university of groningen

Bachelor Project

ArchiMate

Generating Code for UML Diagrams

Author:

Samuel Esposito (s1597183)

Supervisors:

Dr. Paris Avgeriou

Ahmad Waqas Kamal

Organization:

Department of Mathematics and Computing Science

University of Groningen

Year:

2008-2009

Table of Contents

Generating Code for UML Diagrams

1. Introduction

1.1 Unified Modeling Language

The Unified Modeling Language (UML) is a standard language used to specify and visualize the architecture of a software system under construction. The language provides a rich set of elements for modeling the structural as well as the behavioral aspects of a software system. These aspects can then be graphically represented using structure and behavior diagrams.
UML is an extensible language offering profiles and stereotypes as mechanisms for customization.
The benefits of UML are often summarized in the saying 'A picture is worth a thousand words', because its diagrams greatly ease the communication on software architecture.

1.2 Code Generation

Code generation in the context of UML is the derivation of parts of the source code for a software system from UML diagrams. Thus the process of writing code is partly automated.
A concept closely related to code generation is reverse engineering. This is the generation of UML diagrams from source code to reflect its structure and behavior. The generated diagrams ease the understanding of existing code and the process of software maintenance.
When code generation and reverse engineering are combined, a process called round trip engineering is obtained. This process means to keep the UML diagrams and the code consistent through the development of software.

1.3 Relevance

The purpose of code generation is to free the programmer of writing and updating the low level source code of a software system, which especially in larger projects can be very time consuming. With code generation UML can become an efficient meta-language to build the base of a software system. The time thus won can be used to focus on the architecture and required functionality of a software system.

2. Related Work

2.1 Existing Solutions

At the beginning of the 21st century, software solutions that generate code from UML diagrams started to emerge: so-called UML tools. Nowadays tens of tools are available for the developer, many of which are part of larger Integrated Development Environments (IDE) or take the form of plugins for an existing IDE. Most of these UML tools mainly focus on class diagrams defining the structure of a system. They generate the class declarations, constructor, getters, setters and method declarations corresponding with the UML elements of a diagram. Some tools in addition generate code for sequence diagrams reflecting the behavior of the software components.

2.2 Problems and Challenges

Even though the existing tools are very helpful, the value of the generated source code is limited. It is mostly a collection of unrelated packages and class declarations. This is because for most UML elements there is no obvious or unambiguous translation to source code.

Also the main focus is on the structure of the software architecture. The behavior is mostly not covered, because it's complicated to generate code for.

Furthermore the existing solutions don't help the programmer creating sound architectures and implement these architectures in a correct way. They provide no means to validate either the architecture or the implementation.

3. Concept

In this bachelor project we introduce design patterns and primitives to the code generation process in an attempt to solve the problems and challenges in existing UML tools. Design patterns are general reusable solutions to commonly occurring architectural problems in software design. Design primitives are similar to patterns, but have usually a smaller scope. A design primitive can be part of a design pattern, adding finer grain to it.

Patterns are not a finished design and thus cannot be directly translated into source code. However by combining the structural and behavioral information of UML models with the interactional constraints derived from design patterns, the gap between UML and meaningful source code grows significantly smaller. By adding validation on the modeled architecture and the implementation of this architecture, code generation for UML diagrams is taken to a challenging new level.

3.1 UML Diagrams: Structure and Behavior

UML provides a rich variety of diagrams that can be divided in two categories: structure diagrams and behavior diagrams. Figure 1 gives a complete oversight of the diagrams UML 2 defines.

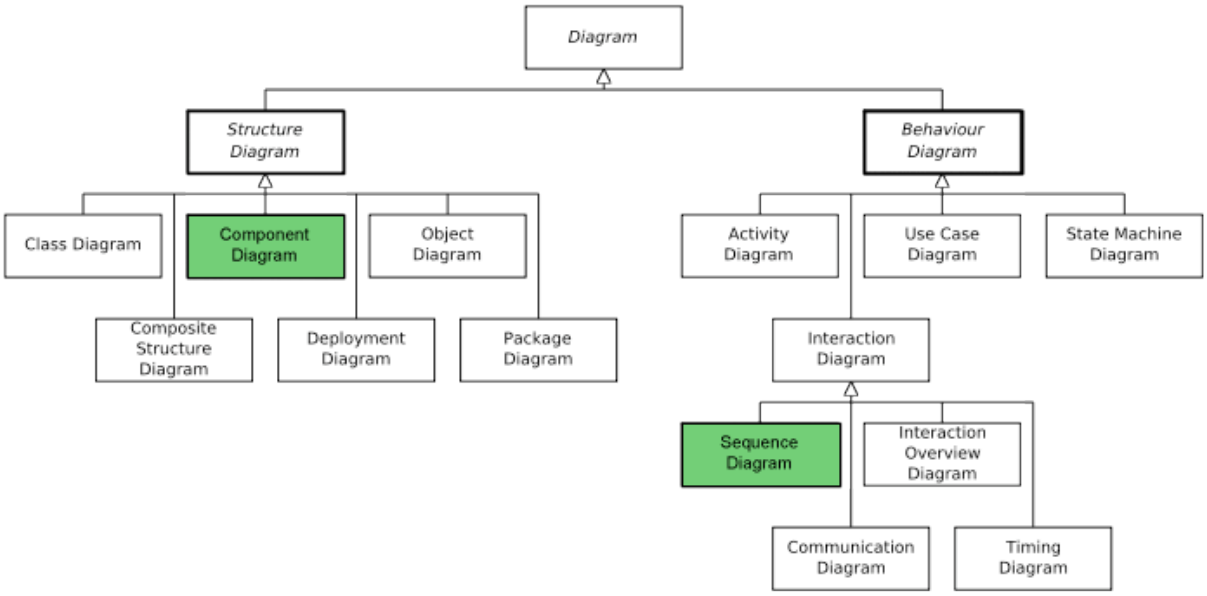

Figure 1: This tree contains all diagrams defined by UML 2.

As its name indicates, the first category consists of diagrams that define the structure of a software architecture. In the context of code generation, most UML tools choose class diagrams to let the user define the structure of the software, as its UML elements are closest to source code elements as classes, attributes and methods. In this project we however choose to generate code from the component diagram, because it lets us define the software architecture at a more abstract level. This is necessary for defining patterns and primitives without putting too many constraints on their concrete implementation.

The component diagram provides components, connectors, ports and interfaces as key elements for modeling. The components can be used to define the objects from the domain model. Components can interact with their environment through ports, which provide or make use of interfaces. Ports of different components can be connected with a connector to define their interaction. To be able to connect two ports, obviously each port needs to provide the interfaces the other port requires.

Figure 2 displays the UML meta-model for component diagrams.

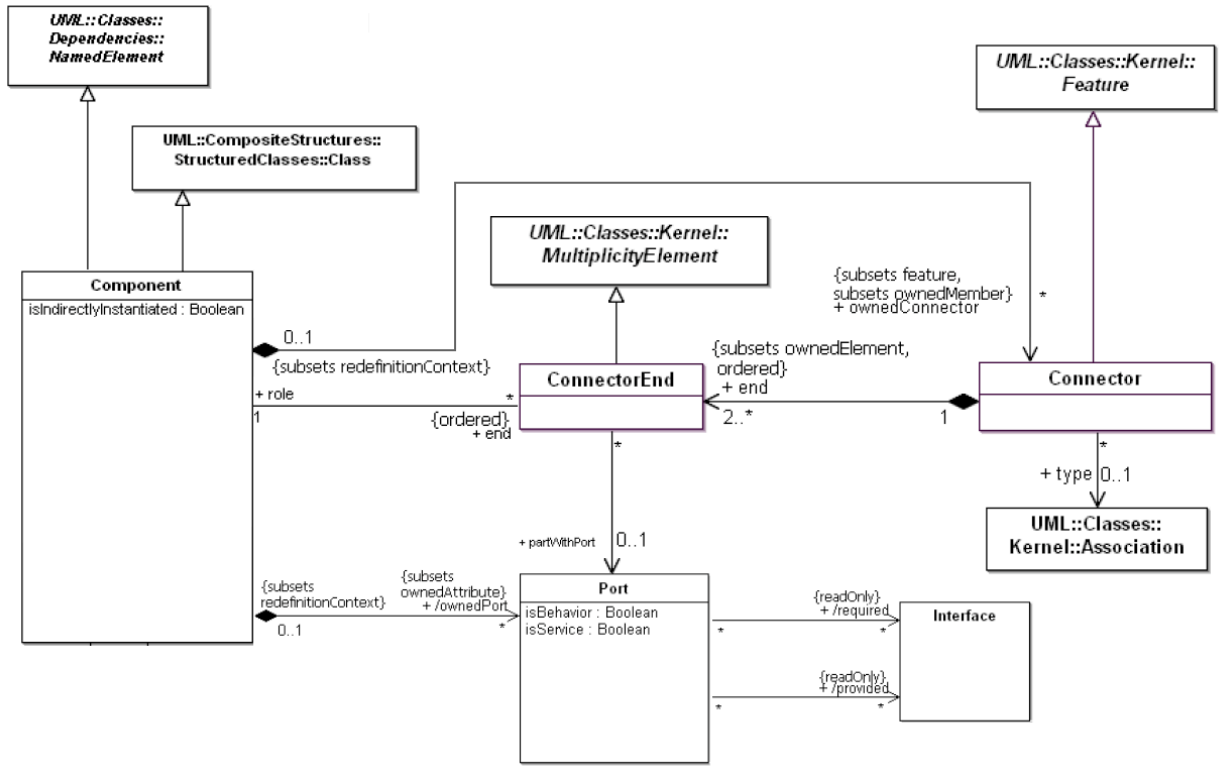

Figure 2: This diagram represents the UML metamodel for a component diagram.

To express the behavior of a software system we choose to use the UML sequence diagram, because it lets you express a sequence of interactions between any objects from the domain model.

A sequence diagram provides lifelines, messages, message ends and execution specifications as key UML elements to define the model. All elements are contained in an interaction. A lifeline can represent any object from the domain model. Lifelines can send messages to each other through message occurrence specifications (message ends). Message occurrence specifications can mark the beginning and end of behavior execution specifications, which represent the execution of a routine.

Figure 3 displays the meta-model for the UML sequence diagram.

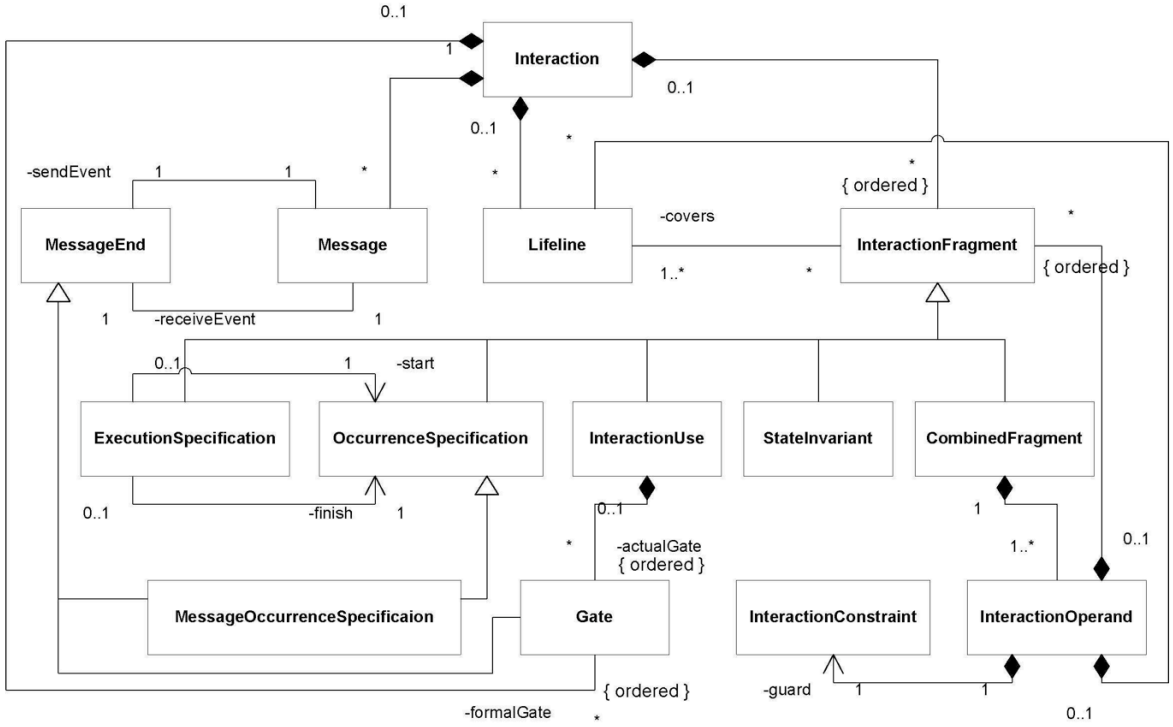

Figure 3: This diagram represents the UML metamodel for a sequence diagram.

3.2 Extending UML: Profiles and Stereotypes

Apart from the structural and behavioral aspects of an architecture, in this project we want to take into account the interactional constraints of any applied design pattern or primitive when generating source code. To express these patterns and primitives, we have to extend UML using profiles.

A profile in UML provides a generic mechanism for customizing models for particular domains. Profiles are defined using stereotypes that extend specific UML elements, such as components and lifelines. A profile is a collection of such extensions that can be applied to a UML model.

So in order to express a pattern or primitive in a UML model, a UML profile has to be defined for it. This profile can then be applied to a UML model and the stereotypes in the profile can be applied to the UML elements in the model, indicating that they make part of the expressed pattern or primitive.

Figure 4 displays the UML meta-model for profiles

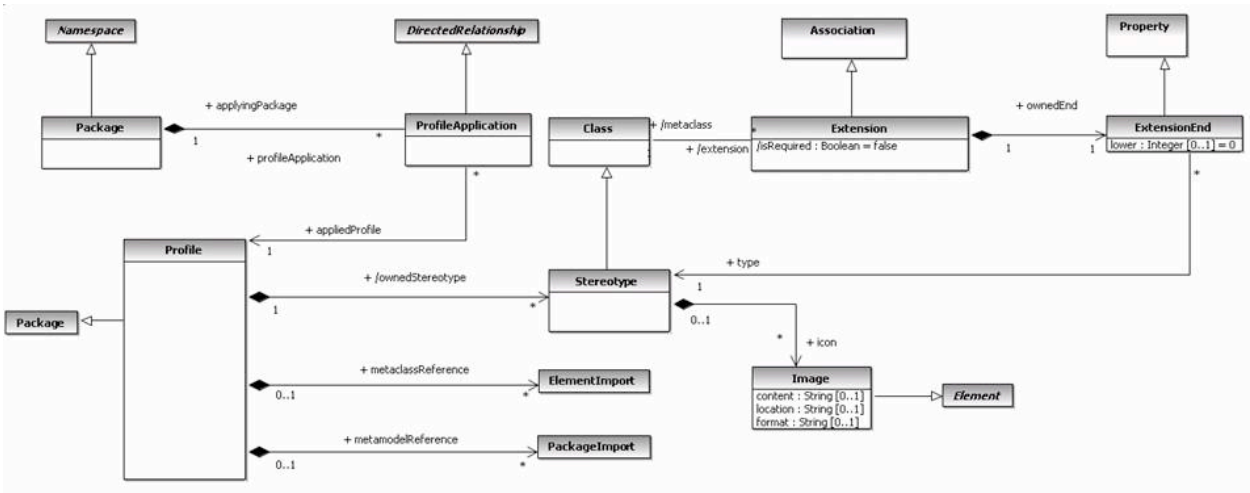

Figure 4: This diagram represents the UML metamodel for profiles.

3.3 Model Validation: Object Constraint Language

The previous section described how to model a design pattern or primitive in UML. But how can we make sure that the modeled architecture is sound? This problem can be solved using the Object Constraint Language (OCL), a declarative language for describing rules that apply to UML models.

Using OCL the constraints on the structure and behavior of the architecture can be specified and verified on a very detailed level.

The definition and implementation of the OCL constraints is not further discussed in this thesis, as it is handled elaborately in Johan Drenthens thesis about UML diagram synchronization.

3.4 Code Generation

Once a sound UML diagram is created defining the structure, behavior and interactional constraints of the architecture, we can turn to source code generation.

