Division of Computer Science at the Department of Informatics and Mathematics

DEGREE PROJECT 2003:PM04

Forward Engineering from Interaction Diagrams - can it be useful?

Daniel Björklund

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

DEGREE PROJECT

University of Trollhättan · Uddevalla
Department of Informatics and Mathematics

Degree project for master degree in Software engineering

Forward Engineering from Interaction Diagrams - can it be useful?

Daniel Björklund

Examiner:
Steven Kirk Department of Informatics and Mathematics

Supervisor:
Steven Kirk Department of Informatics and Mathematics

Trollhättan, 2003
2003:PM04

EXAMENSARBETE

Forward Engineering from Interaction Diagrams - can it be useful?

Daniel Björklund

Sammanfattning

Kodgenerering är en teknik som har varit tillgänglig i modelleringsprogramvaror för Unified Modelling Language (UML) i många år. Aktiviteten att generera källkod från UML-klassdiagram är vanligt förekommande vid utveckling av objektorienterade programvaror. En annan typ av UML-diagram som källkod kan genereras från är interaktionsdiagram. Interaktionsdiagram är väl anpassade för att uttrycka designdetaljer i en programvara men utbudet av programvaror som kan generera källkod från interaktionsdiagram är litet. CircleUML är en sådan programvara. CircleUML kan generera källkod från både sekvens- och kollaborationsdiagram och dess support för modelleringsprogramvaror för UML är inte begränsat till endast en programvara. Denna rapport behandlar generella problem och svårigheter som rör kodgenerering från interaktionsdiagram. Den demonstrerar även utvecklingen av öppen-källkods-programvaran CircleUML och hur den kan vara användbar som en programvara för kodgenerering vid utveckling av objektorienterade programvaror. Rapporten presenterar också hur villkor, iterationer m.m. måste uttryckas i interaktionsdiagram för att CircleUML ska kunna hantera dem. Omfattande interaktionsdiagram som uttrycker fullt fungerande algoritmer presenteras tillsammans med den källkod som genererats från dem för att bevisa den omedelbara användbarheten av en programvara som kan generera källkod från interaktionsdiagram.

Utgivare:	Högskolan Trollhättan · Uddevalla, Institutionen för Informatik och Matematik Box 957, 461 29 Trollhättan Tel: 0520-47 50 00 Fax: 0520-47 50 99		
Examinator:	Steven Kirk		
Handledare:	Steven Kirk, HTU		
Huvudämne:	Programvaruteknik	**Språk**:	Engelska
Nivå:	Fördjupningsnivå 2	**Poäng**:	10
Rapportnr:	2003:PM04	**Datum**:	2003-05-23
Nyckelord:	UML, Forward Engineering, Sequence Diagram, Collaboration Diagram		

Innehållsförteckning

Bilagor

Abstract

Forward engineering is a technique that has been available in Unified Modelling Language (UML) tools for years. The activity of forward engineering or generating code from UML class diagrams is widely used today when developing object-oriented software. Another type of UML diagram that can be forward engineered is an interaction diagram. Interaction diagrams are well suited for expressing design details about software but there are not that many forward engineering tools available that can forward engineer interaction diagrams. CircleUML is such a tool. It can forward engineer both sequence and collaboration diagrams and its support for UML modelling tools is not limited to only one tool. This report discusses general problems and difficulties associated with forward engineering from interaction diagrams. It also demonstrates the development of an open-source software package called CircleUML and how it can be useful as a forward engineering tool in development of object-oriented software. Examples of how conditions, iterations, and other statements must be modelled for CircleUML to be able to handle them are presented. Comprehensive interaction diagrams expressing fully functional algorithms are presented with the generated source code from them to prove the immediate utility of a software package that can forward engineer interaction diagrams.

1. Introduction

Developing object-oriented software may involve the activity of producing design artefacts such as UML diagrams. UML is a modelling language mainly used for modelling of object-oriented software [1]. A type of UML diagram that is especially useful for expressing design details is an interaction diagram. Producing design artefacts is something that can be done in a very detailed way or with few details. The level of details in an interaction diagram can be a problem because a very detailed diagram can easily become difficult to read and grow very large. Such a diagram can become counterproductive.

If interaction diagrams are produced as a pre-programming activity, developers can think about design decisions before implementing the design. Even though many changes to the diagrams may have to be made throughout the programming process, the diagrams can express details about the software that are difficult to express and document in other ways. This way an interaction diagram can be both useful during programming and as a detailed design artefact of the finished software.

Producing source code from diagrams such as UML diagrams can be done either manually or automatically, using code generation software packages. The process of generating source code from UML diagrams is referred to as forward engineering. Forward engineering static diagrams such as UML class diagrams is a rather straightforward process, because class diagrams are well suited for illustrating the software entities that are generated. In contrast, forward engineering dynamic diagrams such as UML interaction diagrams is not such a straightforward process because the generated source code does not follow a predetermined pattern. The information in a static diagram is associated with strict constraints while a dynamic diagram must allow flexibility to be able to illustrate complex situations such as nested conditions and concurrency.

1.1. Problem description

This report will deal with the following questions:

- Can it be useful to automatically forward engineer interaction diagrams?
- Can a forward engineering software package for interaction diagrams work with available UML modelling tools without any adjustments to them or added diagram creation complexity?
- Can the eXtensible Markup Language (XML) and the XML Metadata Interchange (XMI) standard help make such a tool independent of UML modelling tools?

1.2. General objective

The general objectives of this degree project are to:

- Discuss the need for and present general proofs of the utility of a software package that can forward engineer interaction diagrams.
- Prove the immediate utility of a software package that can forward engineer interaction diagrams by implementing the open-source software package CircleUML and provide examples of its usage [2].

1.3. Disposition

This report is divided into nine chapters. The first chapter introduces the problem area and the general objectives of this report. The second chapter goes a little deeper into the background, the gap in previous research, and the limitations. The third chapter explains evaluated technologies and general concepts that are discussed throughout this report such as data binding and code generation patterns. In the fourth chapter, the chosen methods are described and the choices are motivated. Later on in chapter five, a complete description of the implementation of CircleUML is presented. In the sixth chapter, the results are described followed by discussions of the results in chapter seven and conclusions in chapter eight. Chapter nine points out interesting areas for future work. Following chapter nine, one will find the references used throughout the report and after the references, one finds the appendices where among other things the source code of CircleUML can be found.

2. Background

The technique to forward engineer static diagrams in UML has almost become a de facto standard in UML modelling tools today. The reason developers use Computer Aided Software Engineering (CASE) tools to model the architecture and behaviour of software systems is not only to provide an artefact that can be used for communication with other developers. It is also often done for pure documentation reasons and to help the programmer in the process of translating the different diagrams into source code in a programming language [3].

Thanks to the forward engineering functionality in many UML modelling tools, developers can automatically generate source code from class diagrams. This is a common activity in software development projects today. As forward engineering can be an automated process, it can also save time, which in the end means money savings. An automated process is implicitly faster than that same process performed manually. If one considers the situation where a developer manually translates interaction diagrams into source code with the possibility of human errors creeping into the source code, then forward engineering using a specialized software package can save time and money.

An interaction diagram can be either a sequence diagram (see Figure 1) or a collaboration diagram (see Figure 2). The two diagrams can be used in a similar way to express interactions between objects.

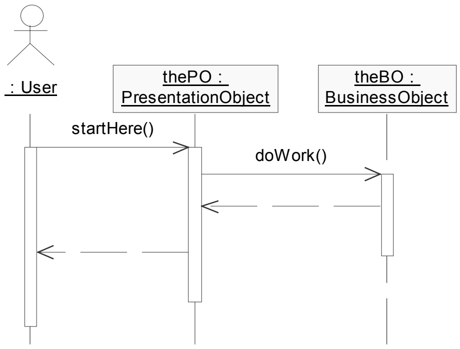


Figure 1. Sample sequence diagram

A sequence diagram unlike a collaboration diagram is focused on illustrating the time and ordering of messages. In collaboration diagrams, sequence numbers are added to the messages to illustrate their order.

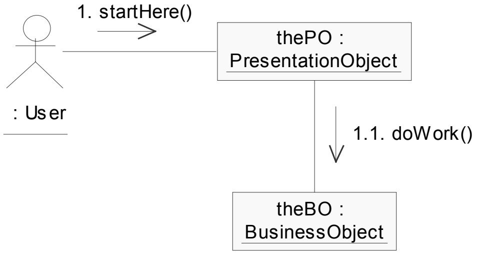


Figure 2. Sample collaboration diagram

One may omit the sequence numbers in a sequence diagram because the diagram itself illustrates the ordering of the messages. As interaction diagrams are used to show interactions between objects, it means that an interaction diagram can show how different objects of a system interact with others by sending messages. The different objects are actually interfaces or instances of classes, and the messages that are sent between them are actually method calls or constructor calls [4]. Because there is such an obvious mapping between the entities in an interaction diagram and their programming language equivalences, one might wonder why there are not that many code generators for interaction diagrams available.

A small number of commercial and non-commercial software packages exist that can reverse engineer source code into interaction diagrams. Reverse engineering means in this case the process of turning source code into UML diagrams. One of the most obvious reasons for reverse engineering into interaction diagrams is to create the diagrams when the source code is finished and keep the diagrams for documentation purposes. If one can use forward engineering or even better iteratively use both reverse and forward engineering then one can keep the interaction diagrams up-to-date with the source code throughout the entire programming process. This would most likely give the interaction diagrams a more central role in UML modelling.

2.1. Related work

The software Together ControlCenter 6.0 developed by the company TogetherSoft Corp. (as of January 15, 2003 acquired by Borland) has the functionality to both forward engineer from and reverse engineer into interaction diagrams [5]. The process of iteratively forward engineering and reverse engineering is called round-trip engineering [6]. One of Borland's main reasons to use the feature to reverse engineer source code into interaction diagrams is that interaction diagrams provide very useful information about the intercommunication of the software. This information is therefore a valuable documentation artefact that should be produced in order to get better documentation coverage of the software. [5].

With the possibility in Together ControlCenter 6.0 to first forward engineer an interaction diagram, do changes to the resulting source code and then reverse that source code into an interaction diagram again, developers can with little effort keep their design artefacts up-to-date with the source code. If UML diagrams for a software project are created for documentation reasons, they must conform to the source code. If they do not conform, then they are of no value because they do not describe the software in its present state.

2.2. Gap in previous research

Software developing companies like former TogetherSoft Corp. has already shown in Together ControlCenter 6.0 that it is possible to generate source code from interaction diagrams [7]. TogetherSoft Corp. has also shown that it is possible to even reverse engineer source code into interaction diagrams, providing a round-trip engineering possibility. UML diagrams can be a very good help for programmers when they create algorithms on which the software is based. They can provide a design foundation where many design decisions already have been made and described in the diagrams. If used in a proper way, interaction diagrams can be translated into source code and thereby save the programmer time translating the diagrams into source code. If this process can be automated with a specialized software package that can work with diagram exports from a number of UML modelling tools, then one can not only generate valuable source code but also use the modelling tool of choice.

2.3. Limitations

The support for translating interaction diagram information into source code using CircleUML will not be focused on a particular file format for a particular tool such as Rational Rose or Poseidon for UML. It will instead be focused on XMI exports produced by these tools or by add-ins for these tools. There are a small number of different Document Type Definitions (DTDs) for different versions of UML that can be used with XMI, therefore referred to as XMI for UML. An immediate reflection to this limitation would perhaps be that if most tools can export diagram metadata into XMI files, then the tool support could be based on the different versions of XMI for UML. The problems associated with retrieving specific diagram information from XMI exports from different tools are discussed in chapter 4.6 about UML modelling tool support in CircleUML. Therefore, CircleUML will support retrieval of diagram information from a smaller number of combinations of UML tools and XMI for UML versions.

This report will only include examples based on XMI-exports from the following two UML modelling tools, Rational Rose Enterprise Edition 2002 and Poseidon for UML Community Edition 1.6. From now on when Rational Rose occur in the text, it actually means Rational Rose Enterprise Edition 2002 and when Poseidon for UML occur in the text it actually means Poseidon for UML Community Edition 1.6.

The main reason for involving Rational Rose and Poseidon for UML was that the use of CircleUML could then be shown with both a commercial software package and a software package that can be used free of charge. Poseidon for UML is not only available as a commercial product but also in a free version called Community Edition [8]. The choice of involving Rational Rose and Poseidon for UML was mainly based on the writer's previous experience from working with both these software packages. Another reason was that for the aspect of modelling tool support in CircleUML, these tools support different versions of XMI for UML. Poseidon for UML uses XMI as its internal file format for saving model information. Poseidon for UML exports into XMI version 1.2 for UML 1.4. Rational Rose can export its models into both XMI version 1.0 for UML 1.3 and XMI version 1.1 for UML 1.3, using an add-in called Unisys Rose XML Tool [9]. The difference in XMI versions in the XMI exporting functionality in these tools increases the possibility of supporting more tools than just Rational Rose and Poseidon for UML. If any other tools conform to the way Unisys Rose XML Tool or Poseidon for UML exports diagram information into XMI, then the range of supported UML modelling tools may be larger than only these two tools.

