSource
: value
: PathElement
orderList : JavaBeanProperty
type = javax.swing.JList : JavaClass
orders : EPProperty
type = Set : EPClass
: eventTarget
: parameterMapping
: properties
listData : JavaBeanProperty
type = java.util.Vector : JavaClass
: ParameterMapping
expressions = if mainWindow.p...

Figure 6.18: Monitor set of orders for changes

CHAPTER 7

Evaluation

In chapter 6 we created a mapping and two binding models with the intent of creating a bridge between an invoicing system created using the EP language and a GUI written in Java using the Swing library. In this chapter we evaluate the models using the code generator described in chapter 5.

In section 7.1 the two created DSMLs are validated by generating code using the case study models. Section 7.2 contains the definition of the scope of the DSMLs. In section 7.3 the advantages of using the DSMLs is discussed, while section 7.4 shows the issues and drawbacks a user or programmer faces currently when using the languages.

7.1 Validation

To validate the DSMLs, we generated code from their instances created for the case study, using our purpose build code generator. As expected, it generated a single file, containing an aspect and a class. The aspect monitors the instantiation of the *System* EP class in the invoicing business domain and in turn instantiates the class created by the code generator. The class contains in its constructor all the modelled connections, each in turn containing the event source and target paths as well as any parameter mapping expressions modelled.

Compiling the GUI's source code as well as the source code generated from the EP models, using the aspect file created by our generator, created a fully functional application. The GUI's lists are populated with the correct data, the different buttons work as intended and the GUI is updated appropriately after selecting an item from the two lists as well as after any of the actions triggered by pressing a button finishes. This means that the information contained by the different models, and more importantly the information we were able to include in them, was sufficient for the purposes of the case study.

Compared against a manual implementation of a bridge between the EP models and its GUI (c.f. chapter B on page 71), the file generated from the DSML instances is a lot larger - around three times as many lines of code - and thus also more difficult to read or maintain manually. On the other hand, this was to be expected. The generator written for the thesis is to be considered as being a proof of concept implementation. Efficiency and maintainability were not high priorities during its development. Furthermore, the size of the generated code does not constitute a disadvantage over a manual implementation. The generated bridge code represents the semantics of the models it is created from and not highly optimised code written specifically to interface this specific GUI with these specific EP models.

In the end, the result of the work conducted for the thesis is a means to create a bridge between an abstract modelling language and concrete code, resulting in a finished and executable application. It is composed of the two DSMLs *Binding (Core) DSML*, to bind or translate a targeted language, and *Mapping DSML*, to model the required connections between a source and multiple targets originating and ending in either language. Additionally, a proof-of-concept code generator was created to demonstrate and validate the two DSMLs.

7.2 Scope

The scope of the *Binding Core* and *Mapping DSMLs* includes but is not limited to creating a bridge between abstract models and a concrete GUI. The following list defines a set of rules to which a language must adhere in order to be supported:

Object Oriented
The two DSMLs have been designed to specifically support object oriented languages and their ability to represent the state as well as the behaviour of a system and generate notifications when the system state changes.

Both EP and Java, the languages used in our case study, are OO (modelling) languages. Examples of other supported languages are C# [88], VB.Net [87] and C++ [14].

Execution platform
The different languages bound by their *Binding DSML* extensions and used in a mapping model have to have a common execution platform. In other words, the programming languages used, either manually or as the language for generated source code, need to be compatible.

The Democles tool used to create the EP models in the case study, generates Java source code for the models and the GUI was written in Java Swing. Hence, the prototypical code generator for the DSMLs has been written to generate Java source code as well. Scala and Jython are examples of different programming languages which could also have been used. Both compile to run on the same Java Virtual Machine than Java itself, hence they have a common platform.

Instantiation
If elements of a language are to be monitored for instantiation, the language either needs to trigger a corresponding event, or the created or generated source code needs to be written in a language for which an AOP framework exists.

For example, see diagram 6.4 on page 40. The *Instance* named ics represents an EP class and is monitored for instantiation. EP may not trigger a corresponding event, but the Democles tool generates Java source code from the EP classes.

The AOP framework AspectJ can then be used to monitor the *Instance* and access its value.

Examples of AOP frameworks includes: PostSharp [68] and Snap [75] for the Microsoft .Net platform, which includes amongst others the languages C# and VB.Net, and AspectC++ [80] for the C++ language. A non-exhaustive list of other language with varying degree of support for AOP can be found here [85].

Furthermore, the number of different languages which can be used by a given *Mapping DSML* instance is unbounded. The diagrams shown in figures 6.6 on page 42, 6.7 on page 43 and 6.8 on page 44 are examples of connections where the event source and the event targets originate from the same targeted language, which in the case of these examples is Java. The EP language only shows up in the parameter mapping expressions. In fact, the DSMLs can be used with a single language, e.g. to connect events with event handlers in a GUI, or multiple languages, e.g. creating a connection between business logic models, a data model, a specialised GUI framework and a GUI.

7.2.1 Limitations

Some constructs of a targeted language may not be supported, even if the language fulfils the rules described above. This is due to the requirement that any communication between the language instances needs to be initiated by an event.

An example of a non-supported language construct is the `ListCellRenderer` used by Java to customise the design and layout of a list's content (c.f. sections 6.3.2 on page 38 and A.2.2 on page 65). In order to properly render the information contained in the *Product* and *Order* EP classes, a custom event based `EListCellRenderer` had to be implemented.

7.3 Advantages

Using the created *Mapping DSML* and *Binding (Core) DSMLs* has multiple advantages over a manual implementation. Mainly, the DSMLs allow to create declarative models instead of having to program imperatively. Secondly, the created models are reusable, which increases productivity over a manual implementation.

7.3.1 Declarative

A mapping model declaratively defines how a system should react to an event. This means that it tells a system what it should do when a specific event occurs but not how this reaction is achieved. Consequently, by using the *Mapping DSML* and given a set of *Binding DSML* models, a user does not need to be able to use the languages targeted by these models in order to create a connection between them. This becomes even more advantageous, when more than two systems need to be interconnected, each using a different modelling or programming language. The user only needs to know how to use two DSMLs, which are smaller and less expressive than the languages bound by their respective *Binding Core DSML* extensions.

7.3.2 Reusability

The reusability of the different DSMLs and their instances is coupled to the reusability of their targets. For instance, a *Binding Core DSML* extension only needs to be created once for the language it binds. Similarly, instances of these extensions need to be created once for every language instance, i.e. once per modelled system or application. Lastly, an instance of the *Mapping DSML* needs to be created for every targeted end-application, as would a programmer need to do for a manual implementation.

The difference in reusability between using the DSMLs and a manual implementation is the code generator. If a new execution platform is chosen or a different programming language should be used for the code generated from the models, only the code generator needs to be recreated and the models may be reused. Although the initial investment in resources is higher compared to a manually written bridge, a code generator only needs to be created once per target execution language.

7.4 Issues & Drawbacks

The case study uncovered some issues that remain to be solved and drawbacks a user will face with the DSMLs. These issues are largely due to the current lack of a proper concrete syntax and corresponding tool support. For example, the modelling environment for working with the DSMLs is not mature yet and does not allow to use the DSMLs efficiently. The Eclipse plugins generated from the Ecore based metamodels are very simple model editors based on the abstract language of the DSMLs. They are missing key features, such as proper instance validation, support for model driven testing or support for a concise concrete language, which would *"maximise the benefits of having models, and [...] minimise the effort required to maintain them"* [53]. Furthermore, an instance of one of the *Binding DSMLs* has to be created manually by someone familiar with the targeted language. Ideally, such an instance would be generated using a dedicated tool.