The UML component diagram defining the structure of the architecture will be translated into abstract classes and superinterfaces implemented by these classes. This part of the source code merely defines the structure of the software system. The UML sequence diagram representing the behavior of the software system will result in classes, extending the abstract classes, the interfaces implemented by these classes, extending the superinterfaces, and finally method declarations and their implementation and invocation. This part of the source code really implements the functionality and thus the behavior of the software.

When generating code, for both types of diagrams the same process is used for integrating the newly generated code with the already existing code.

First all necessary information of the UML elements representing the applied patterns and primitives is collected from the UML diagram. Based on this information a model is created defining the implementation corresponding with each relevant UML element.

In a second phase, the model is matched with the existing code. In this process the missing parts in the implementation of the architecture are filtered out. Finally the necessary source code is added to the existing code.

When the code generation is complete, the source code is said to fully implement the architecture modeled in the UML diagrams. By analyzing the existing code before generating code, we avoid adding duplicate or superficial lines of code. Every line of generated code really plays a role in implementing the architecture.

3.5 Code Validation

The model discussed in the previous section, representing the implementation of all UML elements defining the systems architecture, can be a powerful tool when it comes to code validation. When matching this model with the existing code, not only the missing source code elements, but also the soundness of the existing implementation and code conflicting with the model can be identified. Furthermore the hierarchical ordering of the code elements can be checked.

The result of this process is a detailed report on the soundness of the implementation of the architecture and the possible violation of its constraints. Based on this report, suggestions can be made to change the code in order to better implement the architecture.

3.6 Reverse Engineering

Once the automatically generated source code is in place, a large portion of it will change during the process of software development. To keep a clear vision on the structure and behavior of the system, the UML diagrams representing the architecture can be updated regularly to reflect these changes.
For this process, called reverse engineering, the code generation procedure has to be reversed. By analyzing the source code a model has to be constructed representing the implemented UML elements. This model can then be compared to the existing UML diagrams to identify the missing or changed UML elements. With this information, the UML diagrams can be updated and extended, or new diagrams can be created.

3.7 Design Patterns

A design pattern is a general reusable solution to a commonly occurring architectural problem in software design. In this section we discuss the commonly used MVC pattern, for which in chapter 4 code generation is described as a use case.

3.7.1 Model View Controller

The Model View Controller (MVC) pattern is an architectural pattern used in software engineering. The pattern separates the business logic from the application data and its presentation, permitting independent development, testing and maintenance of each.

In an MVC application each model is associated with one or more views suitable for presentation. When a model changes its state, it notifies its associated views that new data is available. The controller is responsible for initiating change requests and providing any necessary data inputs to the model.

Figure 5 illustrates the architectural structure of an MVC application.

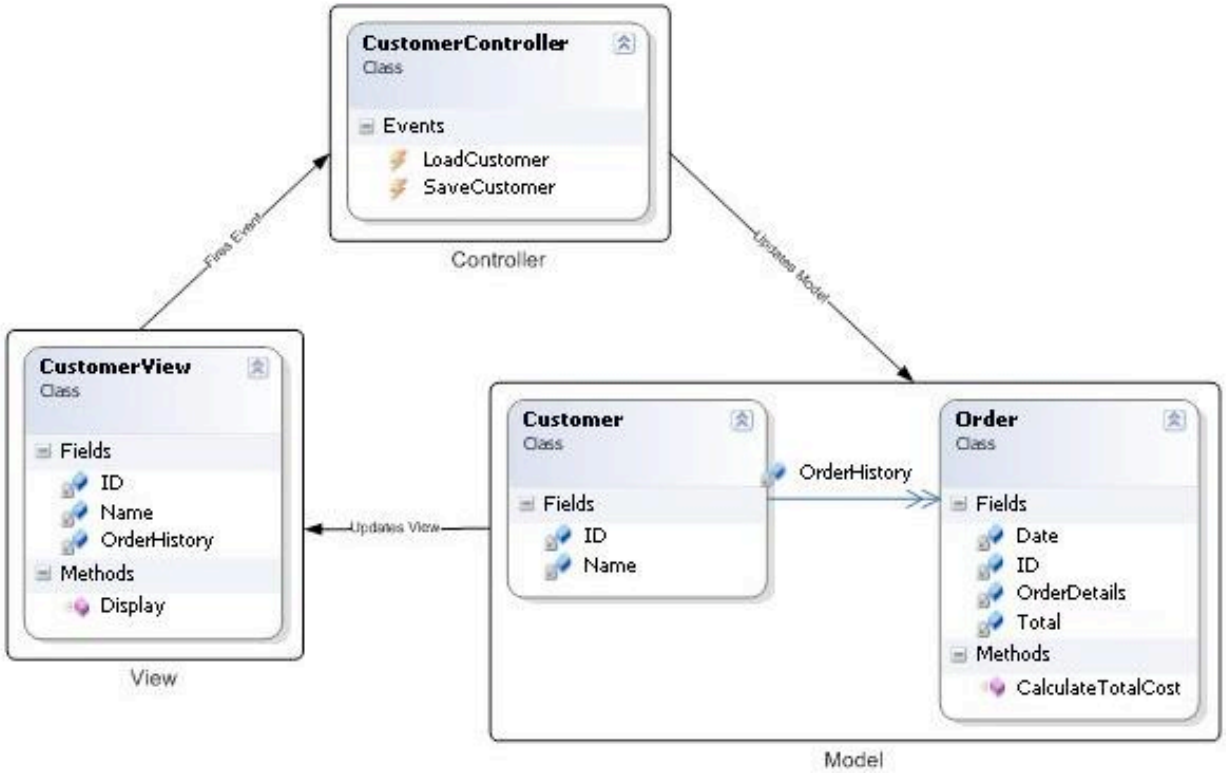

Figure 5: This diagram represents the architectural structure of an MVC application.

We chose this pattern as a use case for code generation because it is a widely used and well-described pattern.

3.8 Design Primitives

A design primitive is general reusable solution to a commonly occurring architectural problem in software design too, though it usually has a smaller scope than a design pattern and can be used inside a pattern to add a finer grain of detail to the architecture. In this section we discuss the commonly used callback primitive, for which in chapter 4 code generation is described as a use case.

3.8.1 Callback

The callback primitive has two actors, the caller and the callee. The idea is that when some particular event happens, the callee calls back to the caller, which can then react on the event in an appropriate way. For the callee to know who is the caller he needs to call back and on what event, the caller has to subscribe to an event at the callee. This is a general solution that allows the handling of an event to be fine-tuned at runtime. It has many applications ranging from exception handling to sorting algorithms.
Figure 6 shows the structure of a callback primitive: first the caller subscribes to an event at the caller. At the occurrence of the event, the callee calls back.

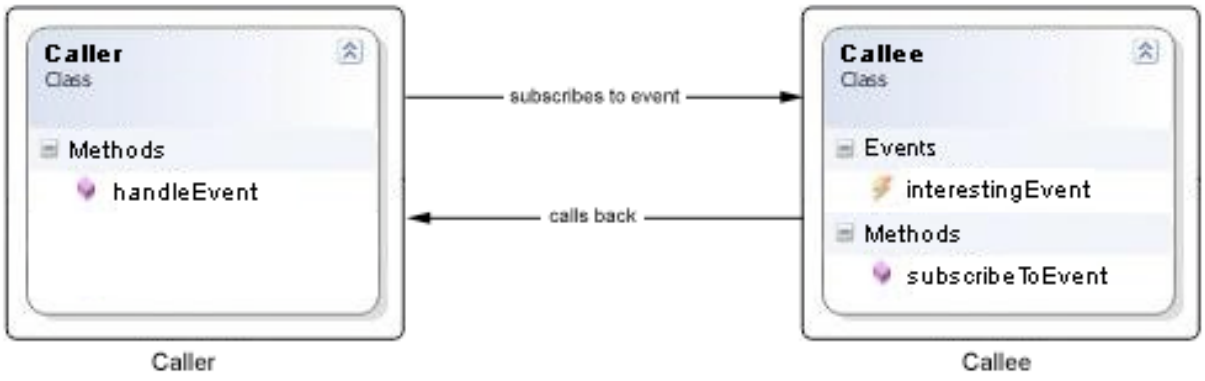

Figure 6: This diagram represents the structure of a callback primitive.

4. Realization

4.1 Eclipse UML2 Framework

In this bachelor project we implement the code generation for UML diagrams in an Eclipse plugin named ArchiMate, your architectural mate.

Eclipse is a multi-language open source Integrated Development Environment (IDE) with a plugin-based architecture for easy extension, initially developed in Java by IBM, and widely used for software development. The UML2 framework of Eclipse is currently the best open source implementation of the UML specification and is widely used as a base for UML tools.

The fact that the IDE is so easily extended by plugins and the possibility to use this UML framework for reading and manipulating UML files are the main reason for choosing Eclipse as a base for the implementation. As a consequence of this choice, the Eclipse Object Constraint Language (OCL) implementation is used to validate the UML models and the Eclipse ASTParser is relied on for parsing, manipulating and validating Java source code. This ASTParser is the java parser Eclipse itself uses for code generation and therefore has excellent and up to date support for all versions of the java language.

4.2 Eclipse Plugin Development

In Eclipse almost all the functionality is provided by plugins. In order to add new functionality to the IDE, the functionality of existing plugins can be extended through so called extension points. The ArchiMate plugin that is constructed in this project extends the functionality of the UML editors in the UML2 framework. More precisely the context menus or the UML2 tree view editor, the component diagram editor and the sequence diagram editor are extended with extra menu options for code generation. In addition when one of these editors is in focus, an extra menu is added to the Eclipse main menu providing the same options for code generation.

Figure 7 shows the extended main menu. Figure 8 shows the extended context menu of the UML editors.

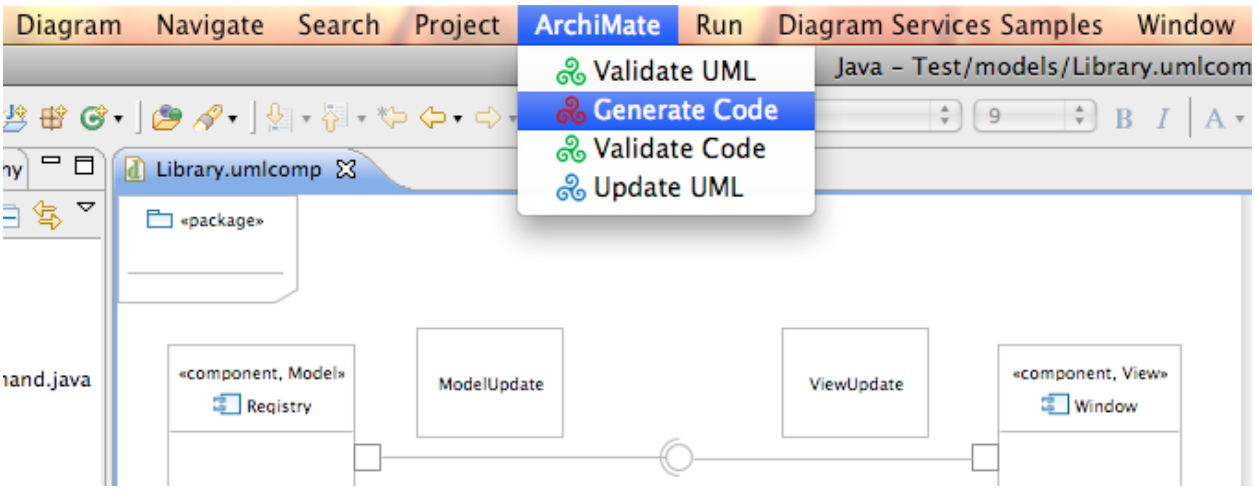

Figure 7: Here you see the menu the ArchiMate plugin adds to the Eclipse main menu. The menu contains the options UML validation, code generation, code validation and updating the UML model.

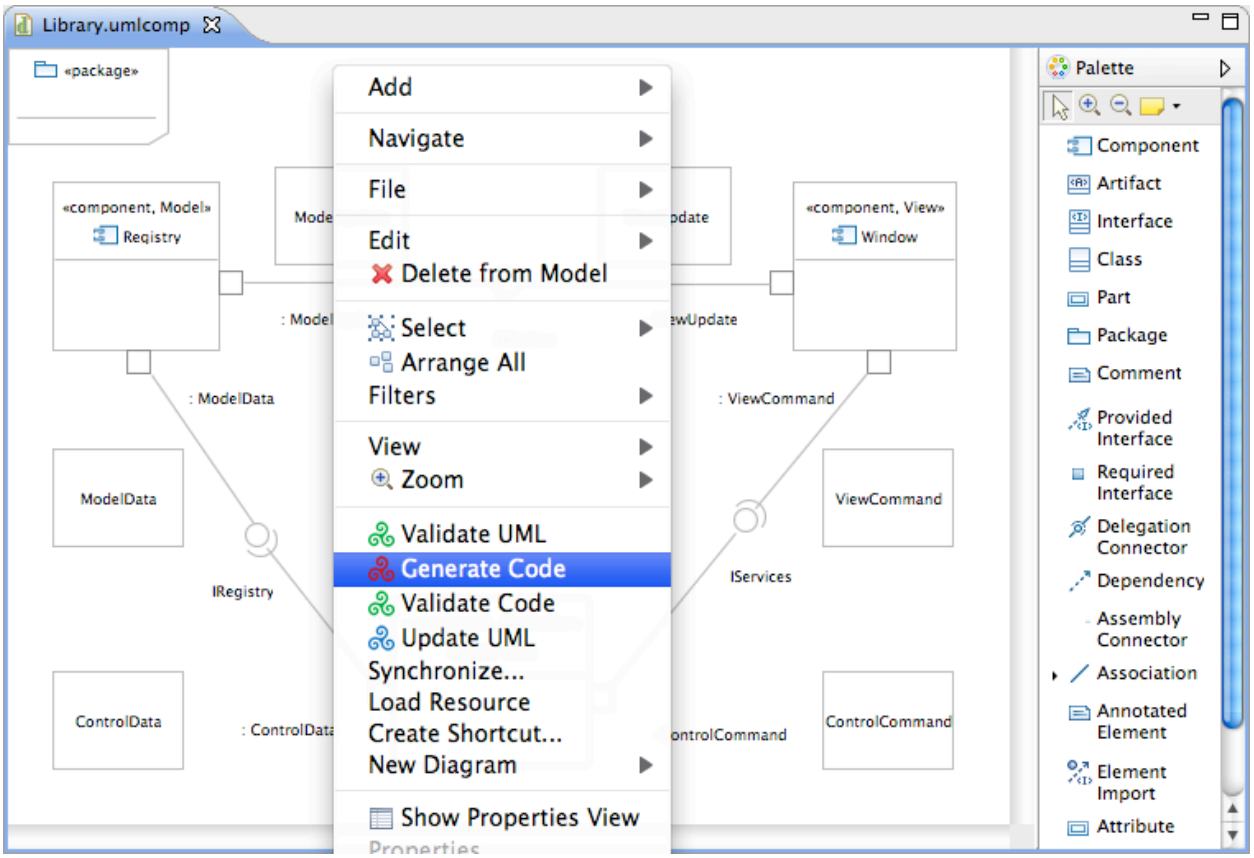

Figure 8: Here you see the context menu of the UML editor extended by the ArchiMate plugin with the options UML validation, code generation, code validation and updating the UML model.

4.3 Implementation

In this section we discuss how the different functionalities of code generation, code validation and reverse engineering are implemented in the ArchiMate plugin.

4.3.1 Implementing Code Generation

The code generation functionality of the ArchiMate plugin can be invoked through the context menu of the UML editor or through the 'ArchiMate' menu in the eclipse main menu. This is illustrated in figure 7 and 8. The first step in code generation is identifying the applied design pattern or primitive in the UML model. This can be done by reading out the UML package from the current editor and searching for an applied profile that was defined for such a pattern or primitive. The Eclipse UML2 API lets you easily search all profiles applied to the package in the open UML file. Once a profile has been identified, all UML elements in the model that have a stereotype from that profile applied to it, have to be collected. This can also be done easily using the UML2 API. This search yields all the UML elements that make part of the pattern or primitive the profile was created for. With this information, the ArchiMate plugin creates a TagTree: a tree structure hierarchically ordering the found UML elements and specifying their source code implementation. This TagTree can be seen as a model representing the implementation of the design pattern or primitive.