CircleUML is only guaranteed to generate correct source code for the example sequence and collaboration diagrams supplied with this degree project. Example XMI files and corresponding source code files can be found in Appendix L.

CircleUML will only support generation of source code for the Java programming language. Both Rational Rose and Poseidon for UML support generation of Java source code from UML class diagrams. CircleUML will not provide the functionality to generate source code from UML class diagrams. This means that CircleUML will need class skeletons for the classes where source code needs to be inserted in. Preferably, these class skeletons are generated by the same UML modelling tool that was used when the interaction diagrams were created. Therefore, CircleUML only supports generation of Java source code making it compatible with the two tools that were chosen for this report.

The way an interaction diagram in UML is designed is not only dependent on the UML modelling tool itself but also on the way the designer is constructing and naming the different messages that are sent between objects. For CircleUML to work properly, one must write these messages in a standardized way described in the Interaction Diagram Reference in chapter 4.7.

Concepts in the area of interaction diagrams in this report are only expressed from an object-oriented programming language perspective where for example interaction messages are mapped to method calls, constructor calls and so on.

3. Evaluation of existing technologies

For the development of CircleUML, the evaluation of software process models was limited to the ones discussed by Ian Sommerville in the sixth edition of his book Software Engineering. Sommerville discusses four process models and two hybrid process models. A hybrid process model is a process model that combines two or more process models. One of the four process models discussed is the formal systems development approach. This process model was not evaluated because its ideal range of application is software with stringent safety and security requirements [1]. This does not apply to the requirements of CircleUML.

The data binding techniques which were evaluated are techniques that are presented as the leading XML processing or XML data binding techniques in the second edition of the book Professional XML [10].

The evaluation of code generation techniques was limited to only involve the different source code generation patterns described by Markus Voelter in his upcoming paper about source code generation [11].

The modelling tool support for CircleUML was evaluated from the perspective of XMI and how XMI can make CircleUML support different UML modelling tools.

The evaluation of different ways of interaction diagram design and interaction message design was evaluated from both the perspective of the UML 1.5 specification and from the suggestions of the author of the book Applying UML and Patterns - An introduction to Object-Oriented Analysis and Design and the Unified Process, Craig Larman.

3.1. Software process models

A software process is a collection of related activities that generates artefacts that finally make up the developed software. A particular software process is not always the most suitable process to use for all software development situations. They are usually only suitable for development of a smaller range of software types or situations. A common approach when developing larger software systems is to combine different software processes during the development, using different processes during different phases of the development life cycle [1].

There can be great differences between software processes concerning the different activities that make up the process. Although there are, some activities that often occur when looking at different software processes. Figure 3 shows four activities that are common to many processes [1].

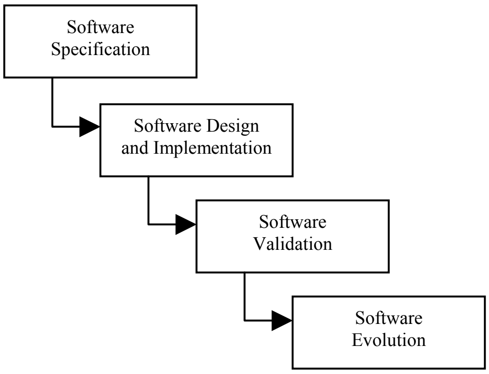


Figure 3. Software process activities from a life cycle perspective [1]

A software process model is a simplified and abstract representation of a software process. The process model is presented from a particular perspective that may for example include the roles people involved in the development play [1].

Some processes are very strict and can for example, restrict that one activity should be finished before one proceeds with a new activity. Often are these activities concerned with producing some artefact that should be used in the next activity [1]. The opposite to this kind of plan-oriented processes, often called heavyweight processes, are the so called lightweight processes or agile software processes that in contrast to the heavyweight processes are a lot less restricted and are focused on short development iterations, where reaction to changes in requirements are considered core activities [12].

The Waterfall model or the software life cycle as it is also known, was the first software process model to be published. It was derived from established engineering processes and got its name from its resemblance with a waterfall because of the cascade from one life cycle phase to another. The Waterfall model is divided into a number of phases that are carried out. According to the model, the work in one phase has to be finished before one begins working with the next phase. In the design phase problems with the requirements may appear, which means that one must go back to the requirements phase again and do the necessary changes before carrying on with the next phase. At some stage, one may have to freeze the artefacts produced in one phase and proceed, even though problems are found that derive from those artefacts. The commitments that then have been made may lead to problems late in the process that actually is the result of those commitments. The fact that the Waterfall model is very strict and inflexible in the sense that it is difficult to respond to changes in late stages has increased the need for other software process models [1].

Evolutionary development or prototyping can be divided into two different types. The first type is exploratory development where an initial prototype is developed from an initial outline specification. This prototype is then evolved throughout the development process. The users give comments on its features and the requirements evolve to finally correspond to the needs of the users [1].

The second type of evolutionary development is throwaway prototyping. Throwaway prototyping is based on the development of individual prototypes that are presented to the users for comments and that are then thrown away. The users give comments on the features of the prototype, which leads to the evolution of the requirements. The prototype is discarded and a new prototype is produced. Because the prototype is discarded after every user review, new features can be experimented with and the structure and design of the prototype does not have to be perfect until the final software is produced [1].

Two of the biggest problems with evolutionary development are the problem to measure the progress of the development and the risk of ruin the structure of the software by responding to requests for changes in the software [1].

Incremental development is distinguished by the iterative approach where small increments of the software are developed, leaving the customers with the possibility to delay detailed requirements decisions. The software is divided into logical subsets or increments, so that each provides some new functionality to the final software. Not all increments must be developed using for example the waterfall model. One can use different process models for different increments, leaving one with the opportunity of choosing the most suitable process model for each increment. By the first increment, the customers only have to outline the requirements of the software and decide on what requirements are most important and what requirements are least important. The most important requirements are implemented in the initial increments and the least important requirements are implemented in the last increments [1].

The approach of developing the software in small increments that are then delivered to the customer for comments gives much of the advantages of prototyping. Because incremental development uses an iterative approach, where the core parts of the software are developed in the beginning and then new features are added to the software, it helps avoid problems where extensive rework is necessary. Because each increment results in a fully usable subset of the final software, the customers can make use of the software even though it does not have all features implemented [1].

Spiral development encompasses (just like incremental development) other software process models. Spiral development does not represent the software process as a sequence of activities as with the waterfall model but instead represents it as a spiral, where each loop corresponds to a phase in the software process. In one loop the waterfall model may be used and in another loop, prototyping may be used. This means that spiral development can be a kind of hybrid process model [1].

The most significant thing about spiral development is its focus on risks. Each loop in the spiral is divided into four sectors where the first sector involves identification of risks. The second sector involves detailed analysis of the identified risks followed by the third sector where a suitable process model is chosen to solve the problem that is identified by the risk. The fourth sector is where decision on looping continuance or not is made [1].

3.2. XML processing

When software systems interchange data in an XML format there are different techniques a programmer can use to retrieve specific parts of this data without binding the data into a more suitable format. One example of such a technique is the Simple API for XML (SAX); another is the Document Object Model (DOM). Both of these provide the programmer with interfaces to the actual structure and data of the XML document. The difference between these two techniques and another technique called XML Data Binding is that they are both structure-centric and work with the data on a very low level, while XML Data Binding is more data-centric and not quite so low-level. SAX and DOM work directly with the structure of the XML document. The programmer must therefore have some knowledge of the structure of the XML document to be able to interact with the interfaces. XML Data Binding instead provides an interface to the data that does not depend on the structure of the XML document, leaving the programmer to work with concepts he/she already knows of such as classes, attributes and methods [10].

SAX is an Application Programming Interface (API) that is built on an event-oriented architecture. SAX consists of a number of interfaces that one can implement. These interfaces define different events that are triggered when for example the start of an element is found or when the end of an element is found [13]. Unlike XML, SAX is not an approved standard. SAX is though considered the de facto standard API for processing XML documents [10]. Although SAX is not an approved standard, there are implementations of SAX for most object-oriented programming languages used today [14]. A disadvantage of working with SAX is that it can require processing of the entire XML document repeatedly. This can make SAX rather ineffective. An advantage of SAX is that it is not as memory intensive as other techniques such as DOM.

DOM is a standard provided by the World Wide Web Consortium (W3C). DOM is, like SAX, an API for processing well-formed XML-documents. DOM parses XML documents and builds a tree-like structure, which is much the same as the structure of the data being parsed. Because DOM works in-memory with a tree-representation of an XML document, it can be very resource-intensive to work with DOM and large XML documents. Just as working with SAX the programmer needs to have knowledge of the actual structure of the XML document when working with DOM. Even though it can be very simple to extract small pieces of data from the tree, one can almost be certain that conversion of the data into basic data types will be necessary [15].

When working as a programmer, using SAX, one will only need to use the different interfaces provided with SAX. Working with DOM is much the same. DOM provides a number of different interfaces that is used together with an XML Parser that implements these interfaces [14]. To be able to use the same technique to both retrieve data from a DOM tree and build a completely new DOM tree in memory can be very useful for a programmer that needs to both read and construct XML documents. DOM and SAX are very useful when one needs to work not only with the actual data but also with the structure of the data [10]. When one only needs to work with the data and not the structure, other techniques are more suitable.

Data binding is a technique used to map the structure of some data from one format into another format that is more suitable for the intended use of the data. For example if a programmer is about to work with data, representing business objects that are stored in an XML document, then perhaps XML is not the most suitable format in which to handle the data. One would most likely want to work with the data on a more abstract level, without needing to know the actual structure of the data. With data binding one can map the structure of an XML document into ordinary object-oriented entities such as classes and attributes, preserving the structure of the original data. This specific usage of data binding is known as XML Data Binding [14].

An XML Data Binding Framework is a software package that can map an XML document with its elements and attributes into an object model in an object-oriented programming language.

> *"An object model in Java is simply a set of classes and primitive types that are typically grouped into a logical hierarchy to model or represent real-world or conceptual objects. An object model could be as simple as consisting of only one class, or very complex consisting of hundreds of classes."*[10]

APIs like SAX and DOM are useful when one needs to extract small pieces of data from an XML document. In contrast, XML Data Binding Frameworks are very useful when one needs to handle very large XML documents with nested elements and complex structures [14]. A framework such as Castor or Java XML Binding (JAXB) can generate Java source code for entire XML documents using XML Schemas [16][17]. They generate code that maps to the XML document structure by generating classes and attributes for the corresponding elements and attributes in the XML document. They generate `get` and `set` methods for all attributes and most importantly, they generate methods for marshalling and unmarshalling [14].

Marshalling or serialization (see Figure 4) means the process of translating for example a Java class into an XML representation [14].

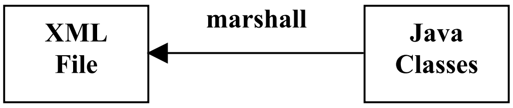


Figure 4. Marshalling from a workflow perspective

Unmarshalling (see Figure 5) is the complete opposite of marshalling, namely translating an XML document into an object model consisting of one or more Java classes, provided that one is unmarshalling into Java source code. In the case of data binding the term, deserialization is synonymous with the term unmarshalling [14].

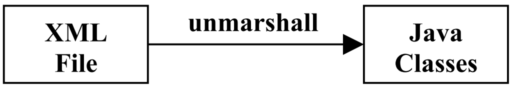


Figure 5. Unmarshalling from a workflow perspective

Another XML Data Binding Framework is the open source project Zeus. Zeus is one of the few frameworks that can generate Java source code from DTDs [18]. DTDs do not provide a rich set of data types. Most object-oriented programming languages today provide a number of different data types for numbers and characters. DTDs only provide a small number of data types while an XML Schema instead provides a more flexible way of working with different data types. XML Schemas are therefore more suitable for data binding into an object-oriented programming language because specific data types in the XML Schema can easily be mapped into data types in the programming language [13].

3.3. Code generation patterns

There are different ways of generating source code from an abstract model such as a UML diagram. Markus Voelter describes seven different patterns that are used for code generation purposes. The patterns evaluated in this report are the patterns/techniques that Voelter describes as being applicable for code generation from XMI/XML documents and UML diagrams [11].