Furthermore, the *Binding Core DSML* binds a specific language by representing the relevant parts using the structure and semantics the DSML provides in a DSML extension. A *Path* in a *Mapping DSML* model defines how to navigate from a “root” instance to one of its “child” instances. This information is required by the code generator to create fully functional code. This also means that a user needs to be somewhat familiar with the systems modelled in instances of the used *Binding Core DSML* extensions. The issue may be circumvented with the proper tool support. For example, a Java Swing GUI may be represented graphically, as done by the WindowBuilder Eclipse plugin, and a selection of an element in this graphical view would then generate the *Path* required to reach it.

CHAPTER 8

Conclusion

This chapter concludes the thesis with a small summary in section 8.1 and a non-exhaustive list of issues which still need to be addressed and features which would simplify the usage of the created Domain Specific Modelling Languages (DSMLs) in section 8.2.

8.1 Summary

Interest in Model Driven Software Development (MDSD) has been rising over the last few years, and its peak is not reached yet. Using models to create applications allows to raise the level of abstraction away from implementation specific issues. The topic of this thesis is to create a bridge between abstract models and concrete code, an approach we find to have advantages over a purely model driven scenario. First of all, it allows to reuse already existing software units and secondly, platform specificities are often better expressed using platform specific notations and tools. The latter holds e.g. for Graphical User Interfaces (GUIs), for which no platform independent modelling language exists, which results in GUIs with a natural feel (although part of the GUI code can be modelled using tools such as WindowBuilder).

The bridging mechanism is realised using two DSMLs. The *Binding Core DSML* uses object oriented notations to describe a language bound by it structurally as well as idiomatically. We have shown two language specific extensions of the Core DSML, namely one for Java and one for the EP modelling language. The idiomatic description, which defines how language elements may be used, i.e. which elements are event sources or event targets, are needed by the second metamodel, the *Mapping DSML*. With it, connections can be created between event sources and targets. This is achieved by navigating from specified root instances to the elements which provide said event source or target.

Afterwards we created a code generator able to generate source code for the Java platform, using the connections defined in a mapping model in combination with the structural information of the bound languages in the binding core extensions. Using the code generator and a case study showing an Invoicing System, which we modelled using the EP modelling language and for which we created a GUI using Java Swing, we validated the DSMLs by generating Java based source code. The resulting code contained all the information necessary to compile a fully functional application.

8.2 Future Work

In its current state, our approach to creating a bridge between abstract models and concrete code still has some issues. We mentioned in section 7.4 on page 56 that the tool support is not mature yet and listed some of the features which would increase the usability of the DSMLs.

Furthermore, the abstract syntax and the semantics of the DSMLs are not yet formally defined. A formal definition would further increase the usefulness of our approach, as it allows to formally reason over the models, and do model based testing. Examples of how to achieve this task can be found in [12] - transformation between the models and well defined “semantic units” - and in [18]. The latter proposes to include the semantics in the domain model, by modelling the API of a well defined domain library. Applying [89] to our DSMLs, in combination with the previously mentioned semantic inclusion in the DSMLs, would allow to define the semantics of the DSMLs using notations specific to the platform targeted by the code generator. We note that the modelling language EP has a formal semantics [51] and thus could be integrated in such a formal approach.

This would also resolve the main remaining issue with the code generator, namely that it was purpose-built to compile the DSML instances created for the case study. In other words, the templates introduced in chapter 5 on page 27 are at this stage incomplete and the generator only supports Java and EP.

Lastly, a more thorough validation should be done on the DSMLs using additional case studies. To validate the theoretical claims in section 7.2 on page 54 the studies should be implemented using different combinations of multiple modelling and programming languages.

APPENDIX A

Case study implementation details

In this chapter we will describe more of the implementation details of the case study application. We will begin with the events and query properties of the different EP models, of which we will explain the OCL code snippets used. Then we will outline more of the code of the GUI, including manually added parts.

A.1 EP dynamic behaviour

Since the static structure of the EP models has already been outlined (c.f. section 6.3.1 on page 35), we will limit the details to their dynamic behaviour.

Event: *invoice*

Invoicing an order means to change the state of the order to *invoiced* and to reduce the stock of the ordered product by the amount specified in the order.

The *invoice* event has only a single parameter named *number* which is the number of the order to invoice. In order to access the corresponding order instance, a query property is needed, which returns the order with the specified number.

```
orders->any(o : Order | o.number = number)
```

Listing A.1: The query property getOrder's OCL code

Furthermore, the order only stores the reference of the ordered product, hence another query property is needed returning the product with the specified reference.

```
products->any(p : Product | p.reference = reference)
```

Listing A.2: The query property getProduct's OCL code

Additionally, since invoicing an order means to change the state of an order and the stock of a product, corresponding "setter" events are needed in the EP models *Product* and *Order* (c.f. figure 6.1 on page 36).

The *invoice* event has a guard associated with it, which ensures that the provided order number exists (line 1) and that the given order is actually invoicable (line 2).

```
not getOrder(number).oclIsUndefined()
and getOrder(number).isInvoiceable
```

Listing A.3: *invoice* event guard

isInvoiceable is a query property of the *Order* model, which returns the boolean value 'true' if the given order instance is in fact invoice-able, i.e. its state is *pending* and the ordered product's stock is high enough:

```
state = 'pending' and system.getProduct(product).stock >= quantity
```

Listing A.4: *isInvoiceable* query property

If the guard holds, i.e. it returns true, the event is forwarded to its child links *setState* and *setStock* (c.f. figure 6.2b on page 36). The latter two are called for every item in the collections orders and products respectively. In fact you iterate over every item one by one and execute the event. In order to ensure that the child-events are only triggered for the order with the specified number and for the product referenced by the order, the child-link-guards in listings A.5 and A.6 are used.

```
self.number = number
```

Listing A.5: *invoice* orders guard

```
self.reference = getOrder(number).product
```

Listing A.6: *invoice* product guard

self in these guards refers to the current item of the iteration. It should also be noted that OCL code snippets in an EP model are link local. This means they can only access and/or modify properties or forward the event to other events linked with or local to the executed event. The call to the query property *getOrder* in listing A.6 is not local to the event, since the event in question is the *setStock* event of the product EP model and getProduct is a query property of the EP model *System*. In order to use it in the guard, the property needs to be linked to the calling *invoice* event, which is why a *feeds* link has been attached between both (c.f. dotted line in figure 6.1a on page 36).

The parameters for the child-events are simply the string 'invoiced' for *setState* and the calculated remaining stock for *setStock*.

Event: *addStock*

The *addStock* event works similarly to the *invoice* event. It has two parameters *reference* and *amount* specifying the reference of the product to which an amount of stock is added. The event's guard checks whether the given reference is valid and that the specified amount is a number bigger than zero. In that case the event is forwarded into the *products* collection with a child-link-guard ensuring that the stock is updated only for the product with the given reference (c.f. figure 6.2a on page 36).

Event: *order*

To add a new order, the *order* event is triggered. It has two parameters: the reference and quantity of the product that is to be ordered. A guard on the event checks whether the provided product reference is valid and if the ordered quantity is a value bigger than 0:

```
not getProduct(reference).oclIsUndefined()
and quantity > 0
```

Listing A.7: *order* event guard

Two properties of the order that is to be created are still missing: the order number and order state. The latter is automatically set to 'pending', but we need a means to create a unique number for the former. Since the case study does not mention any requirements for how an order should be identified, we chose to simply set the new order's number to $max(number) + 1$. Aside from the fact that a more real world numbering system would be quite difficult to implement in EP, this ensures that any newly created order has a unique number associated to it.