The second step in the code generation process is mapping the created TagTree to the existing source code in search for code elements that already implement the pattern or primitive. For the handling of Java source code the eclipse AST API defines the convenient ASTVisitor concept. An ASTVisitor walks the parsed source code as a tree structure and visits the nodes of interest, allowing them to be inspected or edited. The mapping of the TagTree to the source code now happens by simultaneously walking both trees and matching the corresponding elements in two ways. First the name of the stereotype applied to the current UML element is matched with a tag in the JavaDoc of the current source code element. When a match is found, the implementation specification of the UML element itself is compared to the source code elements on identifying attributes as type (package and class name) and name. The parser of the AST API provides the 'resolve binding' option, which enables for easily querying the type of all visited code elements. When the source code element really implements the UML element, the UML element is marked as implemented in the TagTree.

Now we come to the third and final step in code generation. Once all source code has been traversed, the TagTree contains all information about the UML elements that are left unimplemented. For each of these elements the implementation is now added at the right place in the existing source code using the AST API. The UML element is marked as implemented.

During the process of code generation, the ArchiMate plugin shows a progress bar informing the user of the actions executed. The progress bar window also avoids any interference from the user with the code generation taking place. A cancel button allows the user to interrupt the code generation process.

Figure 9 displays the ArchiMate progress bar.

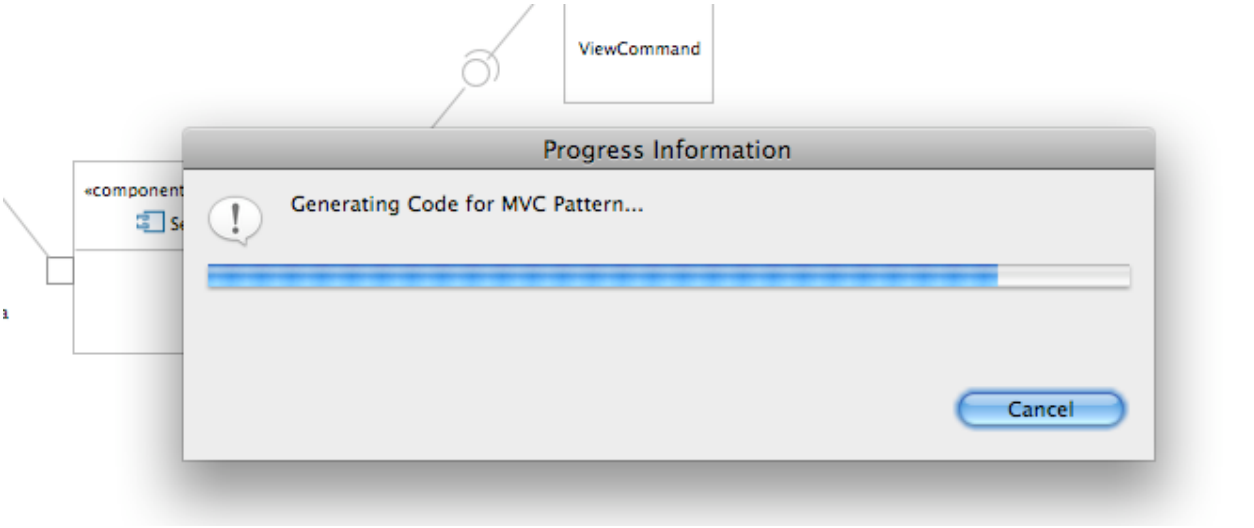

Figure 9: During the process of code generation the ArchiMate Plugin displays a progress bar.

When the code generation process is completed, all UML elements in the TagTree are marked as implemented. Therefore we know that the design pattern or primitive represented by the TagTree is

completely implemented. Because only the missing source code elements were added, we also know no superfluous or duplicate code was added.

In the end, the ArchiMate plugin shows a report of all generated code elements to the user. This is illustrated in figure 10.

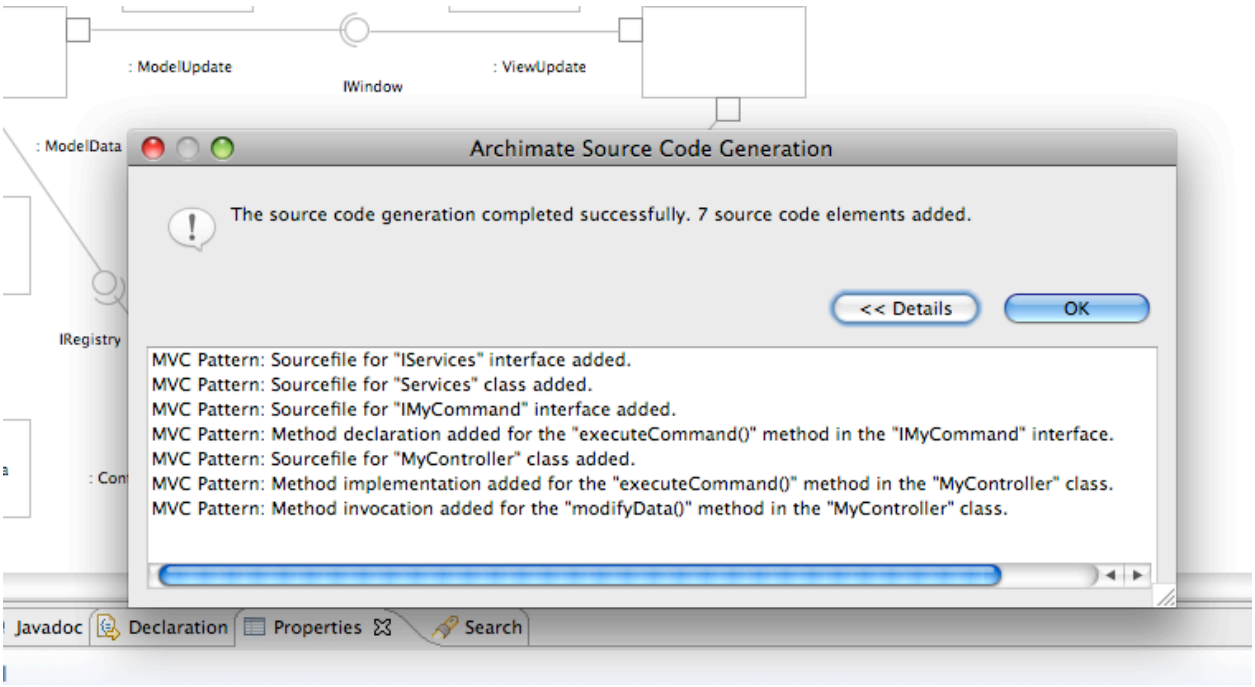

Figure 10: When the process of code generation is finished, the ArchiMate plugin displays a report on all generated code elements.

4.3.2 Implementing Code Validation

Every source code element that implements a part of a design pattern or primitive is marked with a special tag in its JavaDoc. This makes it possible to validate the implementation of a pattern or primitive at any moment during development.

As with code generation, during code validation a TagTree is constructed based on the information extracted from a UML model. This TagTree contains all UML elements relevant to the pattern or primitive and a specification of their implementation. And similar to the process of code generation, the TagTree is mapped to the existing source code in search for code elements that implement the pattern or primitive. When a matching source code element is found, it is validated on completeness and correctness by comparing it to the implementation specification in the TagTree. For every deviation from the intended implementation, a warning or error is generated, which is later incorporated in a detailed report.

During the mapping of the TagTree to the source code, the code is also searched for possible violations of the patterns or primitives interactional constraints. Whenever a code element interacts with a code element that makes part of the implemented pattern or primitive in a way that is not intended, a warning or error is generated.

When all source code has been traversed, the code elements that weren't found during the mapping process are reported too.

During the process of code generation, the ArchiMate plugin displays a progress bar indicating the currently executed action (See figure 9).

In the end the code validation yields a detailed report containing all generated warnings and errors, which is displayed to the user in a status dialog.

In the end the ArchiMate plugin collects all warnings and errors generated in the process of code validation into a report, and displays it in a dialog to the user. This is illustrated in figure 11.

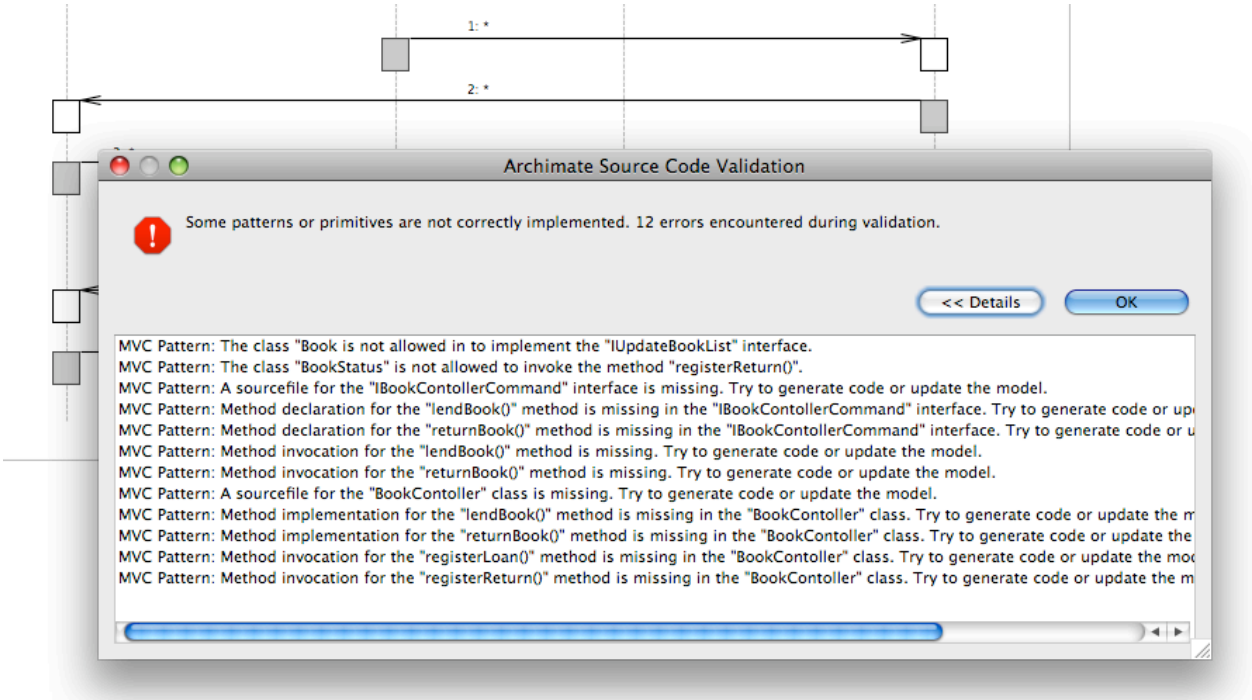

Figure 11: When the process of code validation is complete, the ArchiMate plugin displays a report with all encountered warnings or errors.

4.3.3 Implementing Reverse Engineering

Reverse engineering is implemented in a similar way as code generation and validation. The UML elements that have to be added to the UML diagram are found by matching the source code specifications in the TagTree to the existing source code. Whenever a UML element in the TagTree has fewer children than the matching implementations, a candidate for updating the model is found and further analyzed. In the end for each source code element not represented in the UML diagram a corresponding UML element is created and added in the right place to the diagram.

The reverse engineering functionality of the ArchiMate plugin is only partially implemented and only for the UML2 tree view editor. Therefore it will not be further discussed in this document and is left as a challenge to future developers.

4.4 Use Cases

In the next sections the process of code generation is described for the MVC pattern and the callback primitive discussed in chapter 3. For each pattern or primitive first UML modeling on level M1 and M2 is discussed. Then the key source code elements derived from the UML model are described. After that the validation of the source code is discussed.

In the last section the use of the callback primitive inside the MVC pattern is discussed.

4.4.1 Model View Controller

In this section the process of code generation is discussed for a use case of a library application that is build using the Model View Controller design pattern. First a profile is created to define the structure of the MVC pattern in UML. Then a UML component diagram is created defining the structure of the library application, to which this profile is applied. After that similar steps are taken to define the behavior of the library application. These diagrams then are further used in the discussion of code generation and validation.

4.4.1.1 Defining Profile for MVC Component Diagram: UML M2 Level

In order to be able to identify the key elements of the MVC pattern in a diagram defining the structure of the library application, we have to create stereotypes in a UML profile that specify these key elements. For the structure of the MVC pattern the elements that are most interesting are the components, and the interfaces that define the interaction between these components. So after setting up the profile and referencing the right metaclasses, stereotypes for the tree components of the MVC pattern, model, view and controller, are added and configured so that they extend the UML component element. Then a stereotype for every interface is created and configured so that it extends the UML interface element.

Beneath the precise steps to create an MVC Profile in Eclipse defining the structure of the MVC pattern are summed up. The UML profile is created through the ‘Example EMF Model Creation Wizard’. The actions on the profile are done either through the ‘UML Editor’ menu, the context menu or the properties view of an element. It is imperative that the profile and the stereotypes are named correctly for the ArchiMate plugin to work.

1. Name the profile ‘MVC’.
2. Create an element import for the UML element component.
3. Create an element import for the UML element interface.
4. Create a stereotype named ‘Model’ and create an extension of the UML element component for it.
5. Create a stereotype named ‘View’ and create an extension of the UML element component for it.
6. Create a stereotype named ‘Controller’ and create an extension of the UML element component for it.
7. Create a stereotype named ‘DataInterface’ and create an extension of the UML element interface for it.
8. Create a stereotype named ‘UpdateInterface’ and create an extension of the UML element interface for it.
9. Create a stereotype named ‘CommandInterface’ and create an extension of the UML element interface for it.

Once this profile is created and defined, it is ready to be applied to a component diagram, defining the MVC pattern modeled in it. Figure 12 displays the profile defining the structure of the MVC pattern.

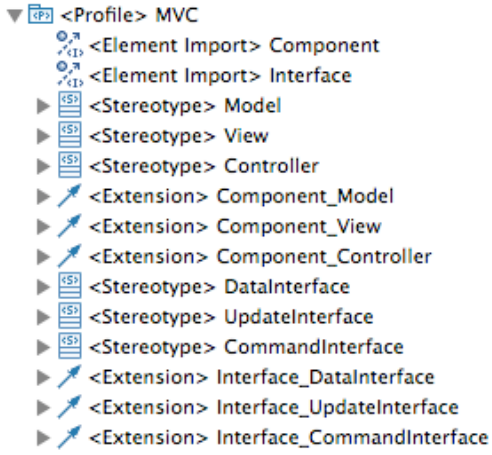

Figure 12: This figure displays the profile defining the structure of the MVC design pattern.

4.4.1.2 MVC Component Diagram: UML M1 Level

The structure of the MVC library application is modeled in a UML component diagram. The diagram has to contain the UML elements every MVC application should at least consist of. First three components have to be added, one for the model, view and controller. These components interact with each other through interfaces one component provides and the other uses.

Beneath the steps to create this diagram using the UML2 tree view editor are summed up. The actions on the model are done either through the 'UML Editor' menu, the context menu or the properties view of an element. The diagram is created using the Eclipse UML 2.1 Component Diagram wizard. The generated '.uml' file has to be opened and edited. It is important to realize that the given class names and interface names will appear in the actual source code, so the use of random names is discouraged.

1. Add a component for the model to the package and give it the (class) name 'Registry'.
2. Add a component for the view to the package and give it the (class) name 'Window'.
3. Add a component for the controller to the package and give it the (class) name 'Services'.
4. Add an interface for the communication between the model and view component and give it the name 'IRegistry'.
5. Add an interface for the communication between the controller and model component and give it the name 'IWindow'.
6. Add an interface for the communication between the view and controller component and give it the name 'IServices'.
7. Add a class for the communication between the view and model implementing the interface from 4.
8. Add an interface implementation to 7 and set its contract to 4.

9. Add a class for the communication between the view and model using the interface from 4.
10. Add a usage and set its client to 9 and its supplier to 4.
11. Add a class for the communication between the model and controller implementing the interface from 5.
12. Add an interface implementation to 11 and set its contract to 5.
13. Add a class for the communication between the model and controller using the interface from 5.
14. Add a usage and set its client to 13 and its supplier to 5.
15. Add a class for the communication between the view and controller implementing the interface from 6.
16. Add an interface implementation to 15 and set its contract to 6 in the properties view.
17. Add a class for the communication between the view and controller using the interface from 6.
18. Add a usage and set its client to 17 and its supplier to 6.
19. Add a port to 1 and type it by 9.
20. Add a port to 1 and type it by 11.
21. Add a port to 2 and type it by 7.
22. Add a port to 2 and type it by 17.
23. Add a port to 3 and type it by 13.
24. Add a port to 3 and type it by 15.