Template based code generation with filtering is well suited for situations when one needs to retrieve small fragments of information from a higher-level specification. This higher-level specification can for example be an XMI document containing large sections of information that are of no interest at all. Retrieving the fragments of interest can be done by using an approach called pattern matching. Pattern matching means that one parses the document looking for sections that matches certain patterns. Such a pattern can for example be expressed as an XPath expression [11].

The retrieved information from the filtering is then mapped into templates that later on can easily be translated into source code for the targeted programming language [11].

Template based code generation from metamodel is a code generation pattern that is well suited for situations when one wants to express a high-level model in terms of some domain specific concepts and then translate these into source code. The process of generating code with this pattern involves two separate steps. The first step is to define the metamodel in terms of the domain specific concepts. The second step is to define the template or templates also in terms of the domain specific concepts. The template includes functionality for validating constraints on the metamodel and generation of source code for the targeted programming language [11].

3.4. UML modelling tool support

XMI is a standard provided by the Object Management Group (OMG). This standard is based on three core parts. These three parts are XML, UML, and the Metadata Object Facility (MOF). The first two of these three parts are of interest in this report whereas MOF is not practically used in CircleUML in any way and therefore is not discussed in this report [19].

The XMI specification, irrespective of version, is increasingly becoming the de facto standard for interchanging metadata between software systems. Because XMI is based on the open standard XML, which is programming language-neutral and platform-neutral, it can be used and incorporated by all UML modelling tools available today. XMI provides a general specification for storing metadata from UML models [19]. This gives developers and modellers the possibility to interchange metadata between UML modelling tools as long as they comply with the XMI standard and use the same XMI version. Unfortunately, this only works well in theory. In practice, this interchange of metadata between UML modelling tools is not as easy as one might imagine. Problems associated with metadata interchange are discussed in chapter 4.6.

There are products such Meta Integration Model Bridge (MIMB) from Meta Integration Technology Inc. that actually can do these conversions of XMI-exports between different modelling tools [20]. The use of XMI for interchanging metadata between tools has not become very useful in practice because it can only interchange metadata and not graphical representations of models. Some of the largest competitors on the market including Rational Software Corp. have worked hard for a diagram interchange metamodel for the upcoming version 2.0 of the UML standard from OMG [21]. The new version will most likely specify how diagrams are graphically represented by specifying how a diagram's placement, a diagram element's alignment, a diagram element's colour, and a diagram element's font and many other properties are represented [22].

There are a number of approved versions of the XMI standard for UML and there are some not yet approved versions as well. UML has gone through a number of revisions since its first version 0.8 back in 1995. In September 1997 the final proposal of UML version 1.1 was submitted to OMG. In November 1997, OMG officially adopted UML as its standard modelling language. A number of changes including resolving some major legacy issues had been made to the UML standard when version 1.3 was presented in 1999. In version 1.0 of UML, there was already a facility for interchanging UML model metadata between modelling tools. However, it was not until version 1.3 of UML that this facility became useful when a DTD for XMI was included in the standard specification [21].

3.5. Interaction message design

The UML specification for version 1.5 does not prescribe the format for all parts of an interaction message. Some parts are more precisely defined while others are less precisely defined. The specification defines an interaction message or interaction stimulus the following way [23]:

```
Predecessor guard-condition
sequence-expression return-value
:= message-name argument-list
```

In this message syntax, the only part that cannot be omitted is the message-name while all other parts may be omitted. The 'Predecessor' is not relevant to this report and it is not defined in the interaction diagram reference for CircleUML in chapter 4.7. The guard-condition can be expressed either as a basic condition clause or as an iteration clause. UML does not prescribe how these conditions must be expressed. The specification suggests either the guard-conditions to be expressed in pseudo code or in a programming language, also the argument-list may be expressed in pseudo code or in a programming language [23].

Craig Larman describes the basic syntax for message expressions the following way [4]:

```
return := message(parameter :
parameterType) : returnType
```

The sequence and guard-condition expressions are omitted in Larman's basic syntax but he suggests that both sequence numbers and guard-conditions should be expressed the same way as the UML specification describes [4].

A collaboration diagram must include sequence numbers to show the ordering of messages while in a sequence diagram the sequence numbers may be omitted. The location of messages in a sequence diagram illustrates the order of the messages and the sequence numbers may be included or not.

4. Methods

The decisions on software process model, XML processing technique, software architecture, development environment, code generation technique, UML modelling tool support, and interaction diagram design will be presented and motivated in the next few sub chapters.

4.1. Development process

An incremental development based on the waterfall model for each increment, was found to be the most suitable process model for the development of CircleUML. The different increments for the development were chosen to closely follow the use cases identified during requirements analysis of CircleUML. The reason an incremental development approach was found most suitable was that it gives the developer the opportunity of not having to make design decisions for all functionality in the beginning of the development process. It also makes it possible to evaluate the work of the developer after each increment, giving opportunities for refinement of the requirements throughout the entire development process. The given arguments for the choice of an incremental development approach were also the arguments for why a traditional waterfall model approach was not chosen.

An evolutionary development approach did not become a serious candidate because its biggest advantage over competing process models is the close collaboration with the users of the resulting system. CircleUML was developed without user contacts or user reviews. The feature of user reviews on the software is part of the incremental development in the sense that for each increment users can use the functionality implemented so far and give comments for the next increment.

The spiral development approach was never seriously considered for the development of CircleUML because of its focus on development risks. Because of this, spiral development was not found the most suitable approach in comparison with the other process models/approaches that were evaluated. Spiral development is much like incremental development except for its focus on risks. CircleUML was not found to be such a high-risk project where it could be valuable to focus on risks such as the developer not being familiar with the programming language or not having enough experience with UML modelling and creating interaction diagrams.

4.2. XML processing technique

The need for XML processing in CircleUML was limited to only retrieving data from XML documents. CircleUML needed to retrieve small parts of the document and these parts were located in various sections. The need for a technique that could easily retrieve specific elements within an XML document where a value of a certain attribute of that element was known was something that gradually became the focus for evaluation and choice of XML processing technique for CircleUML.

The evaluation of the most suitable XML processing technique for CircleUML started out with the testing of the open-source project Zeus. Because OMG provides DTDs for three approved versions of XMI for UML, Zeus was an appropriate XML data-binding framework thanks to its ability to generate Java source code from DTDs. It worked rather well and Zeus managed to generate about

3500 classes and interfaces for the four DTDs that were used. As the testing of the unmarshalling functionality in Zeus started, problems arose. Zeus Beta 3.5 that was used during the testing appeared to have problems with determining the end of an element if that same element contained nested elements with the same name as the outer element.

For example, in the DTD for XMI 1.0 for UML 1.3 there is an element called <Foundation.Core.Namespace.ownedElement>. This element can reside within other <Foundation.Core.Namespace.ownedElement> elements. This is illustrated in Figure 6 where the <Foundation.Core.Namespace.ownedElement> element is abbreviated as <F.C.N.oE>.

```
<XMI>
  <XMI.header>
  </XMI.header>
  <XMI.content>
    ...
      <F.C.N.oE>
        ...
          <F.C.N.oE>
          ...
           </F.C.N.oE>
        ...
      </F.C.N.oE>
    ...
  </XMI.content>
</XMI>
```

Figure 6. Abstract of example XMI 1.0 for UML 1.3 element hierarchy

Zeus was in this case not able to determine the end of the outer <Foundation.Core.Namespace.ownedElement> element and stopped unmarshalling when it reached the first end tag of this element when it instead should have continued until it reached the second end tag. Because unmarshalling the XML documents correctly was an absolute necessity, this ruled Zeus out of the picture and the search for an XML processing technique continued. The evaluation turned to the DOM API that is a technique that requires the developer to have some knowledge of the structure of the XML document. By studying where different parts of an interaction diagram are stored in an XMI document one quickly gets a basic understanding of the structure of the document. This knowledge is needed when working with DOM and it is very useful if one wants to combine it with another technique called XPath that works very well with DOM.

XPath is a language for addressing elements and attributes in a general way. It also includes some basic functions for formatting, manipulation, and conversion of character data. Because W3C provides specifications for both DOM and XPath, it has made the integration of these two techniques easy for developers [14].

SAX could have been a serious candidate at this stage but the advantage of being able to easily find a specific element with a known value of its attributes using XPath, DOM proved to be a technique that would be very well suited for the needs of CircleUML. DOM provides a simple way for the developer to navigate up and down the element hierarchy. Using DOM and XPath it is also very easy to retrieve all elements with a specific name independent of their positions in the document. Such situations occurred when working with CircleUML, which further proved DOM to be the right choice of XML processing technique.

4.3. Software architecture

CircleUML was intended to be used by a single user and therefore a client/server architecture was not a serious candidate. Because CircleUML was intended to run on a single computer, the need for a distributed architecture was not there either. CircleUML was also intended to be used closely with a UML modelling tool and therefore CircleUML could be developed as a Rational Rose add-in or a plug-in for Poseidon for UML or any other UML modelling tool that one can develop plug-ins for. If one looks at the field of application for CircleUML then to develop CircleUML as a plug-in for a specific tool was out of the question. CircleUML was intended to be used with XMI-exports from different UML modelling tools and not only support one specific tool, which left CircleUML to be developed as a typical standalone application.

Many UML modelling tools today are targeting multiple operating systems. Poseidon for UML can run on a number of operating systems just as Together ControlCenter 6.0 and many other tools as well. In the case of Poseidon for UML, it is written in the Java programming language. For CircleUML, it was important that it could run on multiple operating systems and consequently be usable with a larger number of UML Modelling tools running on different platforms and operating systems. The possibility to target multiple platforms and operating systems was the reason why the choice of programming language for CircleUML was Java.

The main objective for CircleUML was to parse an XMI document and generate source code for the interaction diagram information stored in it. CircleUML was not intended to store any information about its parsing or code generation. Therefore, there was no need for storing any information in for example a database and because there are two main packages in CircleUML, a 3-tier architecture did not meet the requirements of CircleUML. The top-most package in CircleUML in Figure 7 is the Graphical User Interface package (GUI layer) and beneath it, there is the business logic package (Logic layer) where the business objects are located. There is also a package for shared objects (Shared package) such as interfaces and objects holding static variables. The package for shared objects was not considered a separate layer because it was not of such significance that a separate layer would be appropriate. CircleUML was therefore developed with a two-tier architecture.

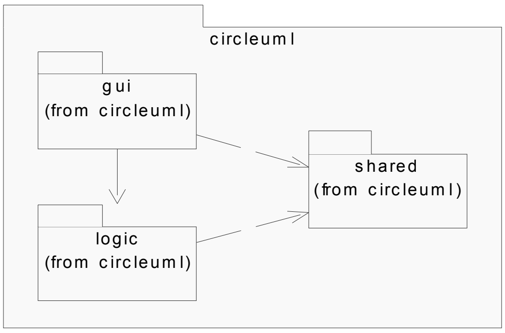


Figure 7. Package hierarchy of CircleUML

4.4. Development environment

CircleUML was developed with Sun Microsystem's Integrated Development Environment (IDE) called Sun One Studio 4 CE. CircleUML uses the DOM implementation from Java API for XML Parsing (JAXP) that is included in Java 2 Platform, Standard Edition version 1.4 [24]. CircleUML also uses the XPath implementation of the Xalan-Java XSLT processor [25]. The example interaction diagrams throughout this report are created using Rational Rose and Poseidon for UML. Example XMI files provided with this report are created either using the Unisys Rose XML Tool add-in for Rational Rose version 1.3.5 or using Poseidon for UML.

4.5. Code generation pattern

For the development of CircleUML the chosen code generation pattern/technique was a template-based pattern with filtering using pattern matching. The reason this pattern was chosen was that it is very well suited for code generation from XML documents and the filtering can easily be implemented using XPath expressions on the DOM tree. Template based code generation from metamodel could have been applicable for the purposes of CircleUML but the need to express constraints on the UML diagram and express it in terms of specific domain concepts was never present for the code generation functionality in CircleUML.

4.6. UML modelling tool support

The ideal solution would be if the UML modelling tool support in CircleUML could be based on the different versions of XMI for UML. Because OMG provides detailed specifications for XMI and DTDs for different combinations of XMI versions and UML versions, it is definitely possible to develop such a software package. The software package would then conform to the XMI specifications and the DTDs that OMG provides but the compatibility with UML modelling tools that can export diagram metadata into XMI files would most likely be limited.