The problem we are faced with now, is that EP does not provide a direct method for determining the maximum value of a property of all the items in a collection. The trick we used is to sort the collection in question on the property in question and retrieve the last item. Adding 1 to that item's number returns the wanted result. The following listing shows the OCL code of the *getNewOrderNumber* query property used to that extent:

```
if orders->isEmpty() then
   1
else
   orders->sortedBy(number)->last().number + 1
endif
```

Listing A.8: The query property *getNewOrderNumber*'s OCL code

Although this method is "good enough" for the purposes of this case study, nonetheless it should be noted that this particular implementation has multiple issues:

1. Depending on the size of the collection of orders, the performance of the *getNewOrderNumber* query property would be very poor.

2. The number of orders the modelled system is able to manage varies and is limited. In an extreme situation, it can only handle a single order, if its number is equal to the maximum value an Integer can take.

If the *order* event's guard holds, then a new order instance is created with the collected property values and added to the set of orders. The latter is achieved by using an impact link between the event and the collection (c.f. figure 6.1a on page 36). Such an impact link is annotated with OCL code and assigns the value calculated from that code to the target of the link.

The OCL code of the *order* event can be seen in listing A.9 on the next page.

```
orders->including(Order::create(Tuple {
   number = getNewOrderNumber,
   product = reference,
   quantity = quantity,
   state = 'pending'
}))
```

Listing A.9: The *order* event impact link OCL code

Event: *cancel*

The *cancel* event behaves similarly to the *order* event in that it directly impacts the collection of orders. It has as the single parameter the number of the order to cancel and it checks whether the given order number is valid and that the corresponding order is in fact cancelable (c.f. listing A.10).

```
not getOrder(number).oclIsUndefined()
and getOrder(number).isCancelable
```

Listing A.10: *cancel* event guard

isCancelable is a query property of the *Order* model, which returns the boolean value 'true' if the given order instance is in fact cancelable, i.e. its state is *pending*:

```
state = 'pending'
```

Listing A.11: *isCancelable* query property

If the guard holds, the value calculated from listing A.12 is assigned to the *orders* property.

```
orders->excluding(getOrder(number))
```

Listing A.12: The *cancel* event impact link OCL code

Product query property: *orders*

The *Product* model contains a query property named *orders*, which simply returns a set containing those orders which reference the given product:

```
system.orders->select(o : Order | o.product = reference)
```

Listing A.13: The *orders* query property of the *Product* model

A.2 GUI implementation

As stated in the introduction, the GUI was created using the WindowBuilder plugin for Eclipse for the Swing target platform. Figure 6.3 on page 37 shows the main window of the application, with a selected product on the left, product details on the top and orders of the selected product on the right.

A.2.1 Custom panels

pnlProductList, pnlEmpty, pnlProductDetails and pnlOrderList are custom JPanels which were also designed using WindowBuilder. pnlEmpty only contains a single JLabel with a text asking the user to select a product to see its details. The pnlProductDetails JPanel contains a collection of labels used to show detailed information about the selected product. In addition, it contains two buttons to create a new order or add stock to the selected product respectively. Listing A.14 shows the code generated by WindowBuilder for pnlProductDetails.

```
public class ProductDetailsPanel extends JPanel {
   private static final long serialVersionUID = 1L;

   private JLabel lblProductName = null;
   private JLabel lblReference = null;
   private JLabel lblStock = null;
   private final JButton btnAddStock;
   private final JButton btnOrder;

   public ProductDetailsPanel() {
      this.setPreferredSize(new Dimension(400, 62));
      final GridBagLayout gridBagLayout = new GridBagLayout();
      gridBagLayout.columnWidths = new int[] { 80, 80, 50, 109, 81 };
      gridBagLayout.rowHeights = new int[] { 42, 20 };
      gridBagLayout.columnWeights = new double[] { 0.0, 0.0, 0.0, 1.0,
          0.0 };
      gridBagLayout.rowWeights = new double[] { 1.0, Double.MIN_VALUE
          };
      this.setLayout(gridBagLayout);

      this.lblProductName = new JLabel("");
      this.lblProductName.setFont(new Font("Arial", Font.BOLD, 17));
      final GridBagConstraints gbc_lblProductName = new
          GridBagConstraints();
      gbc_lblProductName.fill = GridBagConstraints.HORIZONTAL;
      gbc_lblProductName.insets = new Insets(5, 5, 5, 5);
      gbc_lblProductName.gridwidth = 4;
      gbc_lblProductName.gridx = 0;
      gbc_lblProductName.gridy = 0;
      this.add(this.lblProductName, gbc_lblProductName);

      final JLabel ReferenceLabel = new JLabel("Reference:");
      ReferenceLabel.setFont(new Font("Arial", Font.PLAIN, 12));
      final GridBagConstraints gbc_ReferenceLabel = new
          GridBagConstraints();
      gbc_ReferenceLabel.insets = new Insets(0, 5, 0, 5);
      gbc_ReferenceLabel.gridx = 0;
      gbc_ReferenceLabel.gridy = 1;
      this.add(ReferenceLabel, gbc_ReferenceLabel);

      this.lblReference = new JLabel("");
      this.lblReference.setFont(new Font("Arial", Font.BOLD, 12));
      ReferenceLabel.setLabelFor(this.lblReference);
      final GridBagConstraints gbc_lblReference = new
```

```
            GridBagConstraints();
        gbc_lblReference.insets = new Insets(0, 0, 0, 15);
        gbc_lblReference.gridx = 1;
        gbc_lblReference.gridy = 1;
        this.add(this.lblReference, gbc_lblReference);

        final JLabel StockLabel = new JLabel("Stock:");
        StockLabel.setPreferredSize(new Dimension(50, 12));
        StockLabel.setFont(new Font("Arial", Font.PLAIN, 12));
        final GridBagConstraints gbc_StockLabel = new GridBagConstraints
            ();
        gbc_StockLabel.insets = new Insets(0, 0, 0, 5);
        gbc_StockLabel.gridx = 2;
        gbc_StockLabel.gridy = 1;
        this.add(StockLabel, gbc_StockLabel);

        this.lblStock = new JLabel("");
        StockLabel.setLabelFor(this.lblStock);
        this.lblStock.setFont(new Font("Arial", Font.BOLD, 12));
        final GridBagConstraints gbc_lblStock = new GridBagConstraints();
        gbc_lblStock.anchor = GridBagConstraints.WEST;
        gbc_lblStock.insets = new Insets(0, 0, 0, 5);
        gbc_lblStock.gridx = 3;
        gbc_lblStock.gridy = 1;
        this.add(this.lblStock, gbc_lblStock);

        final JPanel buttons = new JPanel();
        final GridBagConstraints gbc_buttons = new GridBagConstraints();
        gbc_buttons.gridheight = 2;
        gbc_buttons.insets = new Insets(5, 0, 5, 0);
        gbc_buttons.fill = GridBagConstraints.HORIZONTAL;
        gbc_buttons.gridx = 4;
        gbc_buttons.gridy = 0;
        this.add(buttons, gbc_buttons);
        buttons.setLayout(new BoxLayout(buttons, BoxLayout.Y_AXIS));

        this.btnAddStock = new JButton("Add Stock");
        buttons.add(this.btnAddStock);

        final Component verticalStrut = Box.createVerticalStrut(5);
        buttons.add(verticalStrut);

        this.btnOrder = new JButton("Order");
        this.btnOrder.setMaximumSize(new Dimension(81, 23));
        buttons.add(this.btnOrder);
    }

    public JButton getAddStockButton() {
        return this.btnAddStock;
    }

    public JButton getOrderButton() {
        return this.btnOrder;
    }
```

```
   public JLabel getProductNameLabel() {
      return lblProductName;
   }

   public JLabel getReferenceLabel() {
      return lblReference;
   }

   public JLabel getStockLabel() {
      return lblStock;
   }
}
```