Once this model is created, the profile defined in the previous section can be applied to it in order to identify the key elements of the MVC pattern that will be used in code generation. First the profile needs to be loaded as a resource. Then the following steps can be carried out:

25. Apply the ‘MVC’ profile to the package.
26. Apply the ‘Model’ stereotype to the component 1.
27. Apply the ‘View’ stereotype to the component 2.
28. Apply the ‘Controller’ stereotype to the component 3.
29. Apply the ‘DataInterface’ stereotype to the interface 4.
30. Apply the ‘UpdateInterface’ stereotype to the interface 5.
31. Apply the ‘CommandInterface’ stereotype to the interface 6.

Open the ‘.umlcomp’ file to see the diagram. Nudge the elements to get a more appealing visualization of the structure of the MVC library application. At this stage the UML component diagram is ready for code generation.

Figure 13 displays the created MVC pattern as shown in the diagram editor.

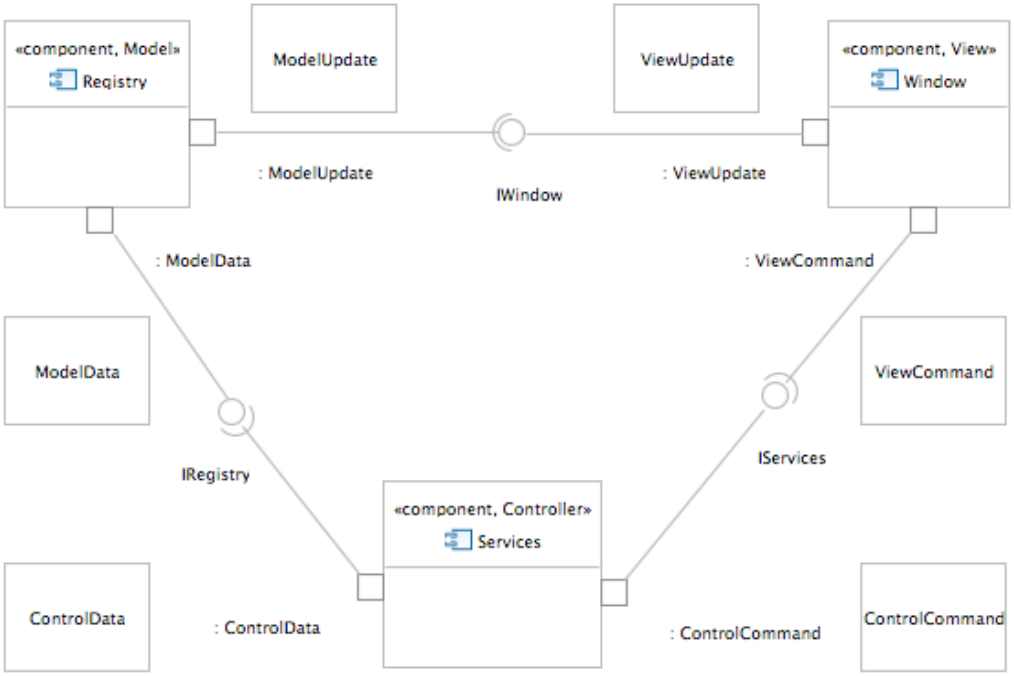

Figure 13: This diagram represents the structure of the MVC library application.

4.4.1.3 Defining Profile for MVC Sequence Diagram: UML M2 Level

For modeling the behavior of the library application sporting an MVC architecture, the elements that are most interesting are the lifelines representing the components, and the messages that define the interaction between these components. So after setting up the profile and referencing the right metaclasses, stereotypes for tree lifelines representing the tree components of the MVC pattern, model, view and controller, are added and configured so that they extend the UML lifeline element. Then a stereotype for the messages between each of the lifelines is added and configured so that it extends the UML message element.
Beneath the steps to create the MVC Profile in Eclipse for the behavior of the MVC pattern are summed up. The UML profile is created through the ‘Example EMF Model Creation Wizard’. The actions on the profile are done either through the ‘UML Editor’ menu, the context menu or the properties view of an element. It is imperative that the profile and the stereotypes are named correctly for the ArchiMate plugin to work.

1. Name the profile ‘MVC’.
2. Create an element import for the UML element lifeline.
3. Create an element import for the UML element message.
4. Create a stereotype named ‘ModelInstance’ and create an extension of the UML element lifeline for it.
5. Create a stereotype named ‘ViewInstance’ and create an extension of the UML element lifeline for it.
6. Create a stereotype named ‘ControllerInstance’ and create an extension of the UML element lifeline for it.

7. Create a stereotype named ‘DataMessage’ and create an extension of the UML element message for it.
8. Create a stereotype named ‘UpdateMessage’ and create an extension of the UML element message for it.
9. Create a stereotype named ‘CommandMessage’ and create an extension of the UML element message for it.

Once this profile is created and defined, it is ready to be applied to a sequence diagram, defining the MVC pattern modeled in it.

4.4.1.4 MVC Sequence Diagram: UML M1 Level

The behavior of the MVC library application is captured in a sequence diagram by modeling the interaction between its components. The lifelines in the sequence diagram represent these components. The interaction between the components is modeled by the messages between the lifelines.

Beneath the steps are described for creating two sequence diagrams representing the behavior of the library application during user registration and the administration of a book loan or return. The actions on the model are done either through the ‘UML Editor’ menu, the context menu or the properties view of an element. The diagram is created using the Eclipse UML 2.1 Sequence Diagram wizard. The generated ‘.uml’ file has to be opened and edited. It is important to realize that the given class names and method names will appear in the actual source code, so the use of random names is discouraged. Also the class names need to differ from the ones used in the component diagram in order to avoid conflicts.

1. Add an interaction to the UML package.
2. Add a lifeline representing the model component and give it the class name ‘User’.
3. Add a lifeline representing the view component and give it the class name ‘UserList’.
4. Add a lifeline representing the controller component and give it the class name ‘UserController’.

Now we describe how to define a message between the ‘UserList’ lifeline and the ‘UserController’ lifeline, representing the ‘addUser()’ method call.

5. Add a message occurrence specification for the start of the method invocation and set the covered attribute to 3.
6. Add a message occurrence specification for the start of the method execution and set the covered attribute to 4.
7. Add a message occurrence specification for the end of the method execution and set the covered attribute to 4.
8. Add a message occurrence specification for the end of the method invocation and set the covered attribute to 3.
9. Add a behavior execution specification for the invocation and set its covered attribute to 3, its start to 5 and its end to 8.
10. Add a behavior execution specification for the execution and set its covered attribute to 4, its start to 6 and its end to 7.
11. Add a message representing the method call, give it the (method) name ‘addUser’ and set its send

event to 5 and its receive event to 6.

12. Set the message attribute of 5 and 6 to 11.

Repeat the steps from 5 to 12 for adding the ‘createUser()’ method call between the ‘UserController’ and the ‘User’ lifelines and once more for adding the ‘updateUserList()’ method call between the ‘User’ and the ‘UserList’ lifelines.

Once the model is complete, the profile defined in the previous section can be applied to it in order to specify the key elements of the MVC pattern that will be used in code generation. First the profile needs to be loaded as a resource. Then the following steps can be carried out:

13. Apply the ‘MVC’ profile to the package.
14. Apply the ‘ModelInstance’ stereotype to the lifeline from 2.
15. Apply the ‘ViewInstance’ stereotype to the lifeline from 3.
16. Apply the ‘ControllerInstance’ stereotype to the lifeline from 4.
17. Apply the ‘CommandMessage’ stereotype any message between lifeline 3 and 4.
18. Apply the ‘DataMessage’ stereotype to any message between lifeline 4 and 2.
19. Apply the ‘UpdateMessage’ stereotype any message between lifeline 2 and 3.

Open the ‘.umlseq’ file to see the diagram. Nudge the elements to get a more appealing visualization of the behavior of the MVC pattern.

Figure 14 shows the sequence diagram modeling the behavior of the library application when registering a new user.

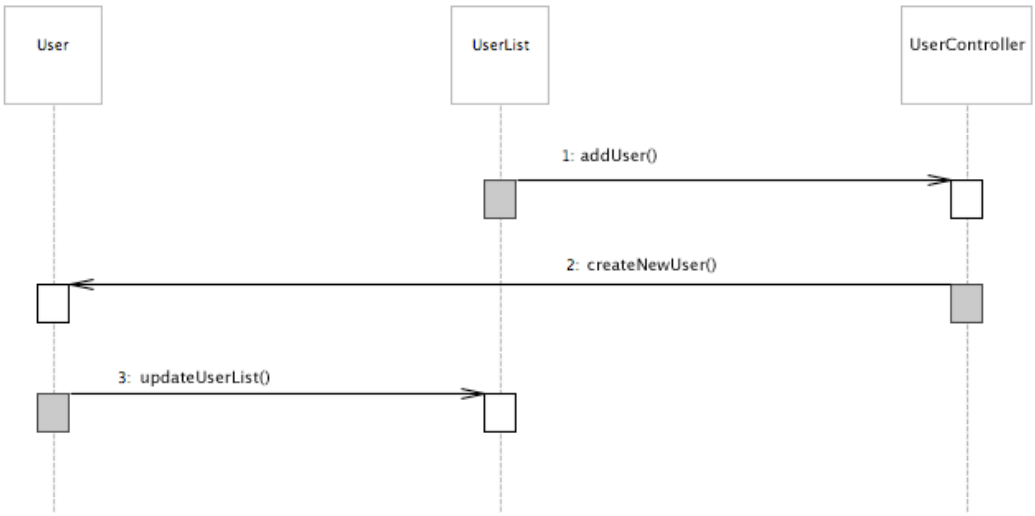

Figure 14: This diagram describes the behavior of the example library application when a new user record is added. First the view commands the controller to add the specified new user. Then the controller asks the model to create a new user record. Finally the model updates the user list with the new data.

Repeat the whole process for creating a sequence diagram specifying the applications behavior while registering the loan and return of a library book. Start with adding a ‘Book’, a ‘BookList’, a ‘BookStatus’ and a ‘BookController’ lifeline and make sure they are stereotyped ‘ModelInstance’, ‘ViewInstance’, ‘ViewInstance’, and ‘ControllerInstance’ respectively. Then add a message ‘lendBook’ stereotyped ‘CommandMessage’ between the ‘BookList’ and the ‘BookController’ lifeline, add a message ‘registerLoan’ stereotyped ‘DataMessage’ between the ‘BookController’ and the ‘Book’ lifeline and add a message ‘updateBookList’ stereotyped ‘UpdateMessage’ between the ‘Book’ and the ‘BookList’ lifeline to model the behavior when lending a book. Add a message ‘returnBook’ stereotyped ‘CommandMessage’ between the ‘BookStatus’ and the ‘BookController’ lifeline, add a message ‘registerReturn’ stereotyped ‘DataMessage’ between the ‘BookController’ and the ‘Book’ lifeline and add a message ‘updateBookList’ stereotyped ‘UpdateMessage’ between the ‘Book’ and the ‘BookStatus’ lifeline to model the behavior when a book is returned.
Figure 15 shows the sequence diagram modeling the behavior of the library application when a book is lent to someone and when a book is returned to the library.

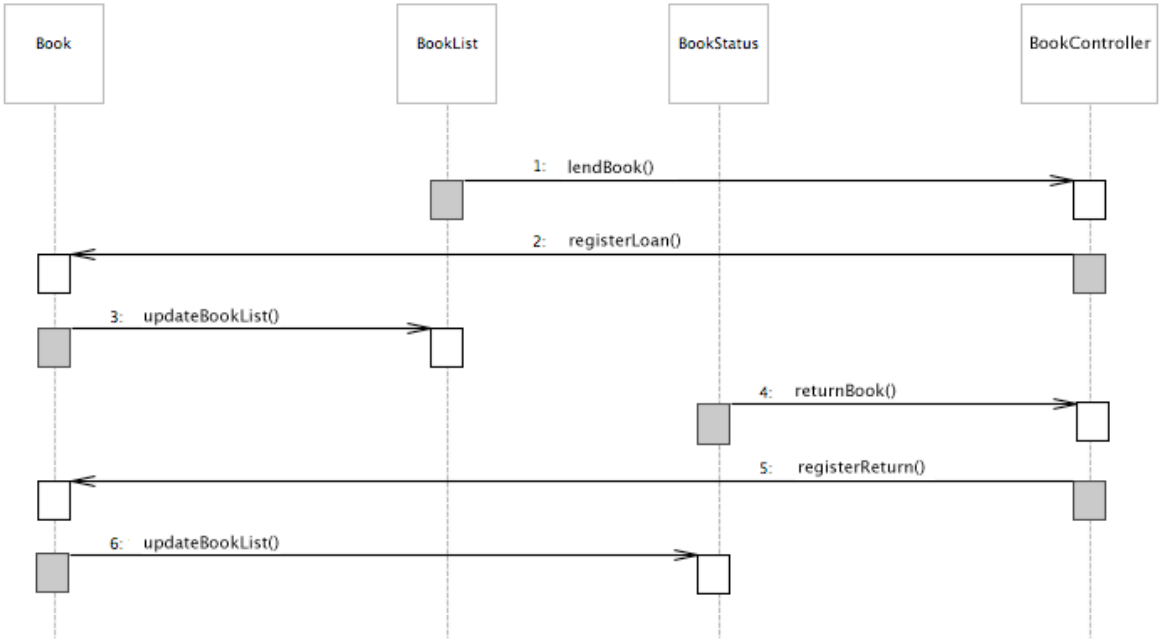

Figure 15: This diagram describes the behavior of the example library application when a book is lent to someone and when a book is returned to the library. When a book is lent, the view containing the book list first commands the controller to lend the book out. Then the controller asks the model to register the book as lent. Finally the model updates the view. When a book is returned, the view containing the book status commands the controller to return the book. The controller then asks the model to register the book as returned. Finally the view updates the view so that is reflects the books new status.

At this point the UML sequence diagrams are ready for code generation.

4.4.1.5 Code Generation for MVC

The process of code generation requires us to create a TagTree containing all UML elements making part of the MVC pattern and a specification of their implementation. The component diagram and the sequence diagram have different contributions to the TagTree and we will discuss them separately.

The UML component diagram defining the structure of the MVC pattern adds nodes for the model, view and controller UML component to the TagTree. Also for every UML interface defining the communication between these components, a node is created in the tree. The implementation of the UML interface is specified as a Java interface. The UML components are implemented as abstract Java classes implementing the interfaces the component provides. The abstract classes and interfaces will define the structure of the source code. Figure 16 illustrates the relationship between the UML component diagram for the MVC pattern and its implementation.

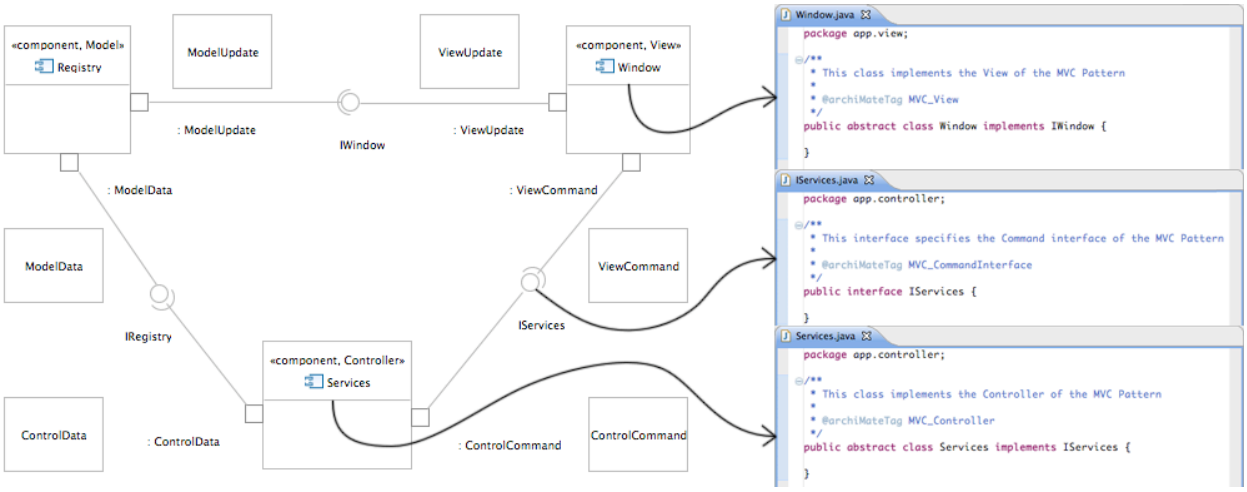

Figure 16: This figure illustrates the relationship between the UML component diagram modeling the MVC patterns structure and its implementation in Java source code. The 'IServices' UML interface is implemented as a Java interface. The 'Services' UML component is implemented as an abstract Java class implementing the 'IServices' interface. The 'Window' UML component finally is implemented as an abstract Java class.