The reason the tool compatibility would be rather limited is that even though the specification is detailed it gives tool vendors some flexibility in the way their tools structures their XMI documents. The following figures illustrates where two different UML modelling tools that can generate XMI files in accordance to XMI 1.0 for UML 1.3 places a certain element in the element hierarchy. The open source UML modelling tool ArgoUML version 0.12 places the `<Behavioral_Elements.Common_Behavior.CallAction>` element in the following place in the element hierarchy as illustrated in Figure 8 [26]. The `<Behavioral_Elements.Common_Behavior.CallAction>` element is in Figure 8 abbreviated as `<B_E.C_B.CallAction>` and the `<Foundation.Core.Namespace.ownedElement>` element is abbreviated as `<F.C.N.ownedElement>`.

```
<XMI>
  <XMI.content>
    <Model_Management.Model>
      <F.C.N.ownedElement>
        <B_E.C_B.CallAction>
        </B_E.C_B.CallAction>
      </F.C.N.ownedElement>
    </Model_Management.Model>
  </XMI.content>
</XMI>
```

Figure 8. Abstract of an example ArgoUML 0.12 XMI document

The Unisys Rose XML Tool version 1.3.5 places the `<Behavioral_Elements.Common_Behavior.CallAction>` element in the following place in the element hierarchy as illustrated in Figure 9.

```
<XMI>
 <XMI.content>
  <Model_Management.Model>
   <F.C.N.ownedElement>
    <B_E.C.Collaboration>
     <F.C.N.ownedElement>
      <B_E.C.Collaboration>
       <F.C.N.ownedElement>
        <B_E.C_B.CallAction>
        </B_E.C_B.CallAction>
       </F.C.N.ownedElement>
      </B_E.C.Collaboration>
     </F.C.N.ownedElement>
    </B_E.C.Collaboration>
   </F.C.N.ownedElement>
  </Model_Management.Model>
 </XMI.content>
</XMI>
```

Figure 9. Abstract of an XMI document generated by Unisys Rose XML Tool v1.3.5

There is a distinct difference in the way ArgoUML version 0.12 and Unisys Rose XML Tool version 1.3.5 structures XMI documents from identical interaction diagrams as illustrated in Figure 8 and Figure 9.

There are also vendors that have made their own expansions to their XMI exports to be able to store other information then just diagram metadata. For example, Unisys has their own expansions that allow them to store some information about the graphical representations of the models. At the time of writing the current version of UML does not define how graphical representations of diagrams should be expressed with XMI. This is something that most likely will become present in future UML versions [21].

Both vendor specific expansions to XMI and the wide range of differences in XMI document creation among tools were the reasons why CircleUML was not focused on the different versions of XMI and the DTDs for UML provided by OMG. CircleUML was instead focused on the implementation of XMI document creation in the two modelling tools that were mentioned in the limitations in chapter 2.3. CircleUML supports XMI 1.2 for UML 1.4 as implemented in Poseidon for UML and XMI 1.0 for UML 1.3 and XMI 1.1 for UML 1.3 as implemented in Unisys Rose XML Tool version 1.3.5.

4.7. Interaction diagram reference

For CircleUML to be able to generate source code from interaction diagrams, CircleUML had to be able to parse the information in an XMI document and translate it into source code. Therefore, CircleUML had to have strict message syntax rules so that interaction messages could be parsed. Otherwise, CircleUML could not translate the messages into the corresponding source code. For CircleUML, the message syntax was mostly based on the message syntax from the specification for UML version 1.5. Some aspects were also derived from the message syntax suggested by Craig Larman, especially the way he suggests that parameter types and return types shall be expressed and added for clarity [4]. To make message parsing and code generation simpler, additional concepts were added to the syntax expressions. Examples of such additions are the reserved words `try`, `catch`, `finally`, `if`, `else if`, `else`, `for`, `while`, `do while`, `switch`, and `case`. For example for CircleUML to be able to determine if a condition is an `else if`-statement following an `if`-statement or if the condition is a new `if`-statement, CircleUML had to have these reserved words.

First, some general guidelines for the interpretation of the message syntax are presented. Then the syntax of the different basic interaction message types for CircleUML are described followed by detailed descriptions for each one.

Expressions within brackets are optional and expressions not within any type of brackets are compulsory. Underscore indicates a single space. Words within brackets are supposed to be replaced with real conditions, variable names, or variable types. Expressions within curly brackets indicate that one of the words from the pipe-separated list should be used. All messages must have a preceding sequence-number no matter if the diagram is a sequence diagram or a collaboration diagram. Parsing the sequence numbers is the only way CircleUML can tell the order of the messages. The different levels of the sequence number must be separated by points. The following sequence number `1.1.2.` is a correct sequence number expression for CircleUML while the following sequence number `2.11` is not because it does not end with a point as needed by CircleUML.

```
sequence-number_:_{if|else  if|for|
do while|while}[condition]
```

The word condition in `[condition]` is supposed to be replaced with an actual condition expression such as `[booleanVariable == true]` or `[intVariable : int > 0]`.

```
sequence-number_:_[return_:=_]
messageName([parameter_:_parameter
Type])[_:_returnType]
```

The word return in `[return_:=_]` is supposed to be replaced with the name of the variable that will hold the result of the method call or constructor call followed by a single space, a colon, an equals sign, and another single space. The word messageName is supposed to be replaced with the name of a method or constructor, the word parameter is supposed to be replaced with the name of a parameter, and the word `parameterType` is supposed to be replaced with the name of the type of the parameter. The final word returnType is supposed to be replaced with the name of the return type of the method and must be omitted for constructor calls. For constructor calls, CircleUML retrieves the type information from the classifier name of the message receiver. The return type indicates of what type the return variable will be in the generated source code and it helps CircleUML locate the correct method in a source code file.

```
sequence-number_:_{try|else|
case[else]|finally}
```

CircleUML supports such diverse statements as try-catch-finally statements and switch statements. The above syntax example describes messages that are static in the sense that one shall not replace any text for conditions or variable names and such things. Note that case[else] shall be left as it is and the word else shall not be replaced as described in the general guidelines for interpretation of the syntax expressions.

```
sequence-number_:_catch[catch-
expression]
```

The word catch-expression is supposed to be replaced with a real expression for a catch statement. Such a statement can be catch[java.io.IOException e] which, preferably is expressed in a programming language.

```
sequence-number_:_switch
[variable_:_variableType]
```

The word variable in [variable_:_ variableType] is supposed to be replaced with the name of the variable that the switch statement shall test. The name of the type of the variable is supposed to replace the text variableType.

```
sequence-number_:_case[case-
condition]
```

The word case-condition is supposed to be replaced with a constant expression of either integer type or character type.

```
sequence-number_:_instanceVar_:=_
null
```

To express an explicit release of an object one has to follow the above syntax example. The word instanceVar is supposed to be replaced with the name of the instance variable that shall be released.

```
sequence-number_:_variable_:_
variableType_:=_value
```

To be able to express a statement where one assigns a variable a specific value, the above syntax example illustrates how this is possible with CircleUML. The word variable is supposed to be replaced with the name of the variable, the word variableType is supposed to be replaced with the name of the type of the variable and finally the word value is supposed to be replaced with the assign value.

5. CircleUML

CircleUML was developed using an incremental development approach. This approach implies that the software is developed in a number of increments. The development of CircleUML was divided into three increments, where each increment corresponded to one of the three use-cases identified during the analysis phase.

Detailed presentation of the requirements of CircleUML can be found in Appendix M.

5.1. Analysis

During analysis of the functional requirements of CircleUML, the following three use-cases were identified as illustrated in figure 10.

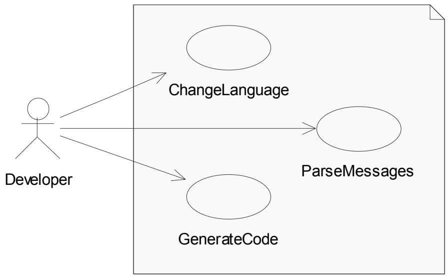


Figure 10. Use case diagram with identified CircleUML use cases

For the first development increment, the ChangeLanguage use case was implemented including the core architecture and the GUI of CircleUML as seen in Appendix A.

For the second development increment, the functionality for the ParseMessages use case was implemented as seen in Appendix B.

For the third development increment, the functionality for the GenerateCode use case was implemented as seen in Appendix C.

5.2. Design and patterns

Because the work with the UML class diagram artefacts of CircleUML was an ongoing process throughout every development increment, they are not displayed as they were at the different increment changes. The different classes and interfaces can be seen in their final state in Appendix D.

The message parsing functionality in CircleUML has already been briefly covered in the methods chapter. The message parser uses pattern matching and wildcard comparisons to try to recognize a message as compatible with CircleUML. The Logic layer of CircleUML has functionality for comparing strings without wildcards with strings that contain wildcards. An asterisk wildcard denotes any number of valid characters and a question mark wildcard denotes a single valid character.

Both the GUI layer and the Logic layer are dependent on the Shared package, but the only other dependency is between the GUI layer and the Logic layer. Therefore, the Logic layer is not dependent on the GUI layer, which means that objects in the Logic layer cannot access objects in the GUI layer.

If one considers the situation where objects within the Logic layer needs to set the state of the GUI, the Logic layer needs to access objects in the GUI layer. This is clearly not possible because only the GUI Layer has access to the Logic layer and not the other way around.

The solution to this problem was implementing a design pattern called the State pattern. This pattern is one of the GoF patterns. GoF is an abbreviation of Gang of Four, which is a nickname for the four authors of the book Design Patterns - Elements of Reusable Object Oriented Software [27]. The State pattern is intended to allow objects to change their behaviour when the internal state of the objects change. In CircleUML, a variant of the State pattern was used to solve the problem of letting the Logic layer change the state of the GUI. The State pattern defines a solution where the different states are implemented as separate classes that all inherit from an abstract class that declares a sort of interface common to all types of states. The different state classes individually provide state-specific implementations [28]. For CircleUML the different states that the GUI can be set to are stored as static constant variables in an abstract class called GUIState. The GUIState class is accessible throughout the entire application because it is located in the Shared package. This way all objects that want to set the state of the GUI can use the GUIState class to get a valid state variable and then use a shared package interface called IGUIUpdater to tell the GUI object to change state and take appropriate action.

A similar dependency problem was concerned with when objects in the Logic layer need to write log information to the GUI. The solution to this problem was also the interface IGUIUpdater. This interface contains a method for setting the state of objects in the GUI and a method for writing log messages and error information to the GUI. When objects within the GUI instantiate objects within the Logic layer an instance of the IGUIUpdater is sent as a parameter to the constructors of these business logic objects. This instance of the IGUIUpdater is actually a casted instance of the MainFrame class from the GUI layer that implements the IGUIUpdater interface.

CircleUML has the functionality for changing language for menus, label captions, log information, and error messages. CircleUML supports both the English language and the Swedish language. The captions, messages, and phrases are used throughout the entire application. Because the language classes contain all phrases and captions, sending an instance of the chosen language class to every object was not considered the best solution to the problem. Instead, CircleUML uses the Observer and Singleton design patterns.

Both the Observer pattern and the Singleton pattern are GoF patterns. The Observer pattern is also known as the Publish-Subscribe pattern where one object is referred to as an observable object and other objects are referred to as observers which observes the observable object. This pattern is very useful in situations where one wants to notify the different observer objects to update themselves because some changes have been made in the observable object [28]. The Singleton pattern is used in situations where one needs to have one and only one instance of a class. Instead of letting other objects create instances of a class, one can implement the Singleton pattern and have that class control its own instantiation and provide a way to return the same instance to all objects that request it [28].

In CircleUML, the Observer pattern was not used in its ideal way where there is a one-to-many dependency. In CircleUML, there is a one-to-one dependency between the GUI class MainFrame and the LanguageObservable class that handles the language changes. The MainFrame object subscribes as an observer to the LanguageObservable object and whenever the language is changed the LanguageObservable object notifies the MainFrame object to update the GUI. The reason why the Observer pattern was used even though there was no one-to-many dependency was that there is an implementation of the Observer pattern in the Java 2 Platform Standard Edition 1.4 API. Using the existing Observable class and Observer interface, which provides the functionality that was needed in CircleUML, was a better approach than to write a special implementation of the functionality for notifying the MainFrame object to update the GUI.

5.3. Restrictions

CircleUML has functionality for retrieving methods and constructors from Java source code files. This functionality means that CircleUML can retrieve the method body for one specific method or constructor within a Java source code file. There can be difficulties in determining the end of a method body if the source code is not free of syntax errors and cannot be compiled. Such problems can be an uneven number of curly brackets or incorrectly written comments. Therefore, CircleUML cannot be guaranteed to be able to determine the end of a method body if the source code is not compilable and involves syntax errors. This means that CircleUML is limited to only work properly with compilable Java source code files.

Another limitation with CircleUML is that it cannot determine if the method body already contains the same code that is about to be inserted into it. CircleUML was designed to only preserve the current code within the method body and insert the new code at the beginning of the method body.

One feature of CircleUML is that it tries to open all source code files, which are needed when CircleUML needs to insert the generated code. This feature is necessary if the user is to avoid having to locate all necessary files on his/her own. A feature in the Java programming language and many other programming languages is to structure classes in packages or namespaces. This gives developers the possibility to have multiple classes or interfaces with identical names but within different packages or namespaces. This is something that CircleUML cannot handle and therefore interaction diagrams parsed by CircleUML cannot include classes with identical names but within different packages.