Listing A.14: Full generated code for the `pnlProductDetails` panel

Of note is also the panel named `productDetailsPanel` as it uses `CardLayout` to stack the `pnlEmpty` and `pnlProductDetails` on top of each other. The appropriate panel is shown depending on whether a product is selected or not, as determined by the method in listing A.15.

```
public void displayProductDetails() {
   final CardLayout cl = (CardLayout)this.productDetailsPanel.getLayout
       ();

   if(this.getProductList().isSelectionEmpty())
      cl.show(this.productDetailsPanel, "Empty");
   else
      cl.show(this.productDetailsPanel, "Product Details");
}
```

Listing A.15: Method to show or hide the product details panel

A.2.2 EListCellRenderer

As explained in section 6.3.2 on page 38, we created a custom event based list cell renderer named `EListCellRenderer`. This renderer represents the event source and its code is shown in listing A.16.

```
public class EListCellRenderer implements ListCellRenderer {
   protected EventListenerList listenerList = new EventListenerList();

   protected Component component = null;

   public EListCellRenderer(final Component component) {
      this.component = component;
   }

   @Override
   public Component getListCellRendererComponent(final JList list,
         final Object value, final int index, final boolean isSelected,
         final boolean cellHasFocus) {
      final EListCellRenderEvent ev = new EListCellRenderEvent(
            this.component, list, value, index, isSelected,
               cellHasFocus);

      for (final EListCellRenderListener listener : this.listenerList
            .getListeners(EListCellRenderListener.class)) {
         listener.eListCellRendered(ev);
      }

      return this.component;
   }

   public Component getComponent()
   {
      return this.component;
   }

   public void addEListCellRenderListener(
         final EListCellRenderListener listener) {
      this.listenerList.add(EListCellRenderListener.class, listener);
   }

   public void removeEListCellRenderListener(
         final EListCellRenderListener listener) {
      this.listenerList.remove(EListCellRenderListener.class, listener)
         ;
   }
}
```

Listing A.16: Custom list cell renderer `EListCellRenderer`

The event object passed to the cell render listeners contains the information passed to the `getListCellRendererComponent` `ListCellRenderer` method as well as a reference to the custom component used for rendering, as shown in listing A.17.

```
public class EListCellRenderEvent extends EventObject {
   private static final long serialVersionUID = -3339979789572534980L;

   protected Component component;

   protected Object value;
   protected int index;
   protected boolean isSelected;
   protected boolean cellHasFocus;

   public EListCellRenderEvent(final Component component,
         final Object source, final Object value, final int index,
         final boolean isSelected, final boolean cellHasFocus) {
      super(source);

      this.component = component;
      this.value = value;
      this.index = index;
      this.isSelected = isSelected;
      this.cellHasFocus = cellHasFocus;
   }

   public Component getComponent() {
      return this.component;
   }

   public Object getValue() {
      return this.value;
   }

   public int getIndex() {
      return this.index;
   }

   public boolean isSelected() {
      return this.isSelected;
   }

   public boolean isCellHasFocus() {
      return this.cellHasFocus;
   }
}
```

Listing A.17: List cell render event `EListCellRenderEvent`

To be able to register itself with a `EListCellRenderer`, an event listener needs to implement the interface in listing A.18.

```
public interface EListCellRenderListener extends EventListener {
   public void eListCellRendered(EListCellRenderEvent renderEvent);
}
```

Listing A.18: List cell render event listener interface `EListCellRenderListener`

As an example for a custom component, the following listing shows the code for the panel named `OrderCell`, as generated by WindowBuilder. In the case study GUI, its `render` method is executed for every `EListCellRenderEvent`.

```
public class OrderCell extends JPanel {
   private static final long serialVersionUID = 1L;

   private JLabel lblQuantity = null;
   private JLabel lblProductName = null;
   private JLabel lblState = null;

   public OrderCell() {
      this.setPreferredSize(new Dimension(300, 37));
      final GridBagLayout gridBagLayout = new GridBagLayout();
      gridBagLayout.columnWidths = new int[] { 42, 0, 0 };
      gridBagLayout.rowHeights = new int[] { 0 };
      gridBagLayout.columnWeights = new double[] { 0.0, 0.0, Double.
         MIN_VALUE };
      gridBagLayout.rowWeights = new double[] { Double.MIN_VALUE };
      this.setLayout(gridBagLayout);
```

```
        this.lblQuantity = new JLabel("");
        this.lblQuantity.setHorizontalAlignment(SwingConstants.RIGHT);
        this.lblQuantity.setHorizontalTextPosition(SwingConstants.RIGHT);
        final GridBagConstraints gbc_lblQuantity = new GridBagConstraints
            ();
        gbc_lblQuantity.fill = GridBagConstraints.HORIZONTAL;
        gbc_lblQuantity.insets = new Insets(0, 10, 5, 5);
        gbc_lblQuantity.gridx = 0;
        gbc_lblQuantity.gridy = 0;
        this.add(this.lblQuantity, gbc_lblQuantity);
        this.lblQuantity.setFont(new Font("Dialog", Font.BOLD, 14));

        this.lblProductName = new JLabel("");
        final GridBagConstraints gbc_lblProductName = new
            GridBagConstraints();
        gbc_lblProductName.insets = new Insets(0, 0, 5, 5);
        gbc_lblProductName.gridx = 1;
        gbc_lblProductName.gridy = 0;
        this.add(this.lblProductName, gbc_lblProductName);
        this.lblProductName.setFont(new Font("Dialog", Font.PLAIN, 14));

        this.lblState = new JLabel("(Pending)");
        final GridBagConstraints gbc_lblState = new GridBagConstraints();
        gbc_lblState.anchor = GridBagConstraints.WEST;
        gbc_lblState.insets = new Insets(0, 0, 5, 0);
        gbc_lblState.gridx = 2;
        gbc_lblState.gridy = 0;
        this.add(this.lblState, gbc_lblState);
        this.lblState.setFont(new Font("Dialog", Font.ITALIC, 14));
    }

    public void setQuantity(final int quantity) {
        this.lblQuantity.setText(quantity + "x");
    }

    public void setProductName(final String name) {
        this.lblProductName.setText(name);
    }

    public void setState(final boolean value) {
        this.lblState.setVisible(value);
    }

    public void setLabelColor(final Color color) {
        this.lblQuantity.setForeground(color);
        this.lblProductName.setForeground(color);
        this.lblState.setForeground(color);
    }

    public void render(Object source, String productName, int quantity,
        String state, boolean isSelected) {
        final JList sourceList = (JList) source;
        this.setProductName(productName);
        this.setQuantity(quantity);
        this.setState(state.equals("pending"));
        this.setBackground(isSelected ? sourceList
              .getSelectionBackground() : sourceList
              .getBackground());

        final Color textColor = state.equals(
              "pending") ? Color.BLACK : Color.LIGHT_GRAY;
        this.setLabelColor(textColor);
    }
}
```

Listing A.19: *OrderCell* custom list cell component

APPENDIX B

Manual Implementation

This chapter introduces a manual implementation of the bridge between the invoicing system modelled using EP and the GUI created using Java Swing. This implementation is composed of two parts: first Java Bean Adapters which are introduced in section B.1 and secondly the bridge code itself which is described in section B.2.