The sequence diagram representing the behavior of the MVC pattern contributes a node for every distinct UML lifeline representing one of the MVC components in the component diagram. Their implementation specification contains a Java interface and a Java class implementing this interface. The Java class extends the abstract class added for the component the lifeline represents. The Java interface extends the interface implemented by this abstract class. For every distinct message in the sequence diagram representing the interaction between the components of the MVC pattern, a method is added to the class associated with the receiving lifeline, a method declaration is added to the interface this class implements and a method invocation is added to the class associated with the sending lifeline. The source code added by the sequence diagram really implements the functionality and thus the behavior of the software system.

Figure 17 illustrates the relationship between the UML sequence diagram modeling the MVC patterns behavior and its implementation.

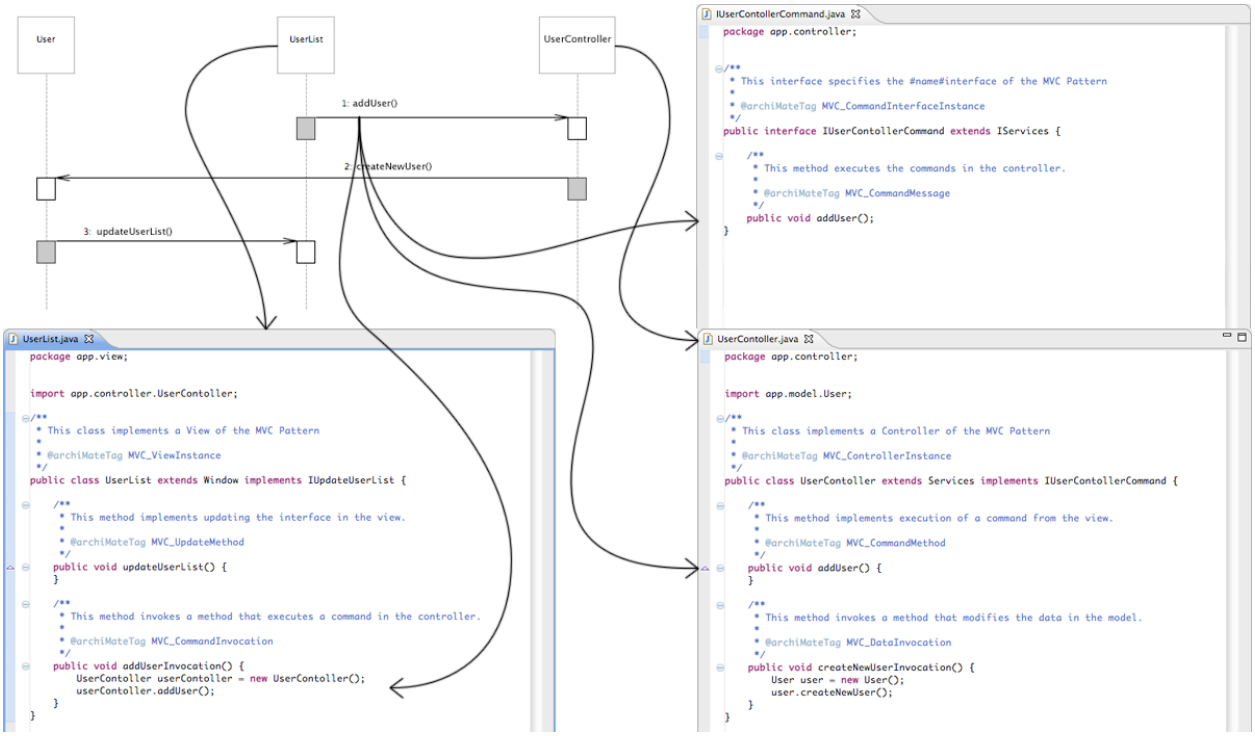

Figure 17: This figure illustrates the relationship between the UML sequence diagram modeling the MVC patterns behavior and its implementation. The ‘UserList’ UML lifeline is implemented as a Java class extending the ‘Windows’ abstract Java class. The ‘UserController’ lifeline is implemented as a Java class extending the Services abstract Java class implementing the ‘IUserControllerCommand’ interface, which in turn extends the ‘IServices’ interface. The ‘addUser’ UML message is implemented as a method declaration in the ‘IUserControllerCommand’ interface, a method implementation in the ‘UserController’ Java class and a method invocation in the ‘UserList’ Java class.

When the code generation is complete for the component diagram as well as the sequence diagram, we end up with three packages: ‘app.model’, ‘app.view’ and ‘app.controller’.

The model package contains an abstract class ‘Registry’ representing the model of the MVC pattern and an interface ‘IRegistry’ defining the interaction between the controller and the model. Furthermore it contains the model instance classes ‘User’ and ‘Book’ and the interfaces ‘IUserData’ and ‘IBookData’ defining the interaction between the controller instances and these model instances. The model classes implement the methods declared in the interfaces they implement and the controller instances invoke the methods of the model interfaces they use. Figure 18 shows the source files in the ‘app.model’ package.

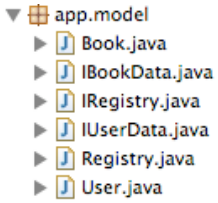

Figure 18: The source files in the ‘app.model’ package.

The controller package contains an abstract class ‘Services’ representing the controller of the MVC pattern and an interface ‘IServices’ specifying the interaction between the view and the controller. It also contains the controller instance classes ‘UserController’ and ‘BookController’ and the interfaces ‘IUserControllerCommand’ and ‘IBookControllerCommand’ defining the interaction between the view instances and the controller instances. The controller classes implement the methods declared in the interfaces they implement and the view instances invoke the methods of the controller interfaces they use. Figure 19 displays the source files in the ‘app.controller’ package.

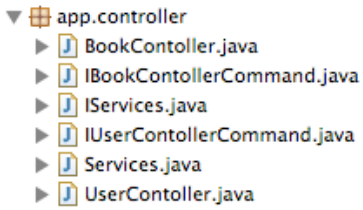

Figure 19: The source files in the ‘app.controller’ package.

The view package finally contains an abstract class ‘Window’ representing the view of the MVC pattern and an interface ‘IWindow’ defining the interaction between the model and the view. The package also contains the view instance classes ‘UserList’, ‘Booklist’ and ‘BookStatus’ and the interfaces ‘IUpdateUserList’, ‘IUpdateBooklist’ and ‘IUpdateBookStatus’ specifying the interaction between the model instances and the view instances. The view classes implement the methods declared in the interfaces they implement and the model instances invoke the methods of the view interfaces they use. Figure 20 shows the source files in the ‘app.view’ package.

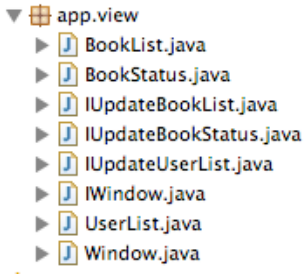

Figure 20: The source files in the ‘app.view’ package.

4.4.1.6 Code Validation for MVC

The validation of the MVC patterns implementation starts with verifying the presence of every code element specified in the TagTree. Whenever a source code element matches the tag and the identifying attributes of the implementation specification in the tree, the implementation is marked as present. Because a tree comparison process is used, not only the presence of a code element is verified, but also its position in the code hierarchy. For instance, the 'addUser' method implementation will not be marked as present unless it is found in the 'UserController' Java class in the 'app.controller' package. Figure 21 illustrates the errors the ArchiMate plugin reports when source code elements are missing.

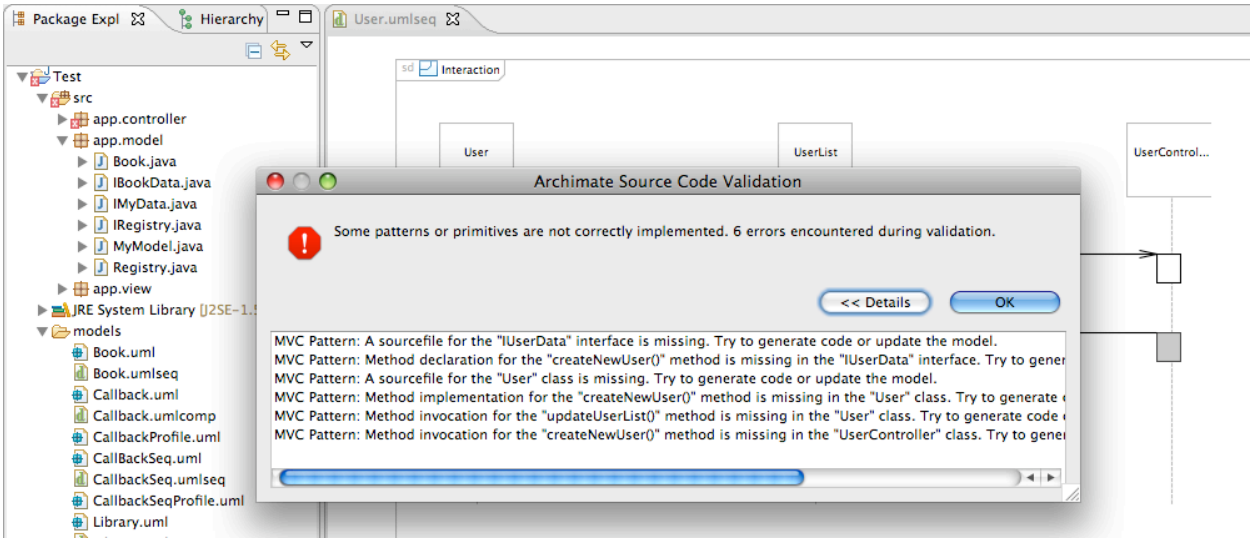

Figure 21: This figure illustrates the errors the ArchiMate plugin reports when source code elements are missing. In this example the 'User' Java class and the 'IUserData' Java interface and their content are missing.

In a second step, the soundness of the found source code elements is checked. The implemented interfaces of every Java class are verified. For instance when the ‘UserController’ Java class is found, the ArchiMate plugin checks whether it implements the ‘IUserControllerCommand’ interface. Figure 22 shows the warning reported by the ArchiMate plugin when a Java class doesn’t implement the right interface.

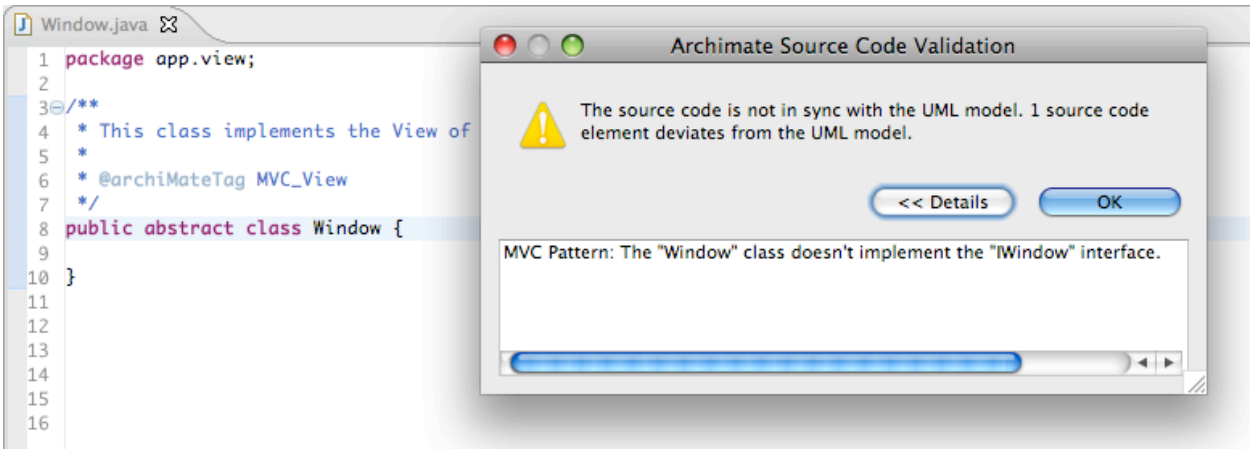

Figure 22: This figure shows the warning reported by the ArchiMate plugin when a Java class doesn’t implement the right interface. In the example the ‘Window’ Java class doesn’t implement the ‘IWindow’ interface.

Furthermore the type of every method is checked. When the method implements a method invocation, the ArchiMate plugin verifies that the method from the right interface is invoked. This means for instance that the method invoked by the ‘addUserInvocation’ method in the ‘UserList’ Java class really has to invoke the ‘addUser’ method declared in the ‘IUserControllerCommand’ interface in the ‘app.controller’ package in order to be valid.

Figure 23 displays the error message reported by the ArchiMate plugin when a method invocation didn't pass validation.

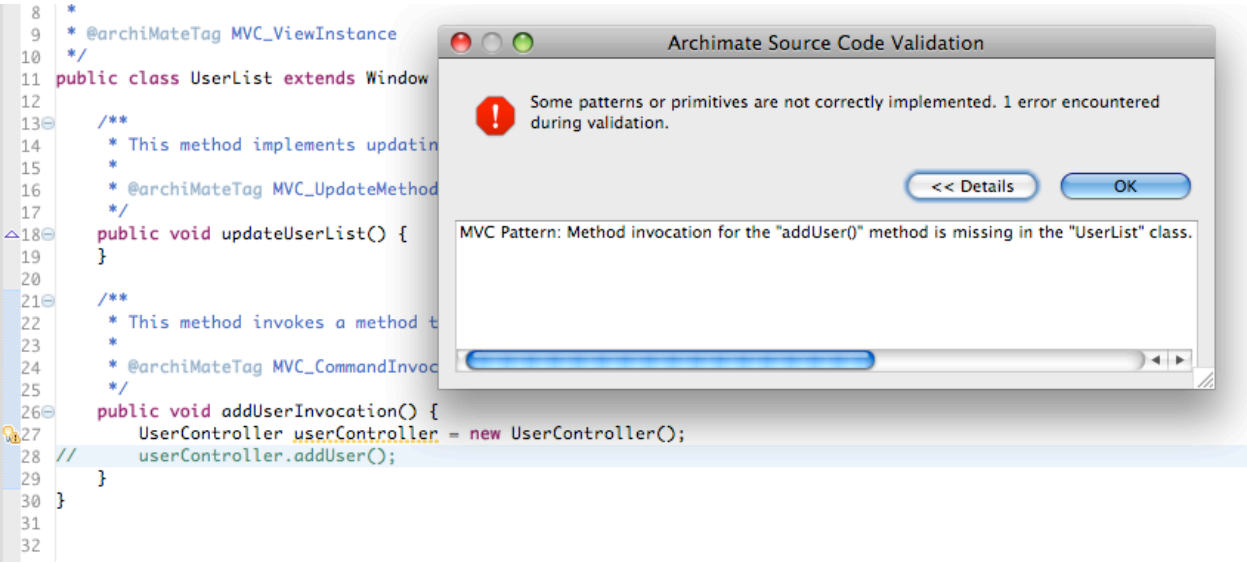

Figure 24: This figure illustrates the error reported by the ArchiMate plugin when a method invocation didn't pass validation. In the example the 'addUserInvocation' method doesn't invoke the 'addUser' method of the 'UserController' Java class.

Finally the ArchiMate plugin also checks for undesired code elements in the source. For every class in the source code the implemented interfaces are checked to make sure that the interfaces belonging to the MVC pattern are only implemented by classes that are intended to implement these interfaces. For instance, when the 'UserList' Java class implements the 'IUserControllerCommand', it implements methods updating the data of the application. This is a violation of the MVC pattern, because only the model is supposed to manage the data of the application. The ArchiMate plugin reports an error when this violation is encountered. Figure 24 shows the error reported when a restricted interface is implemented by a Java class that is not supposed to do so.