Another limitation with CircleUML is the number of parameters for a message that can be interpreted and parsed. The UML 1.5 specification describes multiple parameters to be separated by commas [23]. CircleUML can only parse messages where at maximum one parameter and parameter type is expressed. The following example message is therefore not supported by CircleUML. It will be treated as an unknown statement and inserted directly into the code without any translation.

```
1. : messageName(param1 : type1,
param2 : type2)
```

In CircleUML the number of conditions within a condition expression, is also limited. CircleUML can only parse condition expressions that consist of one condition. Condition expressions that consist of multiple conditions separated by logical AND or logical OR operators are not supported and must be separated and expressed in separate condition expressions. The following condition expression is therefore not supported by CircleUML.

```
1.1. : if[var1 == true && var2 ==
false]
```

The interaction diagram reference for CircleUML defines which variants of messages CircleUML can translate into source code. If CircleUML cannot recognize an interaction message as a supported message then that message will not be translated at all and inserted as it is into the source code file for manual translation.

CircleUML does not have any functionality for sorting the interaction messages retrieved from an XMI file. If they are not stored sorted in the XMI file and parsed by CircleUML in that order then CircleUML cannot be guaranteed to generate correct source code.

When return types and return variables are expressed in an interaction diagram message CircleUML generates two separate lines of code in the resulting source code. One line of code will be the declaration and initialisation of the variable holding the return value. The other line of code will be the actual method call. Because a method or constructor can have parameters, there is a possibility to programmatically let one of the parameters be the variable holding the return value. This type of situation cannot be handled correctly with CircleUML. If one uses the name of a method parameter or constructor parameter as the return variable for an interaction diagram message, that variable will be declared in the beginning of the method body even though it is already declared as a method parameter or constructor parameter.

A message in a sequence diagram or a collaboration diagram represents by definition an invocation of a method, the creation of an object or the destruction of an object. A message is associated with a message expression that maps to a method in the classifier type of the receiving object except for create and destroy messages. Create and destroy messages are often expressed using UML stereotypes such as `<<create>>` or `<<destroy>>` [4]. Generating source code with CircleUML requires the possibility to model conditions, iterations, switch statements, try-catch-finally statements and combinations of these. If one wants to model a situation where for example three methods will be invoked if a condition is met, then there is a difficulty modelling such a situation. The UML specification suggests that one may provide a guard-condition with the message expression and this works well if it only concerns one message. The problem occurs when multiple messages must be modelled as part of a conditional situation. This problem occurs for every kind of condition, iteration, and so on. For CircleUML to handle this problem, messages to self is used to express statements such as conditions or iterations. They are actually not messages in the sense of method invocations but they are mapped into if statements and for loops and so on in the generated code.

5.4. Usage

Usage descriptions for CircleUML can be found in Appendix L.

5.5. Interaction diagram examples

This chapter contains examples of valid interaction message expressions for CircleUML. For each sequence diagram, descriptions of supported operators and statements are presented.

The following four sequence diagrams show the four different ways Boolean conditions can be expressed in CircleUML. The first and third condition is met if booleanVariable has a true value, while the second and fourth condition is met if booleanVariable has a false value. The following logical operators are supported for Boolean conditional expressions ==, != and !.

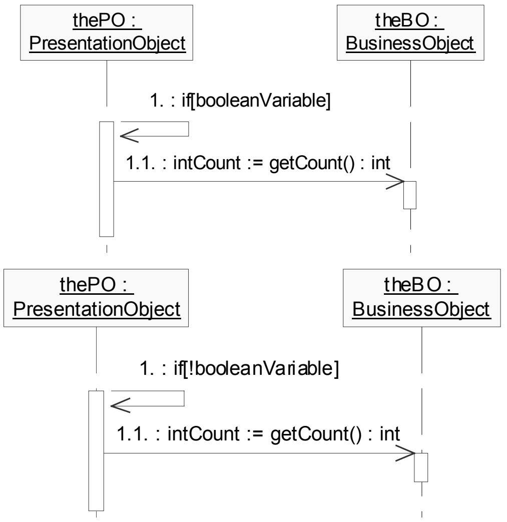

thePO : PresentationObject
theBO : BusinessObject
1. : if[booleanVariable]
1.1. : intCount := getCount() : int
thePO : PresentationObject
theBO : BusinessObject
1. : if[!booleanVariable]
1.1. : intCount := getCount() : int

Figure 11. Compact Boolean conditions

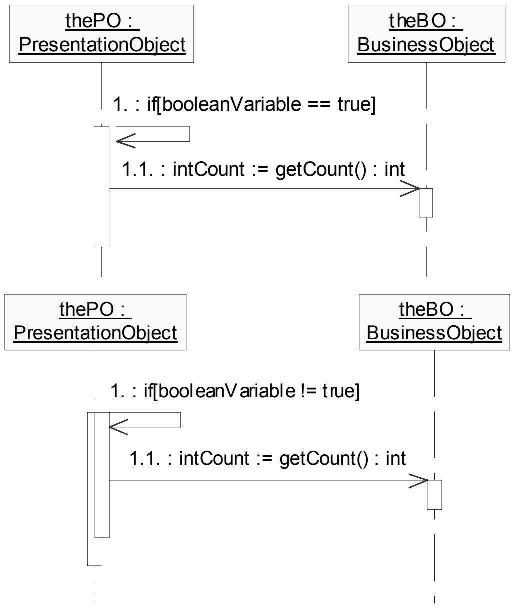

thePO : PresentationObject
theBO : BusinessObject
1. : if[booleanVariable == true]
1.1. : intCount := getCount() : int
thePO : PresentationObject
theBO : BusinessObject
1. : if[booleanVariable != true]
1.1. : intCount := getCount() : int

Figure 12. Basic Boolean conditions supported by CircleUML

The next two condition expressions show how conditions with integer values can be expressed. Basic relational operators may be used such as ==, !=, >, <, >=, and <=.

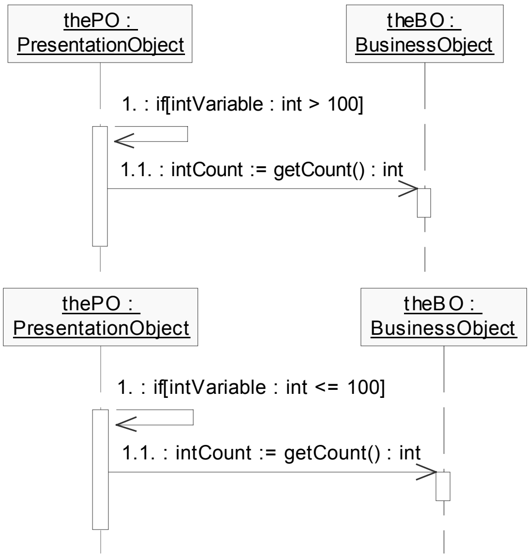

thePO : PresentationObject
theBO : BusinessObject
1. : if[intVariable : int > 100]
1.1. : intCount := getCount() : int
thePO : PresentationObject
theBO : BusinessObject
1. : if[intVariable : int <= 100]
1.1. : intCount := getCount() : int

Figure 13. Example integer conditions supported by CircleUML

To design a constructor call the following sequence diagram shows how such a call have to be designed. CircleUML supports invocation of both constructors without parameters and constructors with one parameter.

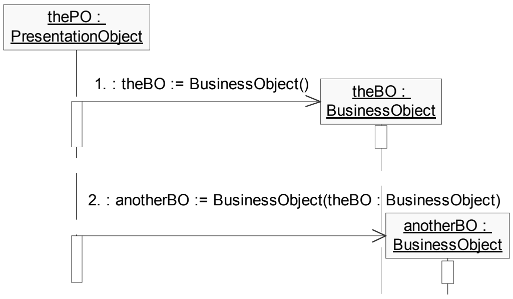

thePO : PresentationObject
1. : theBO := BusinessObject()
theBO : BusinessObject
2. : anotherBO := BusinessObject(theBO : BusinessObject)
anotherBO : BusinessObject

Figure 14. Example constructor calls supported by CircleUML

To design the release of an object the following sequence diagram shows how such a call has to be designed.

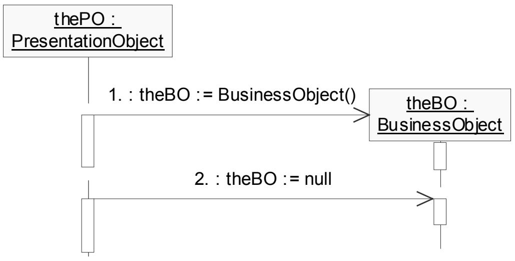


Figure 15. Example of how to release objects in CircleUML

Iterations can be designed in a number of ways. Traditional `for` loops can be designed as described by the following two sequence diagrams.

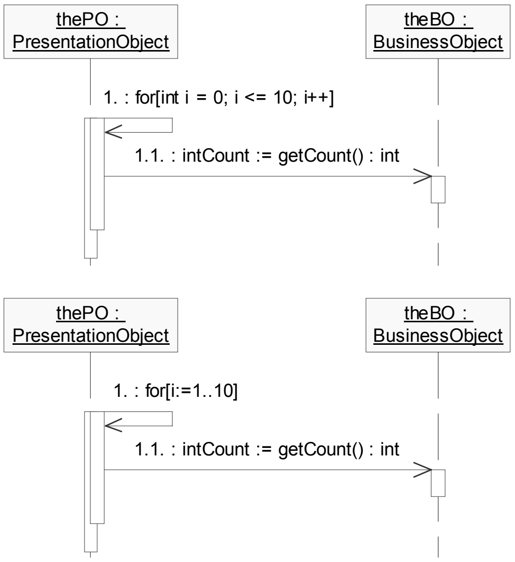


Figure 16. Examples of for-iterations

To describe `while` loops and `do while` loops in an interaction diagram the following sequence diagram shows how such iterations shall be designed.

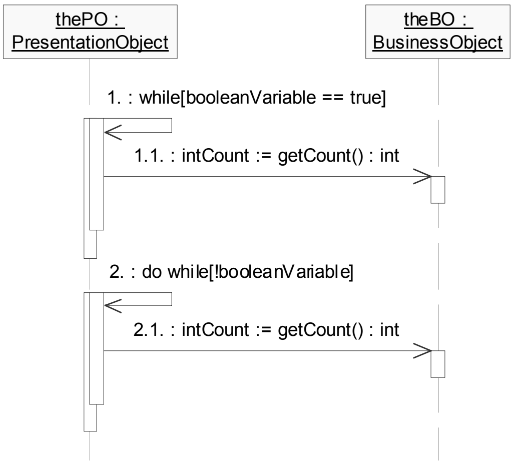

thePO :
PresentationObject
theBO :
BusinessObject
1. : while[booleanVariable == true]
1.1. : intCount := getCount() : int
2. : do while[!booleanVariable]
2.1. : intCount := getCount() : int

Figure 17. Example while-iterations supported by CircleUML

CircleUML do not take into account if the message is synchronous, asynchronous or a call procedure. All three messages will be treated equally by the code generator of CircleUML.

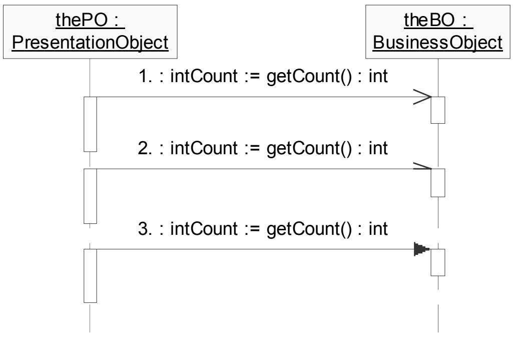

thePO :
PresentationObject
theBO :
BusinessObject
1. : intCount := getCount() : int
2. : intCount := getCount() : int
3. : intCount := getCount() : int

Figure 18. Examples of single statements supported by CircleUML

The following sequence diagram shows how a `switch` statement has to be designed for CircleUML to be able to translate it.