B.1 Java Bean Adapters

The source code generated by the Democles tool, does not create getters and setters for the properties in an EP class, and events are not converted to Java methods. Furthermore, EP uses different types for collections than Java.

The role of a Java Bean Adapter (JBA) is to provide access to an EP class following the Java Bean specification as well as converting types specific to EP to their respective Java types. Furthermore, a JBA is responsible for converting an EP event into a typical Java event. For example, the EP *property value changed* event in turn triggers a *property changed* event using the Java *PropertyChangeSupport* mechanism.

The latter, as well as basic instantiation of an EP class, is combined into the abstract JBA named `Base`, as seen in listing B.1. A JBA representing a specific EP class needs to extend this abstract class.

```
public abstract class Base implements IPropertyListener {
   protected Instance instance;

   protected PropertyChangeSupport pcs = new PropertyChangeSupport(this
       );

   public Base(final Instance instance) {
      this.instance = instance;
      this.instance.initProps();
   }

   @Override
   public void propertyValueChanged(final Property property) {
      this.pcs.firePropertyChange(property.entityName, new Object(),
            property.oldVal);
   }

   public void addPropertyChangeListener(final String propertyName,
         final PropertyChangeListener listener) {
      this.pcs.addPropertyChangeListener(propertyName, listener);
   }
}
```

Listing B.1: Excerpt of the abstract Base JBA

Listing B.2 shows the JBA for the Product EP class. The property named `stock` is monitored for changes and getters for the product's `reference`, `name`, `stock` and set of `orders` are included in this JBA. If the stock property changes, the method `propertyValueChanged` inherited from the Base JBA is executed and in turn triggers a property changed event using the `PropertyChangeSupport` class.

```
public class Product extends Base {
   public Product(final generated.Invoicing.Product product) {
      super(product);

      this.instance.addPropertyListener("stock", this);
   }

   public String getReference() {
      return (String) this.instance.getPropertyValue("reference");
   }

   public String getName() {
      return (String) this.instance.getPropertyValue("name");
   }

   public Integer getStock() {
      return (Integer) this.instance.getPropertyValue("stock");
   }

   public Vector<Order> getOrders() {
      final Vector<Order> orders = new Vector<Order>();

      final Iterator it = ((OCLSet) this.instance.getProperty("orders")
            .evalInContainer()).getValues().iterator();

      while (it.hasNext()) {
         orders.add(new Order((generated.Invoicing.Order) it.next()));
      }

      return orders;
   }
}
```

Listing B.2: Product JBA

B.2 Manual Bridge

The manually written bridge code, shown in listing B.3 on the following page, is divided up into the following components: the constructor in lines 8-108 creates the connections modelled in the case study. Lines 110-218 represent event handlers for the different events which may occur and lines 220-227 show a property change listener implemented as an inner class.

The connections are set up as follows:

Lines 9-13 instantiate the invoicing system and the GUI.

Lines 15-34 create a connection between list selection events and their corresponding event handlers. A list selection event is monitored by creating and adding a `ListSelectionListener`. When a selection event occurs, the `valueChanged` method is executed, which in turn executes the respective methods shown.

Lines 36-65 as described in section 6.3.2 on page 38 and A.2.2 on page 65, handle the render events of the `EListCellRenderer`.

Lines 67-69 populate the two lists with data from the invoicing business domain.

Lines 71-78 create a listener to monitor the *orders* property of the invoicing model named *System*. The listener's event handler is the same than the one for selecting an item from the list of products.

Lines 80-108 create action listeners and attach them to the four buttons used in the GUI. The action listeners react to clicking the button they are attached to. They each in turn execute the methods shown in their respective `actionPerformed` event handlers.

The event handlers for pressing a button are defined as follows:

Lines 110-135 show the `order` event handler. The handler asks the user for the number of product items to order, validates the user input and afterwards delegates the actual ordering to the business domain JBA named *System*.

Lines 141-170 show the `addStock` event handler. The handler works similarly to the `order` event handler.

Lines 172-176 show how an order is invoiced. The `number` property of the currently selected order is passed to the `invoice` method in the *System* JBA. Afterwards, the selection in the list of orders is cleared, which updates the view to reflect the change in state of the invoiced order. The reduction in stock of the order property is monitored, hence no action is necessary to update the corresponding label in the product details.

Lines 178-182 show how an order is cancelled. The handler works similarly to the `invoice` event handler.

A selection event of one of the two lists is handled by the following two methods:

Lines 184-210 handle the selection of an item in the list of products. After clearing any selection in the list of orders, the method stops monitoring the *stock* property of the previously selected product. The `displayProductDetails` method is called next, which shows or hides the product details panel depending on whether a product is selected or not.

If a new item was selected, its *stock* property is now monitored for changes using the class shown in lines 220-226. Afterwards, the `text` properties of the product details label are set appropriately. Lastly, the list of orders is populated with the entire list of orders or the orders associated to the selected product, depending on whether a product is selected or not.

Lines 212-218 handle the selection of an item in the list of orders. In fact, the only actions needed are setting the `enabled` property of the invoice and cancel buttons to their appropriate values.