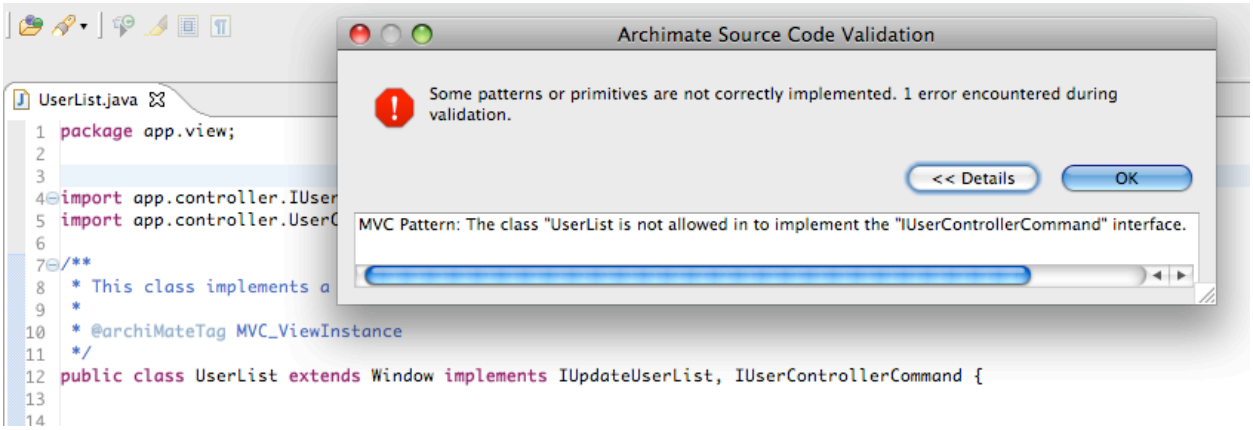

Figure 23: This figure shows the error reported when a restricted interface is implemented by a Java class that is not supposed to do so. In the example the 'UserList' Java class implements the 'IUserControllerCommand' interface.

Similarly every method invocation is checked to make sure that the methods implementing the interaction between the MVC pattern components are only invoked from classes that are meant to do so.
For instance, when the 'UserList' class invokes the 'createNewUser' method from the 'User' class, the view directly asks the model to modify the data, thus bypassing the controller. This is a violation of the MVC pattern, as all requests from the view should be handled by the controller. For this kind of violation the ArchiMate plugin reports an error. Figure 25 illustrates the error the ArchiMate plugin reports when a method is invoked from a class that is not supposed to do so.

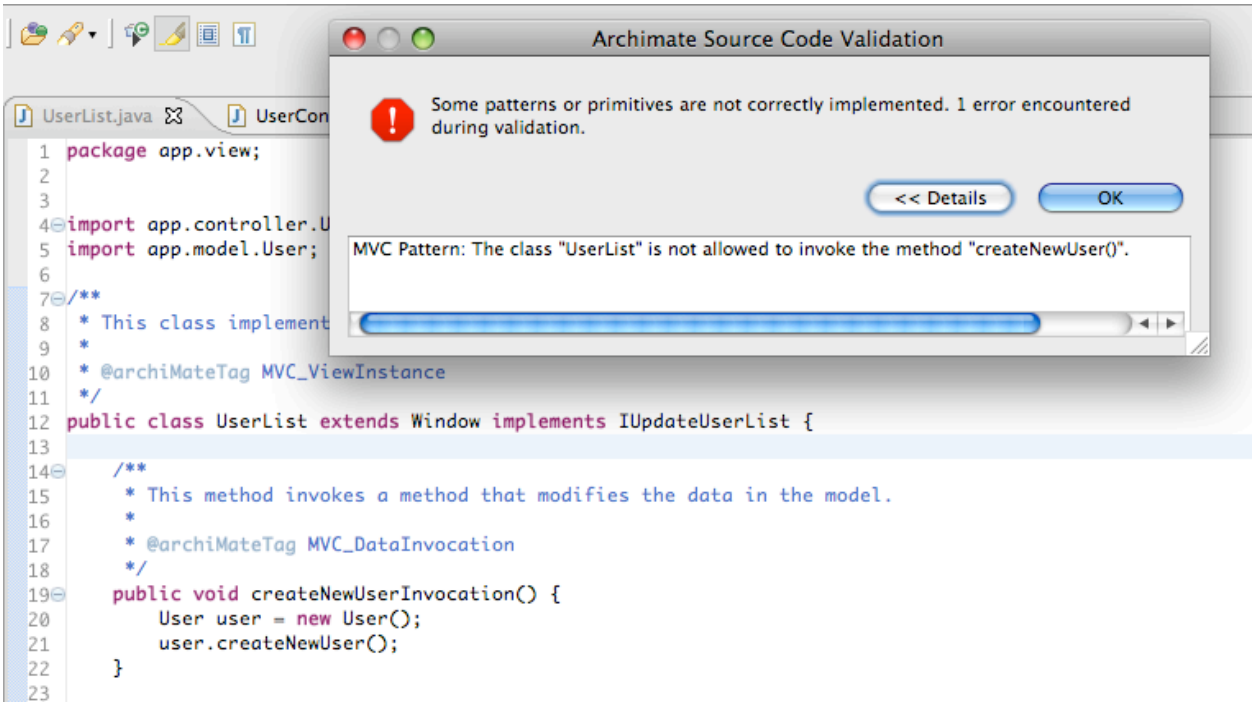

Figure 25: This figure illustrates the error the ArchiMate plugin reports when a method is invoked from a class that is not supposed to do so. In this example the 'UserList' Java class invokes the 'createNewUser' method from the 'User' Java class.

4.4.2 Callback

In this section the code generation and validation process for the callback primitive is discussed. First a UML profile is created defining the structure of the primitive. Then a UML component diagram is build modeling this structure, to which the profile is then applied. The same steps are followed for defining the behavior of the callback primitive. The diagrams created are further used in the discussion of code generation and validation.

4.4.2.1 Defining Profile for Callback Component Diagram: UML M2 Level

The structure of the callback primitive consists of two components, the caller and the callee, and two interfaces defining the interaction between these components, the subscriptioninterface and the eventinterface. Those elements will be represented in a UML profile by stereotypes.

Below the steps for creating a profile defining the structure of the callback primitive are summarized. For more detailed instructions we refer to the MVC pattern use case. Note that the profile and the stereotypes need to be named correctly for the ArchiMate plugin to work.

1. Name the profile ‘Callback’.
2. Create an element import for the UML element component.
3. Create an element import for the UML element interface.
4. Create a stereotype named ‘Caller’ and create an extension of the UML element component for it.
5. Create a stereotype named ‘Callee’ and create an extension of the UML element component for it.
6. Create a stereotype named ‘SubscriptionInterface’ and create an extension of the UML element interface for it.
7. Create a stereotype named ‘SubscriptionInterface’ and create an extension of the UML element interface for it.
8. Define the profile.

The profile is now ready to by applied to a component diagram, identifying the UML elements that model the callback primitive. Figure 26 displays the created profile.

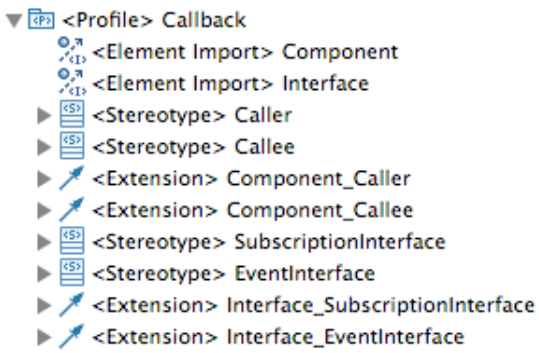

Figure 26: This figure illustrates the UML profile defining the structure of the callback primitive.

4.4.2.2 Callback Component Diagram: UML M1 Level

Once the UML profile defining the structure of the callback primitive has been defined, this structure can be modeled in a UML component diagram. The diagram contains the caller and callee components of the primitive, and the interfaces defining the interaction between the components.

Below the steps for creating this component diagram modeling the structure of the callback primitive are lined out. For more detailed instructions we refer to the MVC pattern use case.

1. Add a component for the caller to the package and give it the name ‘Caller’.
2. Add a component for the callee to the package and give it the name ‘Callee’.
3. Add an interface for the communication from the caller to the callee component and give it the name ‘ISubscription’.
4. Add an interface for the communication from the callee to the caller component and give it the name ‘IEvent’.
5. Add a class for the communication between from the caller to the callee implementing the interface from 3.
6. Add an interface implementation to 5 and set its contract to 3.
7. Add a usage and set its client to 5 and its supplier to 4.
8. Add a class for the communication between from the callee to the caller implementing the interface from 4.
9. Add an interface implementation to 8 and set its contract to 4.
10. Add a usage and set its client to 8 and its supplier to 3.
11. Add a port to 1 and type it by 8.
12. Add a port to 2 and type it by 5.

When all steps are carried out correctly, the callback primitive is successfully modeled in the component diagram, and the profile defined in the previous section can be applied to it in order to identify the callback elements of interest for code generation:

13. Apply the ‘Callback’ profile to the package.
14. Apply the ‘Caller’ stereotype to the component 1.
15. Apply the ‘Callee’ stereotype to the component 2.
16. Apply the ‘SubscriptionInterface’ stereotype to the interface 3.
17. Apply the ‘EventInterface’ stereotype to the interface 4.

Now the UML component diagram is ready for code generation. The diagram is displayed in figure 27.

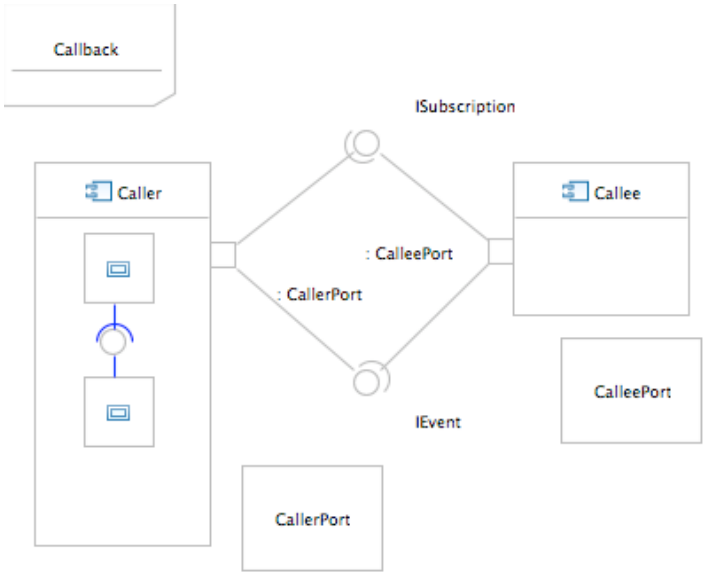

Figure 27: This UML component diagram represents the structure of the callback primitive.

4.4.2.3 Defining Profile for Callback Sequence Diagram: UML M2 Level

The UML profile defining the behavior of the callback primitive contains the lifelines representing the primitives components and the messages between these lifelines representing the communication between these components.

Below the steps to create this profile defining the behavior of the callback primitive are summarized. For more detailed instructions we refer to the MVC pattern use case. Note that the profile and the stereotypes have to be named correctly for the ArchiMate plugin to work.

1. Name the profile ‘Callback’.
2. Create an element import for the UML element lifeline.
3. Create an element import for the UML element message.
4. Create a stereotype named ‘CallerInstance’ and create an extension of the UML element lifeline for it.
5. Create a stereotype named ‘CalleeInstance’ and create an extension of the UML element lifeline for it.
6. Create a stereotype named ‘SubscriptionMessage’ and create an extension of the UML element message for it.
7. Create a stereotype named ‘EventMessage’ and create an extension of the UML element message for it.
8. Define the profile.

When all steps are carried out, the profile is ready to be applied to a UML sequence diagram. It then will identify the UML elements making part of the callback primitive.

4.4.2.4 Callback Sequence Diagram: UML M1 Level

The behavior of the callback primitive can modeled in a UML sequence diagram by representing the components by lifelines and the interaction between these components by the messages that connect the lifelines.

Below the steps to create a sequence diagram representing the behavior of the callback primitive are summed up. For more detailed instructions we refer to the MVC pattern use case.

1. Add an interaction to the UML package.
2. Add a lifeline representing the caller component and give it the name ‘ACaller’.
3. Add a lifeline representing the callee component and give it the name ‘ACallee’.

The following steps describe how to add a message from the caller lifeline to the callee lifeline representing the ‘subscribeToEvent’ method invocation:

4. Add a message occurrence specification for the start of the method invocation and set the covered attribute to 2.
5. Add a message occurrence specification for the start of the method execution and set the covered attribute to 3.
6. Add a message occurrence specification for the end of the method execution and set the covered attribute to 3.
7. Add a message occurrence specification for the end of the method invocation and set the covered attribute to 2.
8. Add a behavior execution specification for the invocation and set its covered attribute to 2, its start to 4 and its end to 7.
9. Add a behavior execution specification for the execution and set its covered attribute to 3, its start to 5 and its end to 6.
10. Add a message representing the method call, give it the name ‘subscribeToEvent’ and set its send event to 4 and its receive event to 5.
11. Set the message attribute of 4 and 5 to 10.

Repeat the steps from 4 to 11 for adding a message from the callee lifeline to the caller lifeline representing the ‘handleEvent’ method invocation.

When all steps are carried out, the model represents the behavior of the callback primitive. Now the profile defined in the previous section can be applied in order to identify the UML elements used in code generation:

12. Apply the ‘Callback’ profile to the package.
13. Apply the ‘CallerInstance’ stereotype to the lifeline from 2.
14. Apply the ‘CalleeInstance’ stereotype to the lifeline from 3.
15. Apply the ‘SubscriptionMessage’ stereotype to the message representing the ‘subscribeToEvent’ method invocation (10).
16. Apply the ‘EventMessage’ stereotype to the message representing the ‘handleEvent’ method invocation.

Now the UML sequence diagram is ready for code generation. The diagram is displayed in figure 28.

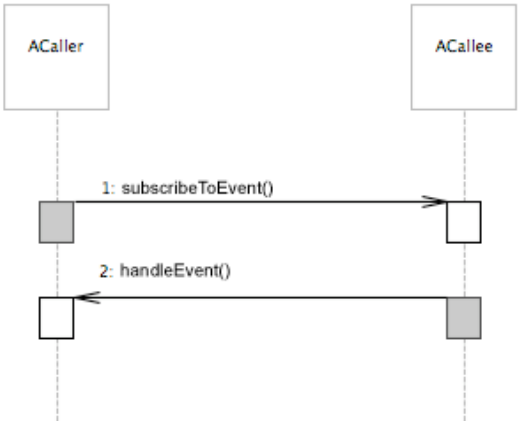

Figure 28: This UML sequence diagram represents the behavior of the callback primitive.

4.4.2.5 Code Generation for Callback

When it comes to code generation, we have to create a TagTree containing all UML elements that represent the Callback primitive in the component diagram and the sequence diagram.

The UML component diagram defining the structure of the callback primitive adds nodes for the caller and callee component to the tree. Nodes are also added for the UML interfaces defining the communication between the caller and the callee. The UML interfaces are implemented as Java interfaces and the components are implemented as abstract Java classes implementing these interfaces. Together, these classes and interfaces define the structure of the implementation. The relationship between the component diagram and the Java source code is further illustrated in figure 29.