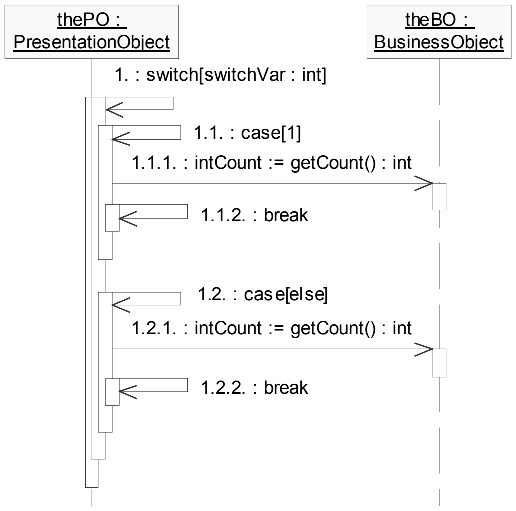

thePO : PresentationObject
theBO : BusinessObject
1. : switch[switchVar : int]
1.1. : case[1]
1.1.1. : intCount := getCount() : int
1.1.2. : break
1.2. : case[else]
1.2.1. : intCount := getCount() : int
1.2.2. : break

Figure 19. Example switch-statement supported by CircleUML

Error handling is an important part of software development. In object-oriented programming languages such as Java `try-catch-finally` statements are a way of dealing with possible errors in an application. The following sequence diagram describes how such a statement has to be designed.

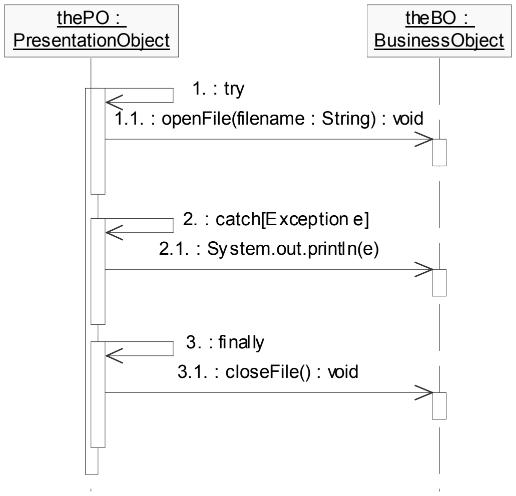

thePO : PresentationObject
theBO : BusinessObject
1. : try
1.1. : openFile(filename : String) : void
2. : catch[Exception e]
2.1. : System.out.println(e)
3. : finally
3.1. : closeFile() : void

Figure 20. Example try-catch-finally statement supported by CircleUML

6. Results

One of the general objectives of this degree project is to prove the immediate utility of a forward engineering tool such as CircleUML. Probably the best way to demonstrate the utility of a forward engineering tool is to show what the tool can produce. To show the utility of CircleUML, four separate example sequence diagrams will be presented with the Java source code generated from these diagrams. The source code presented in this chapter have been indented and modified for better presentation but without changing the statements or the behaviour of the code. The source code generated from the different sequence diagrams are presented directly after each sequence diagram.

The following figures and source code examples differs slightly from the previous sections and may be skipped if one is not interested in seeing detailed examples of what CircleUML can produce.

Figure 21 illustrates a small part of an algorithm that contains a number of constructor calls, single statements, a `while` loop, and some statements that are parsed as unknown by CircleUML. The sequence diagram can be seen as a whole in Appendix I.

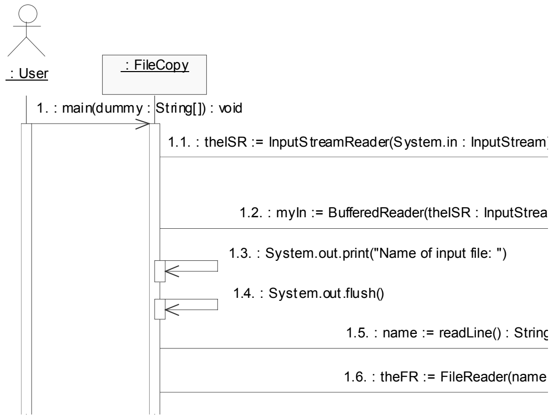
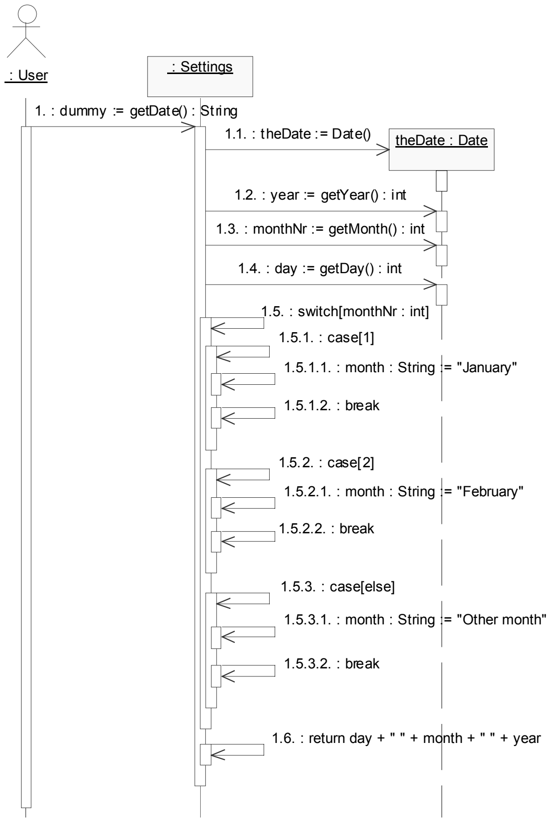

: User
: FileCopy
1. : main(dummy : String[]) : void
1.1. : theISR := InputStreamReader(System.in : InputStream
1.2. : myIn := BufferedReader(theISR : InputStrea
1.3. : System.out.print("Name of input file: ")
1.4. : System.out.flush()
1.5. : name := readLine() : String
1.6. : theFR := FileReader(name

Figure 21. Part of a sequence diagram based on a file copy algorithm [29]

```
public static void
main(String[] args) throws
IOException, FileNotFoundException
{
  InputStreamReader theISR = null;
  BufferedReader myIn = null;
  String name = "";
  FileWriter theFR = null;
  BufferedReader inFile = null;
  FileWriter theFW = null;
  BufferedWriter theBW = null;
  PrintWriter outFile = null;
  String row = "";

  theISR = new InputStreamReader(
  System.in);
  myIn = new
  BufferedReader(theISR);

  System.out.print("Name of input
  file: ");
  System.out.flush();
  name = myIn.readLine();
  theFR = new FileReader(name);
  inFile = new
  BufferedReader(theFR);

  System.out.print("Name of output
  file: ");
  name = myIn.readLine();
  theFW = new FileWriter(name);
  theBW = new
  BufferedWriter(theFW);
  outFile = new
  PrintWriter(theBW);

  while(true) {
    row = inFile.readLine();
    if(row == null) {
      break;
    }
    outFile.println(row);
  }

  outFile.close();
}
```

Figure 22 illustrates a getDate method that consists of a constructor call, a few single statements, and a `switch` statement with incomplete functionality.

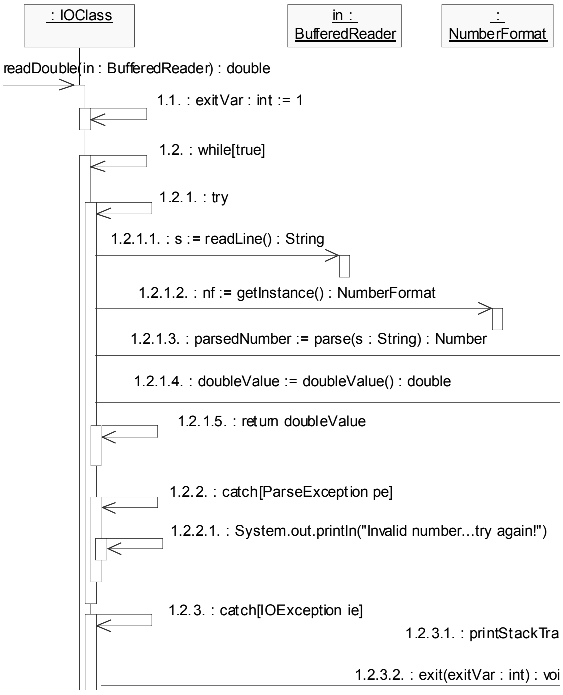


: User
: Settings
theDate : Date
1. : dummy := getDate() : String
1.1. : theDate := Date()
1.2. : year := getYear() : int
1.3. : monthNr := getMonth() : int
1.4. : day := getDay() : int
1.5. : switch[monthNr : int]
1.5.1. : case[1]
1.5.1.1. : month : String := "January"
1.5.1.2. : break
1.5.2. : case[2]
1.5.2.1. : month : String := "February"
1.5.2.2. : break
1.5.3. : case[else]
1.5.3.1. : month : String := "Other month"
1.5.3.2. : break
1.6. : return day + " " + month + " " + year

Figure 22. Fictitious example of a get date algorithm

```
public String getDate() {
  Date theDate = null;
  int year = 0;
  int monthNr = 0;
  int day = 0;
  String month = "";

  theDate = new Date();
  year = theDate.getYear();
  monthNr = theDate.getMonth();
  day = theDate.getDay();

  switch(monthNr) {
    case 1:
      month = "January";
      break;
    case 2:
      month = "February";
      break;
    default:
      month = "Other month";
      break;
  }

  return day + " " + month + " " +
  year;
}
```

Figure 23 illustrates a small part of a method that reads a string from a BufferedReader and parses this string to a `double` value that then is returned. The sequence diagram mainly shows the implementation of `try catch finally` statements and can be seen as a whole in Appendix J.



: IOClass
in : BufferedReader
: NumberFormat
readDouble(in : BufferedReader) : double
1.1. : exitVar : int := 1
1.2. : while[true]
1.2.1. : try
1.2.1.1. : s := readLine() : String
1.2.1.2. : nf := getInstance() : NumberFormat
1.2.1.3. : parsedNumber := parse(s : String) : Number
1.2.1.4. : doubleValue := doubleValue() : double
1.2.1.5. : return doubleValue
1.2.2. : catch[ParseException pe]
1.2.2.1. : System.out.println("Invalid number...try again!")
1.2.3. : catch[IOException ie]
1.2.3.1. : printStackTra
1.2.3.2. : exit(exitVar : int) : voi

Figure 23. Part of a sequence diagram based on a read double algorithm [29]

```
public static double
readDouble(BufferedReader in) {
  int exitVar = 0;
  String s = "";
  NumberFormat nf = null;
  Number parsedNumber = null;
  double doubleValue = 0;
  exitVar = 1;

  while(true) {
    try {
      s = in.readLine();
      nf =
      NumberFormat.getInstance();
      parsedNumber = nf.parse(s);
      doubleValue =
      parsedNumber.doubleValue();
      return doubleValue;
    } catch(ParseException pe) {
      System.out.println("Invalid
      number...try again!");
    } catch(IOException ie) {
      ie.printStackTrace();
      System.exit(exitVar);
    }
  }
}
```

Figure 24 illustrates a small part of an algorithm of a game called Twenty-One. The sequence diagram mainly shows how `while` loops and `if/else` statements can be implemented. The sequence diagram can be seen as a whole in Appendix K.

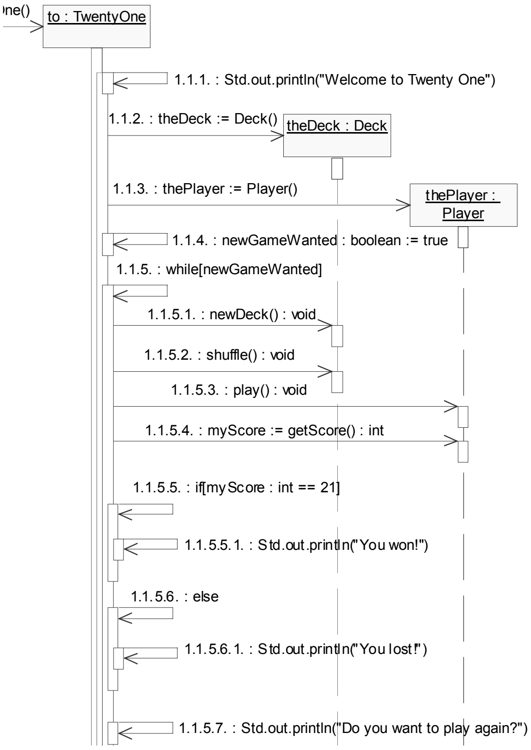

ne()
to : TwentyOne
1.1.1. : Std.out.println("Welcome to Twenty One")
1.1.2. : theDeck := Deck()
theDeck : Deck
1.1.3. : thePlayer := Player()
thePlayer : Player
1.1.4. : newGameWanted : boolean := true
1.1.5. : while[newGameWanted]
1.1.5.1. : newDeck() : void
1.1.5.2. : shuffle() : void
1.1.5.3. : play() : void
1.1.5.4. : myScore := getScore() : int
1.1.5.5. : if[myScore : int == 21]
1.1.5.5.1. : Std.out.println("You won!")
1.1.5.6. : else
1.1.5.6.1. : Std.out.println("You lost!")
1.1.5.7. : Std.out.println("Do you want to play again?")