The class `ProductStockChangeListener` in **lines 220-226** is responsible for reacting to a property change event. It is used by the `selectedProduct` event handler method, to monitor the *stock* property of the currently selected product. If a property change event occurs, the stock label is updated accordingly. The execution of the `selectedOrder` event handler method ensures that the invoice button is enabled appropriately, e.g. if after adding stock, the stock is high enough to invoice the selected order when it was not previously, then the invoice button will be enabled.

```
1  public class ICSPresenter {
2     private ICSView view = null;
3     private System model = null;
4
5     private Product currentProduct = null;
6     private final ProductStockChangeListener productStockChangeListener = new
          ProductStockChangeListener();
7
8     public ICSPresenter() {
9        // create domain model
10       this.model = System.getInstance();
11
12       // create view
13       this.view = new ICSView();
14
15       // list selection listeners
16       this.view.getProductList().getSelectionModel()
17             .addListSelectionListener(new ListSelectionListener() {
18                @Override
19                public void valueChanged(final ListSelectionEvent e) {
20                   if (!e.getValueIsAdjusting()) {
21                      ICSPresenter.this.selectedProduct();
22                   }
23                }
24             });
25
26       this.view.getOrderList().getSelectionModel()
27             .addListSelectionListener(new ListSelectionListener() {
28                @Override
```

```
                public void valueChanged(final ListSelectionEvent e) {
                    if (!e.getValueIsAdjusting()) {
                        ICSPresenter.this.selectedOrder();
                    }
                }
            });

        // list cell renderer content
        ((EListCellRenderer)this.view.getProductList().getCellRenderer())
            .addEListCellRenderListener(new EListCellRenderListener() {
                @Override
                public void eListCellRendered(
                        final EListCellRenderEvent renderEvent) {
                    final ProductCell pc = (ProductCell) renderEvent
                        .getComponent();

                    pc.render(renderEvent.getSource(), ((Product) renderEvent.
                         getValue())
                        .getName(), renderEvent.isSelected());
                }
            });

        ((EListCellRenderer)this.view.getOrderList().getCellRenderer())
            .addEListCellRenderListener(new EListCellRenderListener() {
                @Override
                public void eListCellRendered(
                        final EListCellRenderEvent renderEvent) {
                    final OrderCell oc = (OrderCell) renderEvent
                        .getComponent();

                    final Order order = (Order) renderEvent.getValue();

                    oc.render(renderEvent.getSource(),
                        System.getInstance().getProduct(order.getProduct()).
                            getName(),
                        order.getQuantity(), order.getState(),
                        renderEvent.isSelected());
                }
            });

        // set initial list data
        this.view.getProductList().setListData(this.model.getProducts());
        this.view.getOrderList().setListData(this.model.getOrders());

        // listen to changes to the list of orders
        this.model.addPropertyChangeListener("orders",
            new PropertyChangeListener() {
                @Override
                public void propertyChange(final PropertyChangeEvent e) {
                    ICSPresenter.this.selectedProduct();
                }
            });

        // add button actions
        this.view.getOrderButton().addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(final ActionEvent e) {
                ICSPresenter.this.order();
            }
```

```
    });

    this.view.getAddStockButton().addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        ICSPresenter.this.addStock();
      }
    });

    this.view.getInvoiceButton().addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        ICSPresenter.this.invoice();
      }
    });

    this.view.getCancelButton().addActionListener(new ActionListener() {
      @Override
      public void actionPerformed(final ActionEvent e) {
        ICSPresenter.this.cancel();
      }
    });
  }

  private void order() {
    int quantity;

    while (true) {
      final String q = JOptionPane.showInputDialog(null,
            "Enter the Quantity to order:", "Create a new Order",
            JOptionPane.QUESTION_MESSAGE);

      if (q == null) {
        return;
      }

      try {
        quantity = Integer.parseInt(q);

        if (quantity < 1) {
          throw new NumberFormatException();
        }

        break;
      } catch (final NumberFormatException e) {
        JOptionPane.showMessageDialog(null,
              "Error: please enter a number bigger than 0!",
              "Error adding to Stock", JOptionPane.ERROR_MESSAGE);
      }
    }

    this.model.order(((Product) this.view.getProductList()
        .getSelectedValue()).getReference(), quantity);
  }

  private void addStock() {
    int quantity;

    while (true) {
            final String q = JOptionPane.showInputDialog(null,
                  "Enter the Quantity to add to the Stock:", "Add to Stock",
                  JOptionPane.QUESTION_MESSAGE);

            if (q == null) {
               return;
            }

            try {
               quantity = Integer.parseInt(q);

               if (quantity < 1) {
                  throw new NumberFormatException();
               }

               break;
            } catch (final NumberFormatException ex) {
               JOptionPane.showMessageDialog(null,
                     "Error: please enter a number bigger than 0!",
                     "Error adding to Stock", JOptionPane.ERROR_MESSAGE);
            }
         }

         this.model.addStock(((Product) this.view.getProductList()
               .getSelectedValue()).getReference(), quantity);
      }

      private void invoice() {
         this.model.invoice(((Order) this.view.getOrderList()
               .getSelectedValue()).getNumber());
         this.view.getOrderList().clearSelection();
      }

      private void cancel() {
         this.model.cancel(((Order) this.view.getOrderList()
               .getSelectedValue()).getNumber());
         this.view.getOrderList().clearSelection();
      }

      private void selectedProduct() {
         this.view.getOrderList().clearSelection();

         if (this.currentProduct != null) {
            this.currentProduct.removePropertyChangeListener("stock",
                  this.productStockChangeListener);
         }

         this.currentProduct = (Product) this.view.getProductList()
               .getSelectedValue();

         this.view.displayProductDetails();

         if (this.currentProduct == null) {
            this.view.getOrderList().setListData(this.model.getOrders());
         } else {
            this.currentProduct.addPropertyChangeListener("stock",
                  this.productStockChangeListener);

            this.view.getProductNameLabel().setText(this.currentProduct.getName())
            ;
        this.view.getReferenceLabel().setText(this.currentProduct.getReference
            ());
        this.view.getStockLabel().setText(Integer.toString(this.currentProduct
            .getStock()));

        this.view.getOrderList().setListData(
              this.currentProduct.getOrders());
      }
   }

   private void selectedOrder() {
      final Order order = (Order) this.view.getOrderList()
            .getSelectedValue();

      this.view.getInvoiceButton().setEnabled(order != null && order.
          isInvoicable());
      this.view.getCancelButton().setEnabled(order != null && order.
          isCancelable());
   }

   class ProductStockChangeListener implements PropertyChangeListener {
      @Override
      public void propertyChange(final PropertyChangeEvent evt) {
         ICSPresenter.this.view.getStockLabel().setText(Integer.toString((
             Integer) evt.getNewValue()));
         ICSPresenter.this.selectedOrder();
      }
   }
}
```

Listing B.3: Source code of the manually implemented Bridge

Bibliography

[1] Gustav Aagesen and John Krogstie. “Analysis and Design of Business Processes Using BPMN”. In: *Handbook on Business Process Management 1*. Ed. by Jan vom Brocke and Michael Rosemann. International Handbooks on Information Systems. Springer Berlin Heidelberg, 2010, pp. 213–235. ISBN: 978-3-642-00416-2. DOI: 10.1007/978-3-642-00416-2_10.

[2] Khawar Zaman Ahmed and Cary E. Umrysh. *Developing Enterprise Java Applications with J2EE and UML*. 1st ed. Addison-Wesley Professional, 2001, p. 368. ISBN: 978-0201738292.

[3] Deborah J. Armstrong. “The quarks of object-oriented development”. In: *Communications of the ACM* 49.2 (Feb. 2006), pp. 123–128. ISSN: 0001-0782. DOI: 10.1145/1113034.1113040.

[4] Colin Atkinson and Thomas Kühne. “A Generalized Notion of Platforms for Model-Driven Development”. In: *Model-Driven Software Development*. Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Springer Berlin Heidelberg, 2005, pp. 119–136. ISBN: 978-3-540-28554-0. DOI: 10.1007/3-540-28554-7_6.

[5] Balbir S. Barn and Samia Oussena. “BPMN, Toolsets, and Methodology: A Case Study of Business Process Management in Higher Education”. In: *Information Systems Development*. Ed. by George Angelos Papadopoulos et al. Springer US, 2010, pp. 685–693. ISBN: 978-0-387-84810-5. DOI: 10.1007/b137171_71.

[6] Boualem Benatallah et al. “Developing Adapters for Web Services Integration”. In: *Advanced Information Systems Engineering*. Vol. 3520. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 415–429. ISBN: 978-3-540-26095-0. DOI: 10.1007/11431855_29.

[7] Matthias Böhm et al. *Model-Driven Generation of Dynamic Adapters for Integration Platforms*. 2008.

[8] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. 1st ed. Addison-Wesley Professional, 1998, p. 512. ISBN: 978-0201571684.

[9] Alan W. Brown. “Model driven architecture: Principles and practice”. In: *Software and Systems Modeling* 3 (4 2004). 10.1007/s10270-004-0061-2, pp. 314–327. ISSN: 1619-1366. URL: http://dx.doi.org/10.1007/s10270-004-0061-2.

[10] Alan Brown, Jim Conallen, and Dave Tropeano. “Introduction: Models, Modeling, and Model-Driven Architecture (MDA)”. In: *Model-Driven Software Development*. Ed. by Sami Beydeda, Matthias Book, and Volker Gruhn. Springer Berlin Heidelberg, 2005, pp. 1–16. ISBN: 978-3-540-28554-0. DOI: 10.1007/3-540-28554-7_1.

[11] Fei Cao et al. “Component Specification and Wrapper/Glue Code Generation with Two-Level Grammar Using Domain Specific Knowledge”. In: *Formal Methods and Software Engineering*. Ed. by Chris George and Huaikou Miao. Vol. 2495. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, pp. 103–107. ISBN: 978-3-540-00029-7. DOI: 10.1007/3-540-36103-0_13.