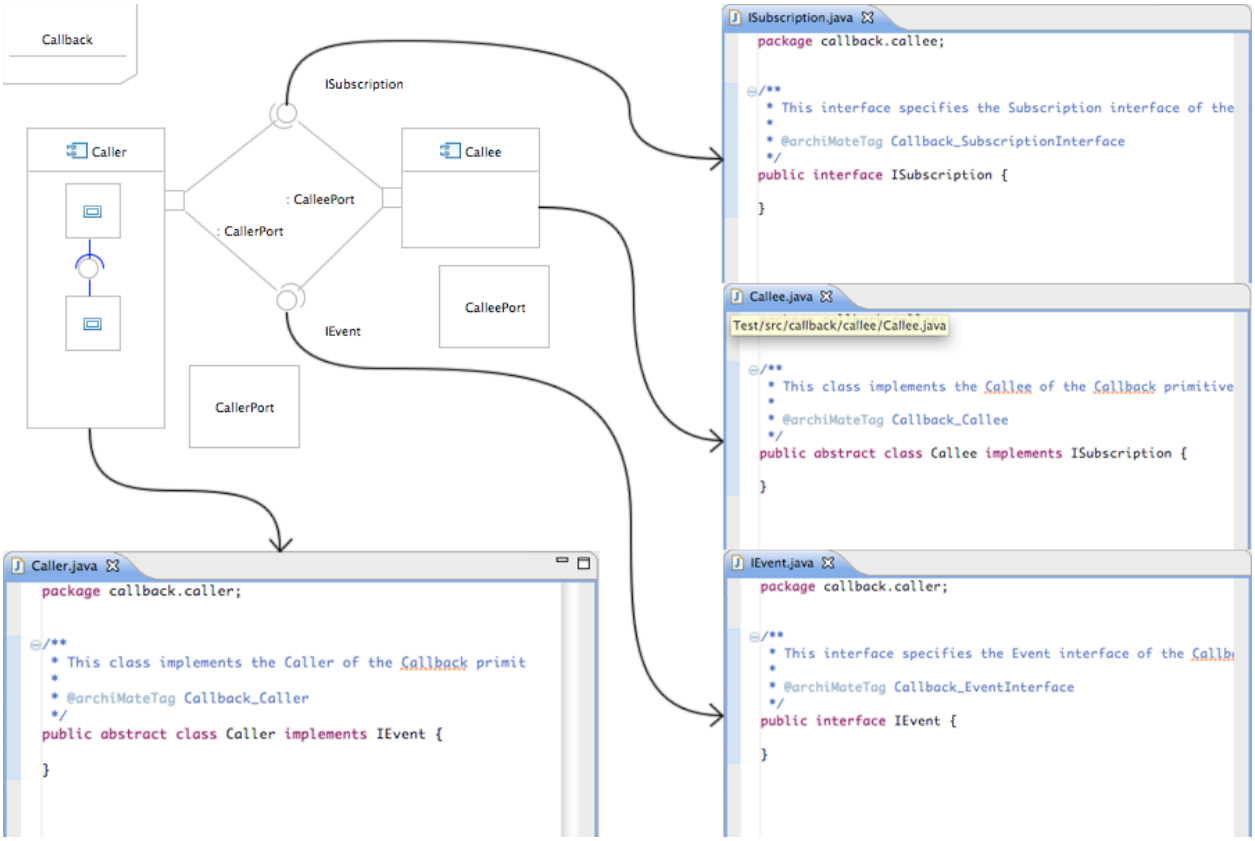

Figure 29: This figure represents the relationship between UML component diagram modeling the callback primitive and its implementation in Java code.

The sequence diagram representing the behavior of the callback primitive contributes a node to the TagTree for every distinct UML lifeline representing either the caller or the callee of the callback primitive. The lifelines are implemented through a Java interface and a Java class implementing this interface. The Java class extends the abstract class related to the component the lifeline represents. The Java interface extends the interface implemented by this abstract class.

Every distinct message in the sequence diagram representing the communication from the caller to the

callee adds three code elements to the tree. A method, implementing the subscription of the caller to an event, is added to the Java class for the lifeline representing the callee. A method declaration for this method is added to the Java interface this class implements. Finally a method invoking this subscription is added to the Java class for the lifeline representing the caller. The relationship between the sequence diagram representing the callback primitive and the code generated for messages from the caller to the callee is illustrated in figure 30.

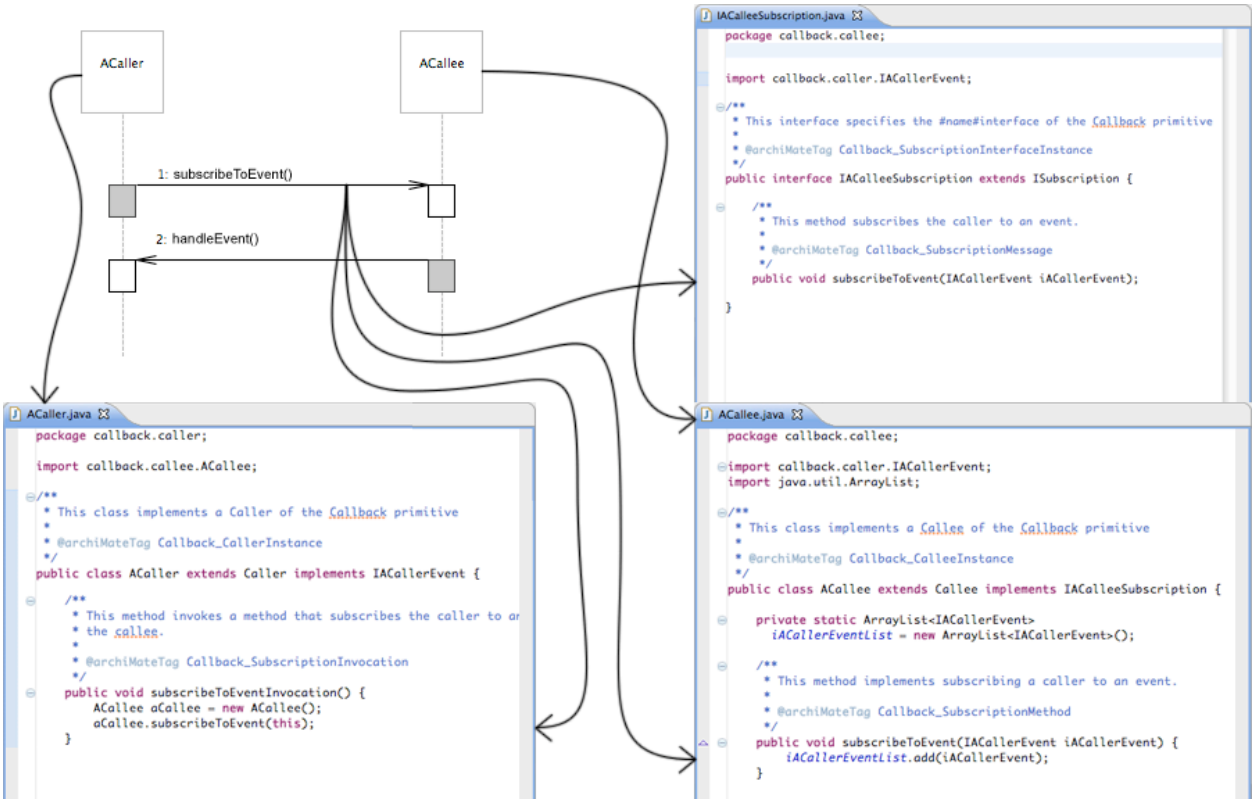

Figure 30: This figure illustrates the relationship between a UML message send from the caller to the callee and its implementation. For every such message a method subscribing the caller to an event at the callee is added to the Java class implementing the callee. The methods declaration is added to the interface this class implements. And finally a method invoking the subscription method is added to the Java class implementing the caller.

Similarly every distinct message in the sequence diagram representing the communication from the callee to the caller adds three code elements to the tree. A method handling the occurred event is added to the Java class for the lifeline representing the caller. A method declaration for this method is added to the Java interface this class implements. Finally a method doing the callback to all callers subscribed to the event is added to the Java class for the lifeline representing the callee.

The relationship between the sequence diagram representing the callback primitive and the code generated for messages from the callee to the caller is illustrated in figure 31.

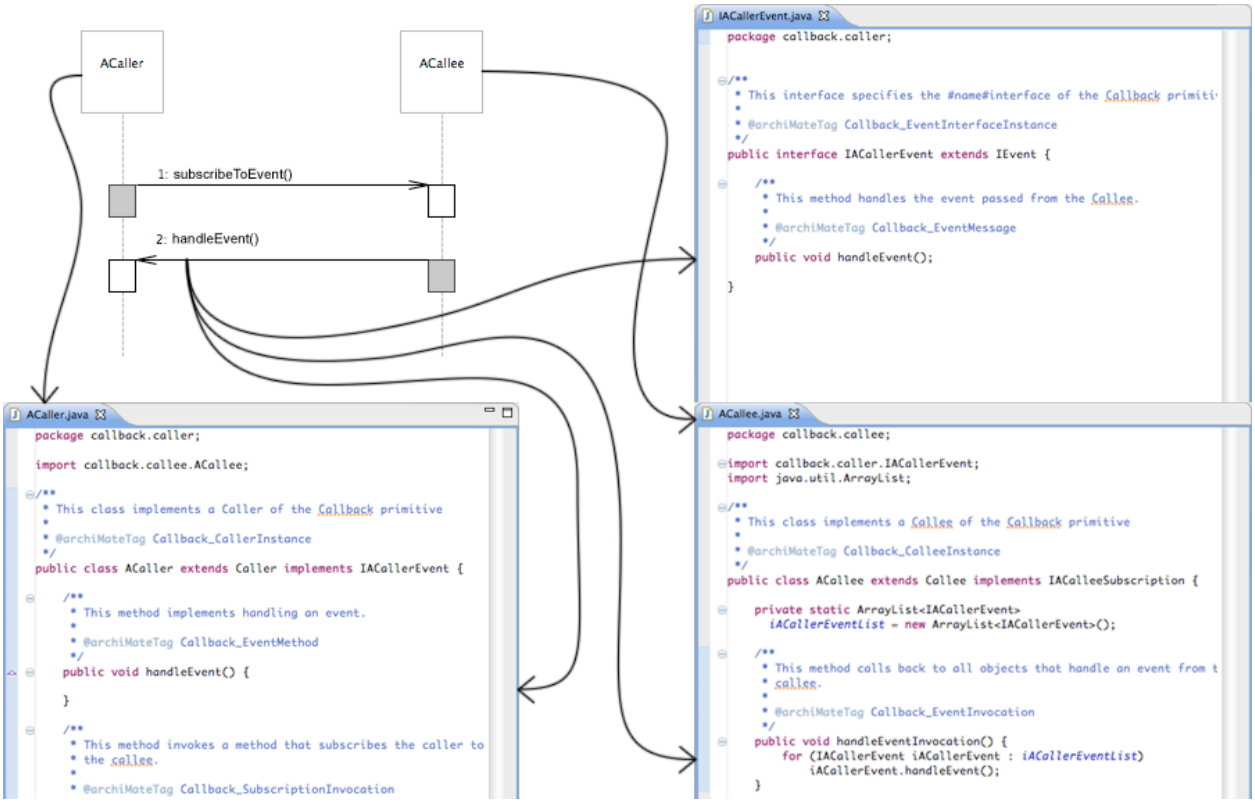

Figure 31: This figure illustrates the relationship between a UML message send from the callee to the caller and its implementation. For every such message a method handling the occurred event is added to the Java class implementing the caller. The methods declaration is added to the interface this class implements. And finally a method doing the callback to all callers subscribed to the event is added to the Java class implementing the caller.

When code generation is complete for the component diagram as well as the sequence diagram, we end up with two packages: 'callback.caller' and 'callback.callee'.

The caller package contains an abstract class 'Caller', representing the caller of the callback primitive, and an interface IEvent, representing the communication from the callee to the caller. Furthermore it contains the caller instance classes, like the 'ACaller' class, and the interfaces defining the communication from the callee instances to the caller instances, like the 'IACallerEvent' interface. The caller instance classes implement the event handling methods declared in the interfaces they implement and invoke the subscription methods declared in the interfaces they use. Figure 32 shows the source files in the 'callback.caller' package.

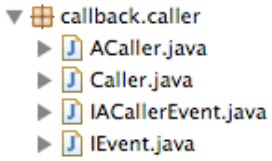

Figure 32: This figure shows the source files in the 'callback.caller' package

The callee package contains an abstract class called ‘Callee’, representing the callee of the callback primitive, and an interface ‘ISubscription’, representing the communication from the caller to the callee. Furthermore it contains the callee instance classes, like the ‘ACallee’ class, and the interfaces defining the communication from the caller instances to the callee instances, like the ‘ACalleeSubscription’ interface. The callee instance classes implement the subscriptions methods declared in the interfaces they implement and invoke the event handling methods declared in the interfaces they use for every caller subscribed to the specific event. Figure 33 shows the source files in the ‘callback.callee’ package.

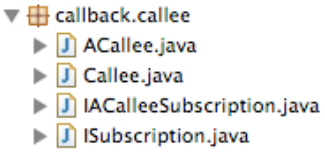

Figure 33: This figure shows the source files in the ‘callback.callee’ package

4.4.2.6 Code Validation for Callback

The validation of the Callback primitive checks in the first place the presence and hierarchical ordering of every code element defined in the TagTree. For every missing element or element which was not encountered at the right place in the hierarchy an error is reported by the ArchiMate plugin.
Secondly, the soundness of every code element identified as implementing the callback is verified. More specifically for every Java class the implemented interfaces are checked. Furthermore the type of every method is checked. When a method is meant to implement a method invocation, the ArchiMate plugin verifies that it really invokes the method from the right interface. For every deviation from the intended implementation, an error is reported to the user.
Finally the ArchiMate plugin searches for undesired code elements in the source. For every Java class in the source code the implemented interfaces are checked to make sure that none of the interfaces implementing the callback primitive are implemented where they weren’t meant to. For example when a caller instance implements the interface meant for the callee instance, it could send a callback to itself, which is a violation of the callback primitive. Similarly, every method invocation in the source code is checked to make sure that methods implementing the callback primitive are invoked only from classes that are meant to do so. For example when a callee instance would invoke a subscription method that it self implements, it would be able to subscribe to its own events, which is a violation of the callback primitive. For every violation of the callback primitive, the ArchiMate plugin reports an error.

Figure 34 illustrates the errors reported by the ArchiMate plugin when validating the callback primitive.

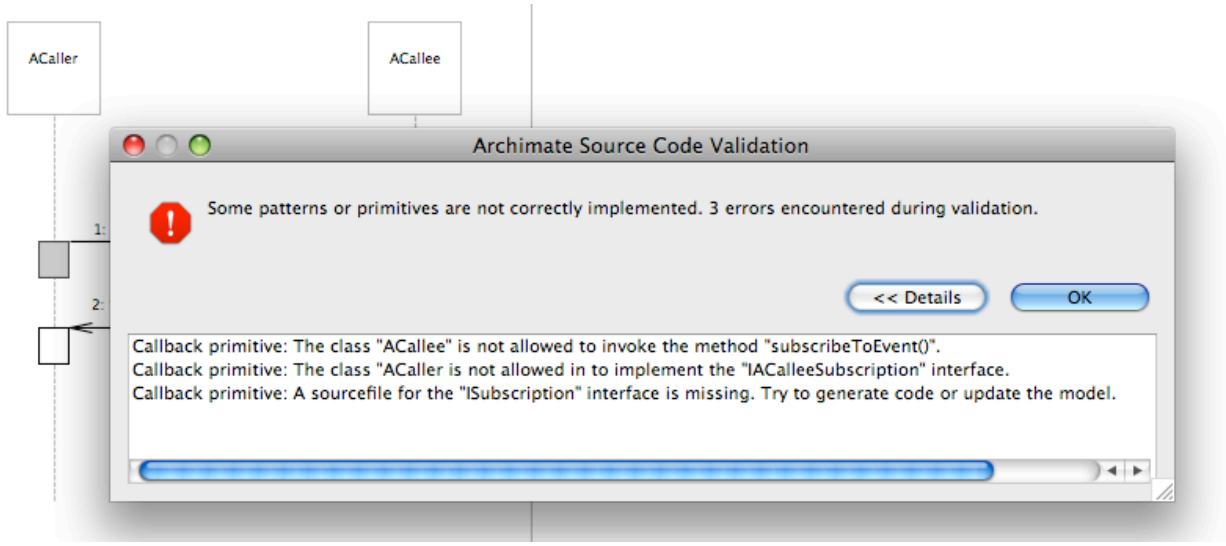

Figure 34: This figure illustrates the errors reported by the ArchiMate plugin when validating the callback primitive. In the example, the callee instance ‘ACallee’ subscribes itself to its own event, the caller instance ‘ACaller’ implements the interface of the callee instance and the superinterface ‘ISubscription’ is missing.

4.4.3 MVC & Callback

In this section the use of the callback primitive inside the MVC pattern is discussed as a use case. First a UML component diagram is created representing the pattern and primitives structure. Then a UML sequence diagram is built modeling the behavior of both. Finally the process of code generation and validation are discussed.

4.4.3.1 MVC & Callback Component Diagram: UML M1 Level

To create the component diagram for the integration of a callback primitive in the MVC pattern, we start with the component diagram created in section 4.4.1.2 of the MVC pattern use case modeling the structure of a library application. The callback primitive components are then integrated at the right place in this diagram.