Figure 24. Part of a sequence diagram based on a twenty-one game algorithm [29]

```
public TwentyOne() {
  Deck theDeck = null;
  Player thePlayer = null;
  boolean newGameWanted = false;
  int myScore = 0;
  String answer = "";
  boolean boolAnswer = false;

  Std.out.println("Welcome to
  Twenty One");
  theDeck = new Deck();
  thePlayer = new Player();
  newGameWanted = true;

  while(newGameWanted) {
    theDeck.newDeck();
    theDeck.shuffle();
    thePlayer.play();
    myScore =
    thePlayer.getScore();

    if(myScore == 21) {
      Std.out.println("You won");
    } else {
      Std.out.println("You lost");
    }

    Std.out.println("Do you want
    to play again?");
    answer = Std.in.readline();
    boolAnswer = false;
    boolAnswer =
    answer.equals("yes");

    if(boolAnswer == true) {
      newGameWanted = true;
    } else {
      newGameWanted = false;
    }
  }
}

public static void
main(String[] args) {
  TwentyOne to = null;
  to = new TwentyOne();
}
```

7. Discussion

For a forward engineering software package to be useful, it must not require interaction diagrams to be complex and unnecessary large. It must not limit the diagram designer in the illustration of both basic and complex programming language constructs and statements.

The four sequence diagrams presented in the previous chapter illustrate the usage of CircleUML with more extensive interaction diagrams than the interaction diagram examples presented in chapter 5.5. For all sequence diagrams in the previous chapter the number of messages corresponds very well to the number of resulting lines of code if one excludes the lines of code for variable declarations. If one considers that `if/else` statements, `for/while/do while` statements and so on require a message to be drawn just to express the condition or just to express the type of statement, then CircleUML does not require one to draw more messages than the number of statements in the resulting source code. This proves CircleUML to not require interaction diagrams to be larger than the resulting code, in the aspect of the number of messages and the number of resulting lines of source code.

When using a forward engineering software package the main reason for using it is to save the time it otherwise would have taken to translate the diagrams into source code. For the generated source code to be valuable, it must be of such quality that one does not need to modify it after it has been generated. As an example, the source code generated from Figure 21 is a perfectly useful file copy algorithm that does not need any post generation modifications. This source code is therefore valuable and the use of CircleUML in this case has been beneficial. This implies that the interaction diagram would have been created even if a forward engineering software package had not been used. It can only add value to the development process if it with little or no effort can either save time or produce valuable artefacts.

CircleUML cannot recognize all kinds of interaction diagram messages. For the interaction diagram designer to express programming language specific constructs, CircleUML must be able to allow a certain amount of flexibility. This flexibility can be to allow the designer to write a message that in a programming language can be much more compactly expressed than as interaction diagram messages that CircleUML can recognize. Such a statement can be like the following message that CircleUML handles as an unknown statement.

```
1.1. : BufferedReader myIn = new
BufferedReader(new
InputStreamReader(System.in))
```

Alternatively, it can be broken into two separate messages that CircleUML can recognize and handle as typical constructor calls, as in figure 21.

```
1.1. : theISR :=
InputStreamReader(System.in :
InputStream)
1.2. : myIn :=
BufferedReader(theISR :
InputStreamReader)
```

This illustrates that interaction diagrams that work with CircleUML does not limit the designer in expressing compact programming language constructs. One can choose to either split compact statement into separate messages or express them as one single message with the possibility of making the interaction diagram smaller and perhaps a little more difficult to read and comprehend.

If interaction diagrams are created during the design phase and details about the algorithm or algorithms are expressed then not getting more out of the diagrams than just a documentation artefact is not optimal. If details were expressed then generating code from the interaction diagrams would add value to the diagrams.

8. Conclusions

This report has dealt with three problems concerned with forward engineering from interaction diagrams in general and forward engineering tools such as CircleUML.

The first problem or question was whether it can be useful to automatically forward engineer interaction diagrams. It can definitely be useful to use a forward engineering software package for interaction diagrams when developing software. This was shown in the results chapter where different algorithms were generated into source code that needed no post generation modifications. If interaction diagrams are created during the design phase of a software development project and detailed algorithms are expressed in the diagrams then the little effort it requires to create the diagrams to work with CircleUML is definitely worth the work. The source code that CircleUML will generate would otherwise have been written by a programmer. If interaction diagrams would not had been created then creating CircleUML supported diagrams and generate source code from them would probably not be worth the work if not the diagrams could be of value as documentation artefacts.

The second problem or question was whether a forward engineering software package for interaction diagrams can work with available UML modelling tools without any adjustments to them or added diagram creation complexity. As far as diagram creation complexity is concerned, CircleUML has proven that diagrams do not have to be more complex than the programming language constructs or statements that one is expressing in the interaction diagrams. In all interaction diagram examples, presented throughout this report and in the appendices no adjustments or extra features have had to be made or added to the UML modelling tools used for their creation. This proves that CircleUML only requires the UML modelling tool to allow the diagram designer to set message expressions for the different messages. To definitely prove that CircleUML can work with not only one modelling tool, CircleUML has been implemented with support for both Rational Rose and Poseidon for UML.

The third problem or question was whether XMI could help make such a tool independent of UML modelling tools. This has been analysed in chapter 4.6 about UML modelling tool support. The answer to this question is that XMI can make such a tool independent but with a high possibility of not being able to support a wide range of UML modelling tools. The reason for this is that some tools differ in their way of storing interaction diagrams in XMI files and some vendors have made their own additions to the XMI specification.

9. Future work

The research in the area of forward engineering interaction diagrams could benefit from studies on how UML and XMI can be used in a general way. Existing concepts such as messages in an interaction diagram, cannot as of the time of writing provide support for expressing the type of iteration or whether a condition is actually an else if statement following an if statement or a new if statement. To be able to use existing concepts to express such statements one may have to deviate from the intended meaning of an interaction diagram message as needed in CircleUML.

CircleUML is restricted in its functionality in many ways. Removing these limitations such as the number of supported parameters can make CircleUML even more useful as a forward engineering software package. It can then even further prove the immediate usage of a forward engineering software package for interaction diagrams. For example, implementing support for concurrency can increase the usage of CircleUML to be able to handle interaction diagrams expressing situations where threads are involved.

The range of supported UML modelling tools can be increased and help make CircleUML available for developers that use other tools than Rational Rose and Poseidon for UML.

If there will be a specification for how graphical representations of UML diagrams can be expressed in an XMI document in a future version of UML, then CircleUML can become not only a forward engineering software package but also a reverse engineering software package. This could make CircleUML capable of providing round-trip engineering for interaction diagrams created in a number of supported UML modelling tools.

References

[1] Sommerville, I., "Software Engineering," 6th ed. Harlow : Addison-Wesley, 2001, pp 8-55.

[2] The CircleUML Project Homepage. [Online], Available: https://sourceforge.net/projects/circleuml/

[3] Fowler, M., "Is Design Dead?," *Software Development*, vol 9, No 4, pp 43-46, April. 2001.

[4] Larman, C., "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process," 2nd ed., Upper Saddle River: Prentice Hall PTR, 2002, pp. 197-213.

[5] J. Kern. (2001, Oct). Sequence Diagram Generation : Effective Use of Options. TogetherSoft Corp. Raleigh, North Carolina. [Online], Available: http://www.togethersoft.com/services/publications/white_papers/effective_use_of_sequence_diagram_generation.htm

[6] T. Systä. (2000, May). Static and Dynamic Reverse Engineering Techniques for Java Software Systems, University of Tampere. Tampere, Finland. [Online], Available: http://acta.uta.fi/pdf/951-44-4811-1.pdf

[7] Borland Completes Acquisition of TogetherSoft Press Release. [Online], Available: http://www.borland.com/news/press_releases/2003/01_15_03_borland_completes_acquisition_of_togethersoft.html

[8] Poseidon for UML Homepage. [Online], Available: http://www.gentleware.com/products/index.php3

[9] Rational Software - Rational Rose Add-ins. [Online], Available: http://www.rational.com/support/downloadcenter/addins/rose/index.jsp?SMSESSION=NO

[10] Birbeck, M., Ducket, J., Gauti Gudmunsson, O., Kobak, P., Lenz, E., Livingstone, S., Marcus, D., Mohr, S., Ozu, N., Pinnock, J., Visco, K., Watt, A., Williams, K., Zaev, Z., "Professional XML," 2nd ed. Birmingham: Wrox Press, 2001, pp 671-722.

[11] M. Voelter. (2003, Apr). A Catalog of Patterns for Program Generation. [Online], Available: http://www.voelter.de/data/pub/ProgramGeneration.pdf

[12] U. Kelter, M. Monecke, M. Schild. (2003, Feb). Do we need 'agile' Software Development Tools. [Online], Available: http://citeseer.nj.nec.com/551041.html

[13] Ahmed, K., Ayers, D., Birbeck, M., Cousins, J., Dodds, D., Lubell, J., Nic, M., Rivers-Moore, D., Watt, A., Worden, R., Wrightson, A., "Professional XML Meta Data." Birmingham: Wrox Press, 2001, pp. 25-60.

[14] Jasnowski, M., "Java, XML, and Web Services Bible." New York: Hungry Minds, 2002, pp. 91-382.

[15] W3C. (2000, Nov). Document Object Model (DOM) Level 2 Core Specificaiton. [Online], Available: http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/DOM2-Core.pdf

[16] The Castor Project Homepage. [Online], Available: http://castor.exolab.org/

[17] The Java Architecture for XML Binding (JAXB) Homepage. [Online], Available: http://java.sun.com/xml/jaxb/

[18] The Zeus Project Homepage. [Online], Available: http://zeus.enhydra.org/

[19] OMG. (2002, Jan). OMG XML Metadata Interchange (XMI) Specification. [Online], Available: http://www.omg.org/docs/formal/02-01-01.pdf

[20] The Meta Integration Model Bridge (MIMB) Homepage. [Online], Available: http://www.metaintegration.net/Products/MIMB/index.html

[21] Kobryn, C., "UML 2001: A Standardization Odyssey," *Communications of the ACM*, vol. 42, No. 10, Oct. 1999.

[22] OMG. (2003, Jan). UML 2.0 Diagram Interchange, Second (Final) Revised Submission in Response to OMG Document ad/2002-12-20. [Online], Available: http://www.omg.org/docs/ad/02-12-20.pdf

[23] UML 1.5 Specification. [Online], Available: http://www.omg.org/cgi-bin/apps/doc?formal/03-03-01.pdf

[24] Enterprise APIs included with J2SE v1.4. [Online], Available: http://access1.sun.com/SRDs/access1_srds.html

[25] Xalan-Java. [Online], Available: http://xml.apache.org/xalan-j/

[26] The ArgoUML Project Homepage. [Online], Available: http://argouml.tigris.org/

[27] Gang of Four. [Online], Available: http://hillside.net/patterns/gang_of_four.htm

[28] Gamma, E., Helm, R., Johnsson, R., Vlissides, J., "Design Patterns - Elements of Reusable Object-Oriented Software." Reading: Addison-Wesley, 1995, pp. 127-134, 293-313.

[29] Skansholm, J., "Java direkt." Lund: Studentlitteratur, 1999, pp. 97, 255-256, 270.