[12] Kai Chen et al. “Semantic Anchoring with Model Transformations”. In: *Model Driven Architecture - Foundations and Applications*. Ed. by Alan Hartman and David Kreische. Vol. 3748. Lecture Notes in Computer Science. Springer Berlin - Heidelberg, 2005, pp. 115–129. ISBN: 978-3-540-30026-7. DOI: 10.1007/11581741_10.

[13] Mihai Ciocoiu, Dana S. Nau, and Michael Gruninger. “Ontologies for integrating engineering applications”. In: *Journal of Computing and Information Science in Engineering* 1.1 (Mar. 2001), pp. 12–22. ISSN: 1530-9827. DOI: 10.1115/1.1344878.

[14] cplusplus.com. *C++ Reference*. cplusplus.com. 2012. URL: http://www.cplusplus.com/reference/.

[15] Alan Dennis, Barbara Haley Wixom, and David Tegarden. *Systems Analysis and Design with UML*. 4th ed. Wiley, 2012, p. 592. ISBN: 978-1118037423.

[16] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-specific languages: an annotated bibliography”. In: *ACM SIGPLAN Notices* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340. DOI: 10.1145/352029.352035.

[17] S. Edwards et al. “Design of embedded systems: formal models, validation, and synthesis”. In: *Proceedings of the IEEE* 85.3 (Mar. 1997), pp. 366–390. ISSN: 0018-9219. DOI: 10.1109/5.558710.

[18] Hajo Eichler, Markus Scheidgen, and Michael Soden. *A Meta-Modelling Framework for Modelling Semantics in the Context of Existing Domain Platforms*. TR, University of Humboldt. Berlin (Germany), 2006.

[19] Matthew Emerson and Janos Sztipanovits. “Techniques for metamodel composition”. In: *The 6th OOPSLA Workshop on Domain-Specific Modeling (OOPSLA)*. ACM, ACM Press, 2006, pp. 123–139.

[20] Jakob Engblom. *On Hardware and Hardware Models for Embedded Real-Time Systems*. 2001.

[21] The Eclipse Foundation. *AspectJ*. 2012. URL: http://www.eclipse.org/aspectj/.

[22] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley Professional, 2003, p. 208. ISBN: 978-0321193681.

[23] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. 1st ed. Wiley, 2003, p. 352. ISBN: 978-0471319207.

[24] <FreeMarker>. *FreeMarker Java Template Engine*. 2012. URL: http://freemarker.sourceforge.net/.

[25] Giovanni Giachetti, Beatriz Marín, and Oscar Pastor. “Using UML as a Domain-Specific Modeling Language: A Proposal for Automatic Generation of UML Profiles”. In: *Advanced Information Systems Engineering*. Ed. by Pascal van Eck, Jaap Gordijn, and Roel Wieringa. Vol. 5565. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009, pp. 110–124. ISBN: 978-3-642-02143-5. DOI: 10.1007/978-3-642-02144-2_13.

[26] Christian Glodt, Pierre Kelsen, and Elke Pulvermüller. “DEMOCLES: A Tool for Executable Modeling of Platform-Independent Systems”. In: *OOPSLA Companion*. Montreal, Canada, 2007.

[27] Hassan Gomaa. *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. 1st ed. Cambridge University Press, 2011, p. 576. ISBN: 978-0521764148.

[28] James Gosling et al. *The Java Language Specification, Java SE 7 Edition*. Oracle Inc. July 2012.

[29] Gregor Gössler and Joseph Sifakis. "Composition for component-based modeling". In: *Science of Computer Programming - Formal methods for components and objects pragmatic aspects and applications* 55.1-3 (Mar. 2005), pp. 161–183. ISSN: 0167-6423. DOI: 10.1016/j.scico.2004.05.014.

[30] R. Grangel et al. "Ontology Based Communications Through Model Driven Tools: Feasibility of the MDA Approach in Urban Engineering Projects". In: *Ontologies for Urban Development*. Ed. by Jacques Teller, John Lee, and Catherine Roussey. Vol. 61. Studies in Computational Intelligence. Springer Berlin / Heidelberg, 2007, pp. 181–196. ISBN: 978-3-540-71975-5. DOI: 10.1007/978-3-540-71976-2_16.

[31] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. 1st ed. Addison-Wesley Professional, 2009, p. 736. ISBN: 978-0321534071.

[32] Object Management Group. *Model Driven Architecture (MDA) Guide, V1.0.1*. Tech. rep. June 2003. URL: http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[33] Object Management Group. *Object Constraint Language (OCL), V2.3.1*. Tech. rep. Jan. 2012. URL: http://www.omg.org/spec/OCL/2.3.1/PDF.

[34] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models (FUML), V1.0*. Tech. rep. Feb. 2011. URL: http://www.omg.org/spec/FUML/1.0/PDF.

[35] Object Management Group. *Unified Modeling Language (UML), Infrastructure, V2.4.1*. Tech. rep. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF.

[36] Object Management Group. *Unified Modeling Language (UML), Superstructure, V2.4.1*. Tech. rep. Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF.

[37] Henri Habrias. *Case Study Text: Invoicing Orders*. Apr. 1996. URL: http://www.dmi.usherb.ca/~spec/texte-cas-invoicing.htm.

[38] Christian Hofer et al. "Polymorphic embedding of dsls". In: *Proceedings of the 7th international conference on Generative programming and component engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 137–148. ISBN: 978-1-60558-267-2. DOI: 10.1145/1449913.1449935.

[39] Paul Hudak. "Building domain-specific embedded languages". In: *ACM Comput. Surv.* 28.4es (Dec. 1996). ISSN: 0360-0300. DOI: 10.1145/242224.242477.

[40] Paul Hudak. "Modular Domain Specific Languages and Tools". In: *Proceedings of the 5th International Conference on Software Reuse*. ICSR '98. IEEE Computer Society, 1998, pp. 134–. ISBN: 0-8186-8377-5.

[41] IBM Inc. *EMF Ecore package API*. Aug. 2012. URL: http://download.eclipse.org/modeling/emf/emf/javadoc/2.8.0/org/eclipse/emf/ecore/package-summary.html.

[42] Adobe Systems Inc. *Adobe Flash Professional CS6*. 2012. URL: http://www.adobe.com/products/flash.html.

[43] Adobe Systems Inc. *PhoneGap*. 2012. URL: http://phonegap.com/.

[44] Appcelerator Inc. *Appcelerator*. 2012. URL: http://www.appcelerator.com/.

[45] Apple Inc. *Quartz 2D Programming Guide*. 2010. URL: https://developer.apple.com/library/mac/#documentation/graphicsimaging/conceptual/drawingwithquartz2d/dq_overview/dq_overview.html.

[46] Corona Labs Inc. *Corona SDK.* 2012. URL: `http://www.coronalabs.com/products/corona-sdk/`.

[47] Google Inc. *WindowBuilder User Guide.* Google Inc. 2011. URL: `http://code.google.com/javadevtools/wbpro/index.html`.

[48] Oracle Inc. *Java 2D API.* 2010. URL: `http://java.sun.com/products/java-media/2D/index.jsp`.

[49] Staircase 3 Inc. *The many faces of a little green robot.* 2012. URL: `http://opensignalmaps.com/reports/fragmentation.php`.

[50] Pierre Kelsen. *A Declarative Executable Model for Object-Based Systems Based on Functional Decomposition.* Tech. rep. TR-LASSY-06-06. University of Luxembourg, May 2006.