Below the steps used to modify the MVC component diagram so that it contains a callback primitive for the communication between the model and the view are summed up.

1. Add an interface named ‘IRegistryEvent’ for the communication from the callee to the caller component.
2. Add a usage and set its client to the existing ‘ViewUpdate’ class and its supplier to 1.
3. Add an interface implementation to the existing ‘ModelUpdate’ class and set its contract to 3.

Now the structure of MVC pattern and a callback are correctly represented in the UML diagram and the profile defining the structure of the callback primitive can be applied to identify the callback UML elements of interest for code generation.

4. Apply the ‘Callback’ profile to the package.
5. Apply the ‘Caller’ stereotype to the view component.
6. Apply the ‘Callee’ stereotype to the model component.
7. Apply the ‘SubscriptionInterface’ stereotype to the interface from 1.
8. Apply the ‘EventInterface’ stereotype to the updateinterface.

Repeat the steps from above to use the callback primitive for the communication between any other pair of MVC components.

Now the UML component diagram is ready for code generation. The diagram is displayed in figure 35.

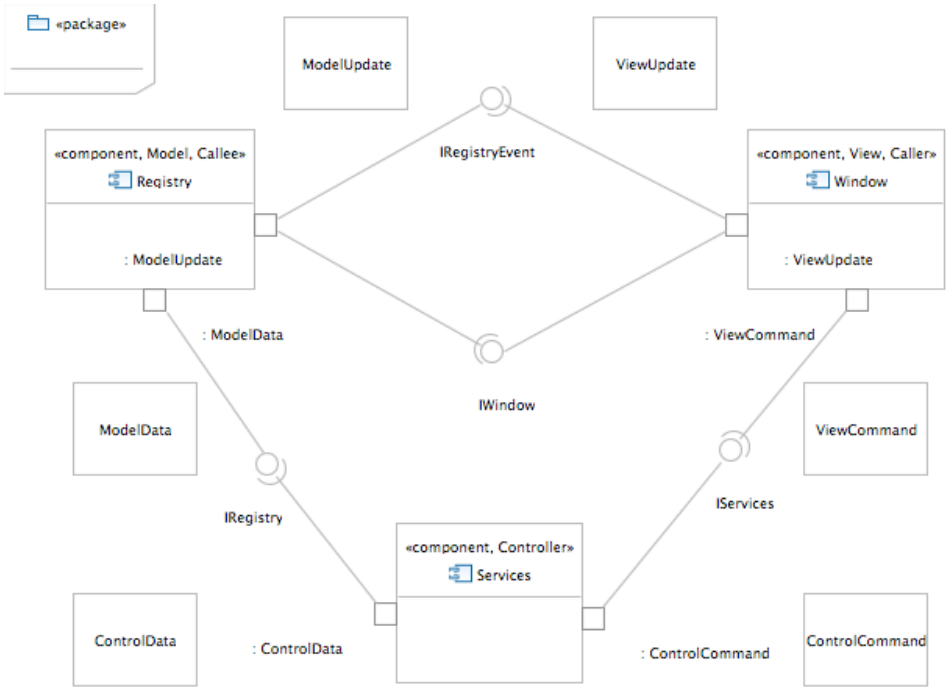

Figure 35: This UML component diagram represents the structure of an MVC pattern featuring a callback primitive for the communication between the model and the view.

4.4.3.2 MVC & Callback Sequence Diagram: UML M1 Level

To model the behavior of the MVC pattern using a callback primitive, we start with the sequence diagram created in the section 4.4.1.4 of the MVC pattern use case modeling the behavior of the library application when a new user is registered. The callback UML elements are then integrated at the right place in this sequence diagram.

Below the steps taken to modify the MVC sequence diagram so that it contains a callback primitive for the communication between a model instance and a view instance are summed up.

1. Add a message occurrence specification for the start of a method invocation and set the covered attribute to the lifeline representing the view instance.
2. Add a message occurrence specification for the start of a method execution and set the covered attribute to the lifeline representing the model instance.
3. Add a message occurrence specification for the end of the method execution and set the covered attribute to the lifeline representing the model instance.
4. Add a message occurrence specification for the end of the method invocation and set the covered attribute to the lifeline representing the view instance.
5. Add a behavior execution specification for the invocation and set its covered attribute to the lifeline representing the view instance, its start to 1 and its end to 4.

6. Add a behavior execution specification for the execution and set its covered attribute to the lifeline representing the model instance, its start to 2 and its end to 3.
7. Add a message representing the method call, give it the name ‘subscribeToUserCreated’ and set its send event to 1 and its receive event to 2.
8. Set the message attribute of 1 and 2 to 7.

Now the sequence diagram represents the behavior of the MVC pattern and the callback primitive. Add the profile defining the behavior of the callback primitive to identify the callback UML elements of interest for code generation.

9. Apply the ‘Callback’ profile to the package.
10. Apply the ‘CallerInstance’ stereotype to the lifeline representing the view instance.
11. Apply the ‘CalleeInstance’ stereotype to the lifeline representing the model instance.
12. Apply the ‘SubscriptionMessage’ stereotype to the message from 7.
13. Apply the ‘EventMessage’ stereotype to the updatemessage.

Repeat the steps from above to use the callback primitive for the communication between any other pair of MVC components.

Now the UML sequence diagram is ready for code generation. The diagram is displayed in figure 36.

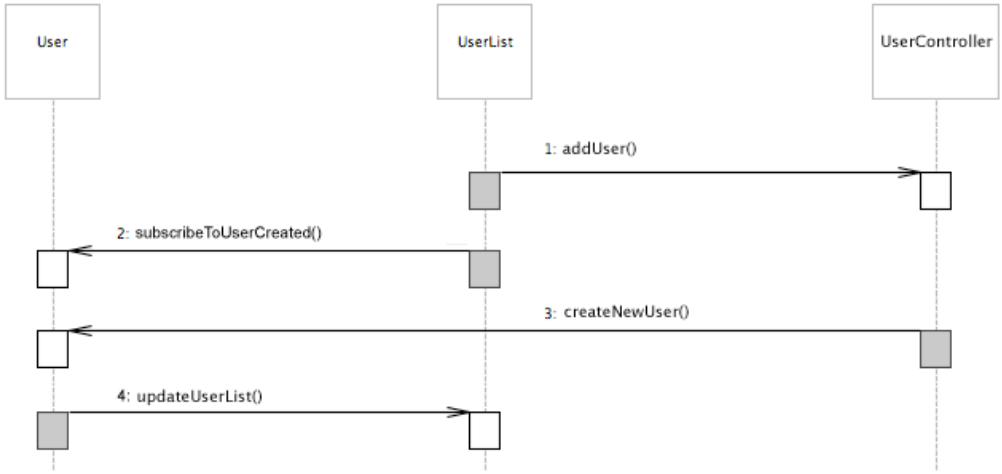

Figure 36: This UML sequence diagram represents the behavior of an MVC application sporting a callback primitive for the communication between the model and the view.

4.4.3.3 Code Generation for MVC & Callback

For generating code for the MVC pattern using a callback primitive, the ArchiMate plugin starts with the TagTree created for the MVC pattern and integrates the UML elements of the callback primitive and their implementation at the right place.

First the plugin searches the TagTree for the abstract classes representing the caller and callee from the component diagram. It adds the event interface to the caller class and the subscription interface to the callee class unless they already implement those.

Secondly it searches the TagTree for the classes representing the caller and callee instances. An event interface instance is created for every caller instance. For every eventmessage the caller instance receives, a method and its declaration is added to the caller instance and event interface instance respectively. For every subscriptionmessage the caller instance sends, a method invocation is added. A subscription interface instance is created for every callee instance. For every subscriptionmessage the callee receives, a method and its declaration is added to the callee instance and subscription interface instance respectively. For every eventmessage the callee instance sends, a method invocation is added.

When all code is generated for the UML component diagram as well as the sequence diagram, we end up with three packages: the 'app.model', 'app.view' and 'app.controller' package. From here on we assume that the callback primitive is used for the communication between the model and the view of the MVC pattern.

The 'app.controller' package contains the same classes and interfaces as in the plain MVC use case. It has undergone no changes because in this example it makes no use of the callback primitive. Figure 37 shows the source files in this package.

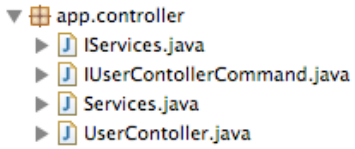

Figure 37: This figure shows the source files in the 'app.controller' package.

The 'app.view' package also contains the classes and interfaces from the plain MVC use case, except that the update interface instance is replaced by an event interface instance, and the superInterface 'IEvent' has been added. Furthermore the classes implementing the view instances now have methods subscribing them to events at the classes implementing the model instances. Figure 38 shows the source files in this package.

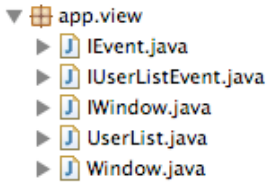

Figure 38: This figure shows the source files in the 'app.view' package.

The 'app.model' package contains the classes and interfaces from the plain MVC use case, except that the superinterface 'IRegistrySubscription' and the interface 'IUserSubscription' have been added. Furthermore the classes implementing the model instances now also have methods implementing a subscription of a view instance to an event. Figure 39 shows the source files in this package.

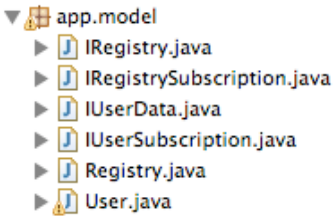

Figure 39: This figure shows the source files in the 'app.model' package.

4.4.3.4 Code Validation for MVC & Callback

The code validation for the MVC pattern using a callback primitive is similar to that for the plain MVC pattern or the callback primitive itself. First the presence and hierarchical position of the code elements is verified. Next every matched source code element is checked for its soundness. Classes have to implement the right interfaces and methods have to invoke the right methods from other classes or interfaces in order to pass validation. Finally the source code is searched for undesired elements. The use of restricted interfaces and the invocation of restricted methods are reported whenever they are undesired. For more details we refer to the MVC and callback use cases.

5. Evaluation and Results

In this bachelor project we built a UML tool for code generation: the ArchiMate Eclipse plugin. With this UML tool we meet various common problems and challenges in existing UML tools for code generation.

5.1 Architectural Structure and Behavior

To begin with, the ArchiMate plugin supports both UML component diagrams and sequence diagrams, enabling the software developer to model not only the structure of a software architecture, but also its behavior.

Based on this more complete representation of an architecture in UML, the plugin can generate significantly more and more significant code. As the use cases in this document illustrate, the structure of an architecture is translated into a limited set of classes and interfaces with almost no content, while the model of its behavior yields a rich and unbounded set of classes and interfaces with partially implemented methods and method invocations.

The structure of an architecture rarely changes during software development, once the architecture has been defined, whereas its behavior is continuously expanded when functionality is added. This shows us another advantage of taking the behavioral aspect into account: code generation now is not only relevant at the beginning of a software project, when the architecture is designed, but during the whole course of software development, as for every newly added functionality new code can be generated.

5.2 Design Patterns and Primitives

An important innovation to code generation is the explicit use of design patterns and primitives: general solutions to common occurring problems in software design.

The use of patterns and primitives in software development has two major benefits. First, problems in software engineering are solved using a proven and optimal solution. Second the communication about software architecture is made a lot easier, because the patterns and primitives are widely known. So encouraging software developers to use patterns and primitives leads to software of higher quality and eases processes like software integration and maintenance through the communication factor.

Design patterns and primitives bring advantages to the process of code generation too. For most UML elements there is no obvious or unambiguous translation to source code. However, when we know that a UML element plays a specific role in a pattern or primitive, we have a lot more information about this elements implementation. So by extending UML to represent patterns and primitives, the gap between UML and source code grows significantly smaller. We are able to generate more code and more meaningful code, because we know that every peace of generated code plays a part in the pattern or primitives implementation, and therefore is indispensible.

5.3 UML and Source Validation

A last interesting feature of the ArchiMate plugin is architectural validation. It comes in two flavors: UML model validation and source code validation.

The fact that a UML model represents a specific design pattern or primitive, gives us lots of information about what its structure should look like, what behavior it should show and how the different components should interact with each other. Using this information the deviations of a modeled architecture from the intended pattern or primitive can be localized and the likely architectural flaws identified.

The structural, behavioral and interactional constraints on a modeled architecture can be translated into constraints on the structure and functionality of a software system. By mapping these constraints to the source code, detailed feedback can be generated on the soundness of the architectures implementation.

Combining these two forms of validation, the ArchiMate plugin can be a valuable support for designing sound architectures and implementing these architectures the right way.

6. Conclusion

With this bachelor project we aspired to take the concept of code generation for UML diagrams to a challenging new level. To take it from being a convenient and timesaving trick to an indispensible tool for designing sound architectures and producing high quality code. Of course we didn't accomplish all this with the ArchiMate plugin. But we do hope to have convincingly shown that there is much more possible with code generation than most UML tools already do. May this be an inspiration to the future research on generating code for UML diagrams.

7. Future Work

There are many things to the ArchiMate plugin that can be improved or extended. In this section we would like to give some recommendations for future work on this UML tool.

7.1 Design Patterns and Primitives

More patterns and primitives can be added to the plugin in order to provide a vast library of general solutions for the problems software developers are likely to be confronted with.

7.2 Reverse Engineering

A feature that has been left partially unimplemented is the reverse engineering functionality. Currently it is only working for the tree view UML editor of Eclipse, and not for the graphical editors. Also only the process of adding messages to the sequence diagram for every new method declaration in an interface is implemented. This can be extended to adding lifelines for every new class and interface combination that implements a pattern or primitives behavior. And maybe it could be taken as far as to create a whole new sequence diagram or component diagram modeling the behavior or structure of an implemented pattern.

7.3 Integrating Sequence and Component Diagrams

The current version of the used UML2Tools framework (0.9.0) requires every UML diagram file to be based on its own UML file. Because of this, it is not possible to link the UML elements from the sequence diagram to the elements they represent in the component diagram, as one can do in other UML tools. This integration of sequence and component diagrams would make the process of code generation easier and more robust. This problem has to be further explored to see if there is a way to circumvent the limitations of the UML2Tools framework.

The ArchiMate plugin currently only takes the open UML file into account for generating and validating code. It could be interesting to take all the UML files in an Eclipse project into account when executing these processes. The benefits and drawbacks of this approach are to be explored.

7.4 Using Wizards

At the moment, the ArchiMate shows a status dialog with detailed information whenever model validation, code generation, code validation or reverse engineering is complete. It would be nice however to make the plugin more interactive using wizards. When validating the UML model or the source code, a wizard could show all found errors and offer the user one or more possible solutions for every error. The user could then select the fixes he wants to be carried out. In the case of code generation or reverse engineering, a wizard could let the user select the code or UML elements he would like to be generated.

8. Appendix

8.1 System Requirements and Installation

The ArchiMate plugin has the following system requirements:

- Java J2SE-1.5 execution environment (http://java.sun.com/javase/downloads/index_jdk5.jsp)
- Eclipse Classic 3.5 Integrated Development Environment (IDE) (http://www.eclipse.org/downloads/)

Within the Eclipse IDE, the ArchiMate plugin requires the following plugins to be installed from the MDT Update Site (http://download.eclipse.org/modeling/mdt/updates/):

- MDT OCL SDK 1.3.0
- MDT UML2 SDK 3.0.1
- MDT UML2TOOLS SDK (Incubation) 0.9.0

Installing the plugin:

- Use the console to open your eclipse workspace (`cd /path/to/your/workspace`)
- Clone the ArchiMate Git repository (`git clone git://github.com/krikis/ArchiMate.git`)
- In Eclipse, create a new general project and name it 'ArchiMate' (`File > New > Project... | General > Project`)
- Now export the ArchiMate plugin to the 'plugins' folder of your eclipse installation (`File > Export > Deployable plug-ins and fragments`)
- Restart Eclipse and the plugin is ready to use

Using the plugin with the demo UML files:

- Create a new Java project in eclipse (`File > New > Java Project`)
- Go to the Java project folder using your console (`cd NewJavaProject/`)
- Clone the models Git repository (`git clone git://github.com/krikis/models.git`)
- In Eclipse, refresh the Java projects folders and open either the UML or diagram files of the component or sequence diagrams in the newly created 'models' folder to use the plugins functionality