[51] Pierre Kelsen and Qin Ma. *A Formal Definition of the EP Language.* Tech. rep. May. University of Luxembourg, LASSY, 2008. URL: `http://wiki.lassy.uni.lu/Special:LassyBibDownload?id=2508`.

[52] Pierre Kelsen and Qin Ma. "Domain Hierarchies: A Basic Theoretical Framework for Integrating Software Domains". In: *Theoretical Aspects of Software Engineering* 1 (2009), pp. 295–296. DOI: `10.1109/TASE.2009.55`.

[53] Stuart Kent. "Model Driven Engineering". In: *Proceedings of the Third International Conference on Integrated Formal Methods.* IFM '02. Springer-Verlag, 2002, pp. 286–298. ISBN: 3-540-43703-7.

[54] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* 1st ed. Addison-Wesley Professional, 2003, p. 192. ISBN: 978-0321194428.

[55] LASSY. *Democles.* Sept. 2011. URL: `http://democles.lassy.uni.lu/`.

[56] Ideaworks 3D Limited. *Marmalade.* 2012. URL: `http://www.madewithmarmalade.com/`.

[57] Peter Marwedel. *Embedded System Design.* Ed. by Nikil D. Dutt and Peter Marwedel. 2nd ed. Springer Science+Business Media B.V., 2011. ISBN: 978-1-4419-0503-1. DOI: `10.1007/978-94-007-0257-8`.

[58] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201748045.

[59] Stephen J. Mellor et al. *MDA Distilled.* Addison-Wesley Professional, 2004, p. 176. ISBN: 978-0201788914.

[60] Stephan Merz and Nicolas Navet, eds. *Modeling and Verification of Real-Time Systems.* ISTE, 2008, p. 448. ISBN: 978-1847040244.

[61] MetaCase. *MetaEdit+ Workbench - Build your own Domain-Specific Modeling language.* 2012. URL: `http://www.metacase.com/mwb/`.

[62] Russ Miles and Kim Hamilton. *Learning UML 2.0.* 1st ed. O'Reilly Media, 2006, p. 290. ISBN: 978-0596009823.

[63] Dragan Milicev. *Model Driven Development with Executable UML.* Wiley Publishing, Inc., 2009. ISBN: 978-0-470-48163-9.

[64] Mika Mobile. *Our Future with Android.* 2012. URL: `http://mikamobile.blogspot.com/2012/03/our-future-with-android.html`.

[65] Inc. Motorola Solutions. *Rhomobile Suite.* 2012. URL: `http://www.motorola.com/Business/US-EN/Business+Product+and+Services/Software+and+Applications/RhoMobile+Suite`.

[66] Michael Muehlen and Jan Recker. “How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation”. In: *Advanced Information Systems Engineering*. Ed. by Zohra Bellahsène and Michel Léonard. Vol. 5074. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, pp. 465–479. ISBN: 978-3-540-69533-2. DOI: 10.1007/978-3-540-69534-9_35.

[67] Frédéric Peschanski. “A versatile event-based communication model for generic distributed interactions”. In: *Proceedings of the 22snd International Conference on Distributed Computing Systems Workshops (ICDCSW'02)*. 2002, pp. 503–510. DOI: 10.1109/ICDCSW.2002.1030818.

[68] *PostSharp - Code to the Point*. SharpCrafters s.r.o. 2011. URL: http://www.sharpcrafters.com/.

[69] Rachel A. Pottinger and Philip A. Bernstein. “Merging models based on given correspondences”. In: *Proceedings of the 29th international conference on Very large data bases - Volume 29*. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 862–873. ISBN: 0-12-722442-4.

[70] Chris Raistrick, Paul Francis, and John Wright. *Model Driven Architecture with Executable UML(TM)*. Cambridge University Press, 2004. ISBN: 0521537711.

[71] Catherine Roussey et al. “Ontologies for Interoperability”. In: *Ontologies in Urban Development Projects*. Ed. by Lakhmi C. Jain and Xindong Wu. Vol. 1. Advanced Information and Knowledge Processing. Springer London, 2011, pp. 39–53. ISBN: 978-0-85729-724-2. DOI: 10.1007/978-0-85729-724-2_3.

[72] David Ruddock. *Mika Mobile's Decision To Stop Making Games For Android Raises Troubling Questions And Concerns About Developing For The Platform*. 2012. URL: http://www.androidpolice.com/2012/03/12/mika-mobiles-decision-to-stop-making-games-for-android-raises-troubling-questions-and-concerns-about-apps-on-android/.

[73] Bran Selic. “The pragmatics of model-driven development”. In: *Software, IEEE* 20.5 (Sept. 2003), pp. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146.

[74] Darius Silingas et al. “Domain Specific Modeling Environment Based on UML Profiles”. In: 2009.

[75] *Snap - Simple Aspects*. Notion One. 2010. URL: http://www.simpleaspects.com/.

[76] Institute for Software Integrated Systems. *GME: Generic Modeling Environment*. Vanderbilt University. 2008. URL: http://www.isis.vanderbilt.edu/Projects/gme/.

[77] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. 2nd ed. Addison-Wesley Professionals, 2009, p. 744. ISBN: 978-0-321-33188-5.

[78] Dave Steinberg et al. “Model Editing with EMF.Edit”. In: *EMF: Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. 2nd ed. Addison-Wesley Professionals, 2009, pp. 41–68. ISBN: 978-0-321-33188-5.

[79] Dave Steinberg et al. “Running the Generators”. In: *EMF: Eclipse Modeling Framework*. Ed. by Erich Gamma, Lee Nackman, and John Wiegand. 2nd ed. Addison-Wesley Professionals, 2009, pp. 41–68. ISBN: 978-0-321-33188-5.

[80] *The Home of AspectC++*. 2012. URL: http://www.aspectc.org/.

[81] Christian Ullenboom. *Java 7 - Mehr als eine Insel*. Ed. by Judith Stevens-Lemoine. 2011th ed. Galileo Press, 2012, p. 1434.

[82] Christian Ullenboom. *Java ist auch eine Insel*. Ed. by Judith Stevens-Lemoine. 10th ed. Galileo Press, 2012, p. 1309.

[83] Christian Ullenboom. “Java ist auch eine Sprache”. In: *Java ist auch eine Insel.* Ed. by Judith Stevens-Lemoine. 10th ed. Galileo Press, 2012, pp. 55–83.

[84] Antonio Vallecillo. “On the Combination of Domain Specific Modeling Languages”. In: *Modelling Foundations and Applications.* Ed. by Thomas Kühne et al. Vol. 6138. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, pp. 305–320. ISBN: 978-3-642-13594-1. DOI: `10.1007/978-3-642-13595-8_24`.

[85] Various. *Aspect-oriented programming.* Aug. 2012. URL: `http://en.wikipedia.org/wiki/Aspect-oriented_programming`.

[86] Various. *Software Specification Methods.* Ed. by Henri Habrias and Marc Frappier. ISTE Ltd, 2006.

[87] *Visual Basic.* Microsoft Inc. 2012. URL: `http://msdn.microsoft.com/en-us/vstudio/hh388573`.

[88] *Visual C#.* Microsoft Inc. 2012. URL: `http://msdn.microsoft.com/en-us/vstudio/hh388566.aspx`.

[89] Dennis Wagelaar and Viviane Jonckers. “Explicit Platform Models for MDA”. In: *Model Driven Engineering Languages and Systems.* Ed. by Lionel Briand and Clay Williams. Vol. 3713. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 367–381. ISBN: 978-3-540-29010-0. DOI: `10.1007/11557432_27`.