Appendix A
CircleUML use cases

ChangeLanguage use case

Actors:
    Developer
Description:
    The Developer will be able to change the language for all menus, messages and log information in the GUI
Pre-conditions:
    No pre-conditions
Post-conditions:
    CircleUML has changed the language for menus, messages and log information in the GUI
Main flow:
    1. The Developer chooses a language
    2. CircleUML changes the language for all menus, messages and log information
Alternative flows:
    No alternative flows

ParseMessages use case

Actors:
    Developer
Description
    The Developer will be able to open an XMI-file and let circleUML parse the interaction messages in the file
Pre-conditions:
    No pre-conditions
Post-conditions:
    CircleUML has generated a sort of pseudo code stored in memory
    CircleUML has generated log information presented in the GUI
Main flow:
    1. The Developer opens an XMI-file
    2. CircleUML extracts the interaction messages from the XMI file
    3. CircleUML performs pattern matching to find out if the messages can be parsed and in what way they must be parsed
    4. CircleUML generates a set of pseudo code objects that correspond to the statements described in the interaction messages
    5. CircleUML searches for the files that the code will be inserted into and displays them to the User
Alternative flows:
    5a. CircleUML cannot locate all needed files and lets the Developer locate the files manually
    6a. CircleUML aborts until all needed files are located

GenerateCode use case

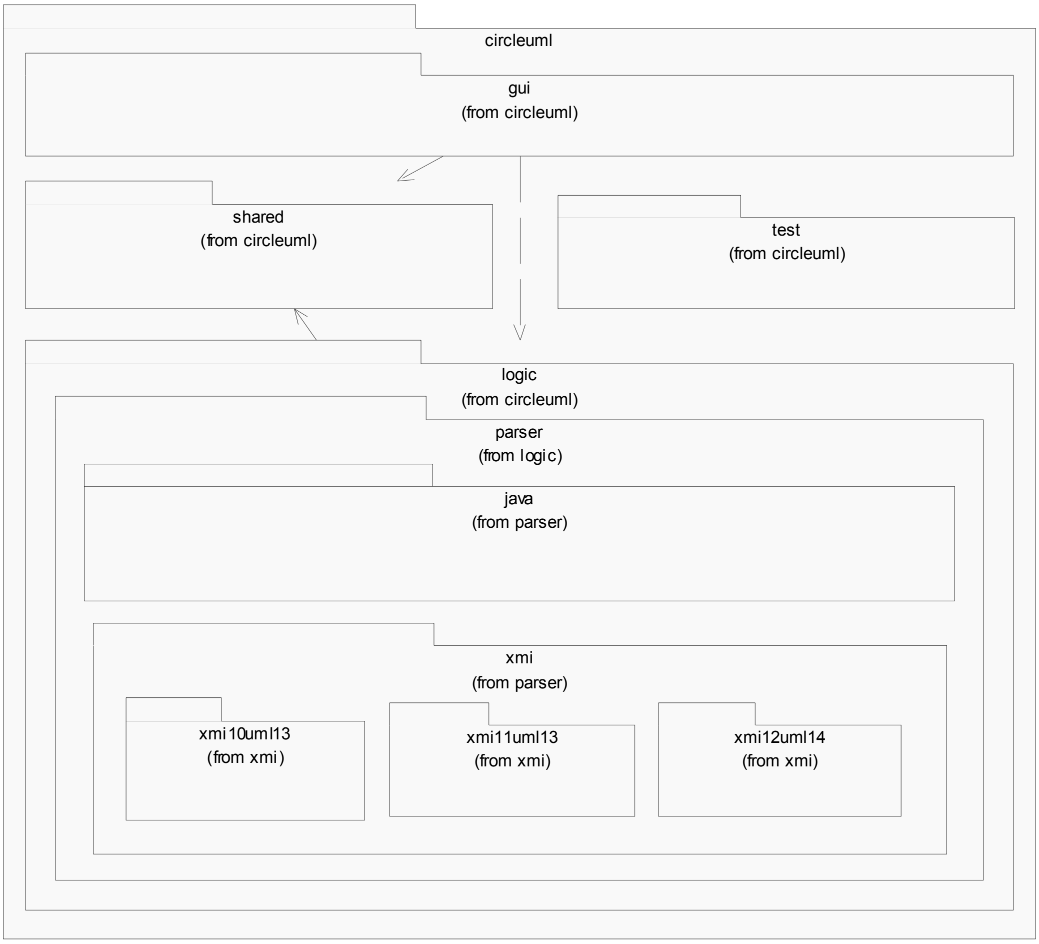
Actor:
    Developer
Description:
    The Developer will be able to generate code from an XMI export into Java source code
Pre-conditions:
    CircleUML has generated pseudo code objects from the opened XMI file
Post-conditions:
    CircleUML has generated log information presented in the GUI
    CircleUML has inserted the generated source code into the files
Main flow:
    1. The Developer chooses to generate source code

2. The collection of pseudo code objects are converted into a source code for a particular programming language
3. CircleUML extracts the information from the pseudo code objects that describe from what class and method a message is sent
4. CircleUML inserts the generated code into the correct methods in the correct files

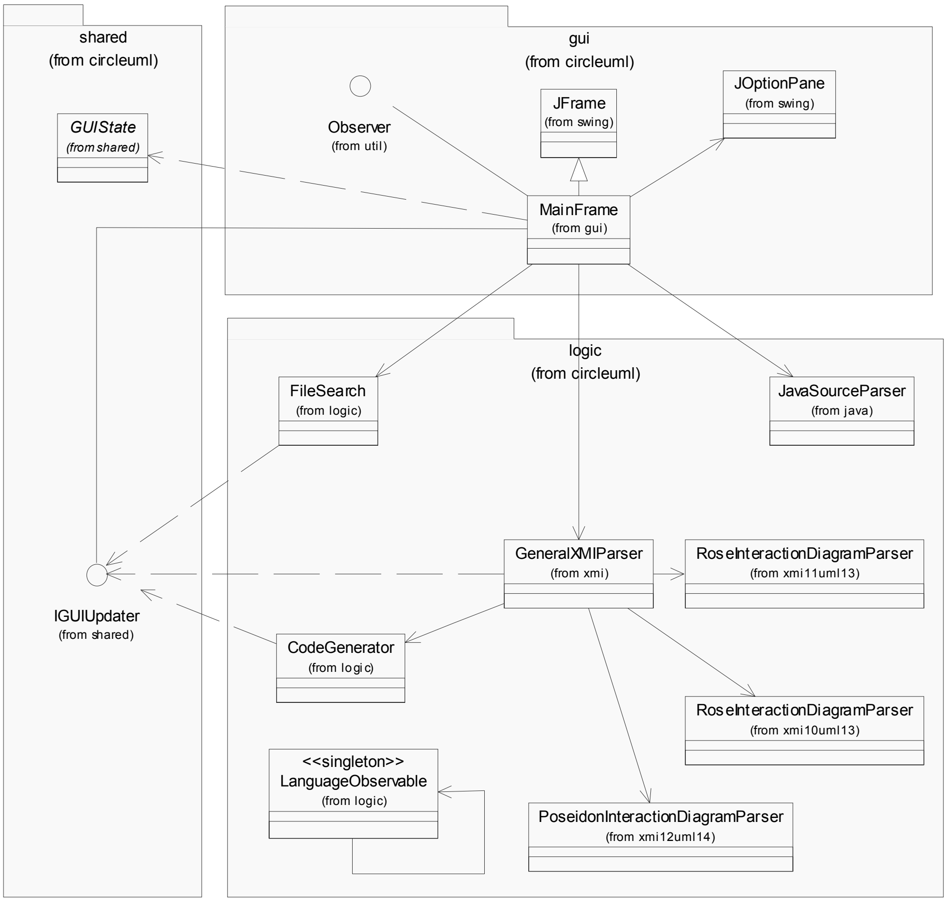
Alternative flows:

No alternative flows

Appendix B
CircleUML package hierarchy

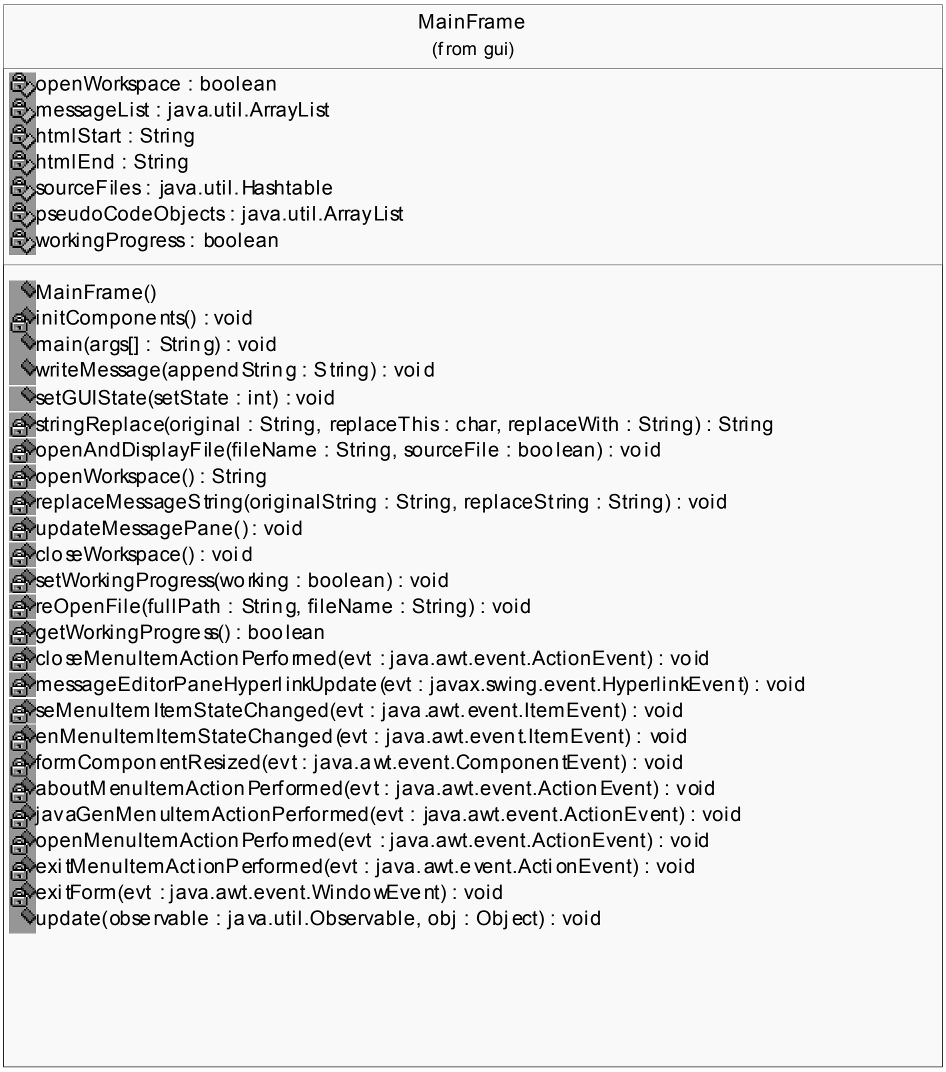


Appendix C
Non-detailed CircleUML class ciagram



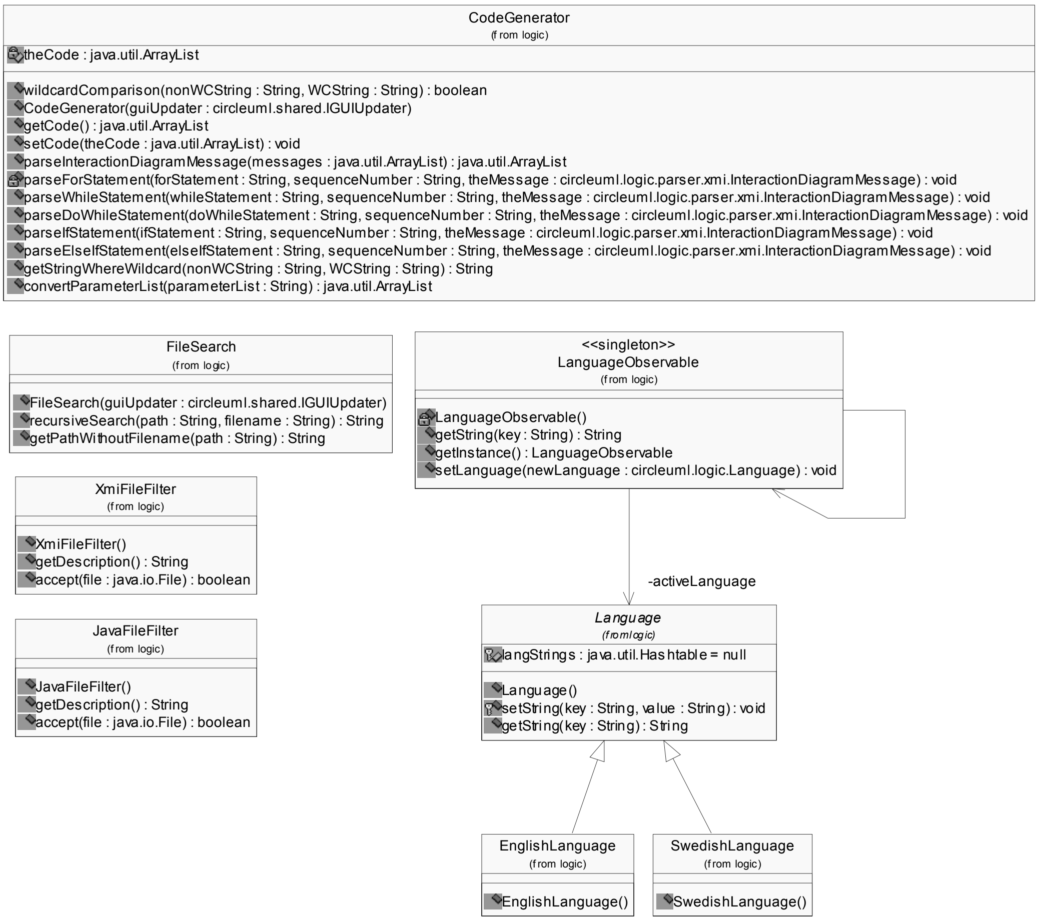
Appendix D
CircleUML Java classes and interfaces

Java classes within the circleuml.gui package.

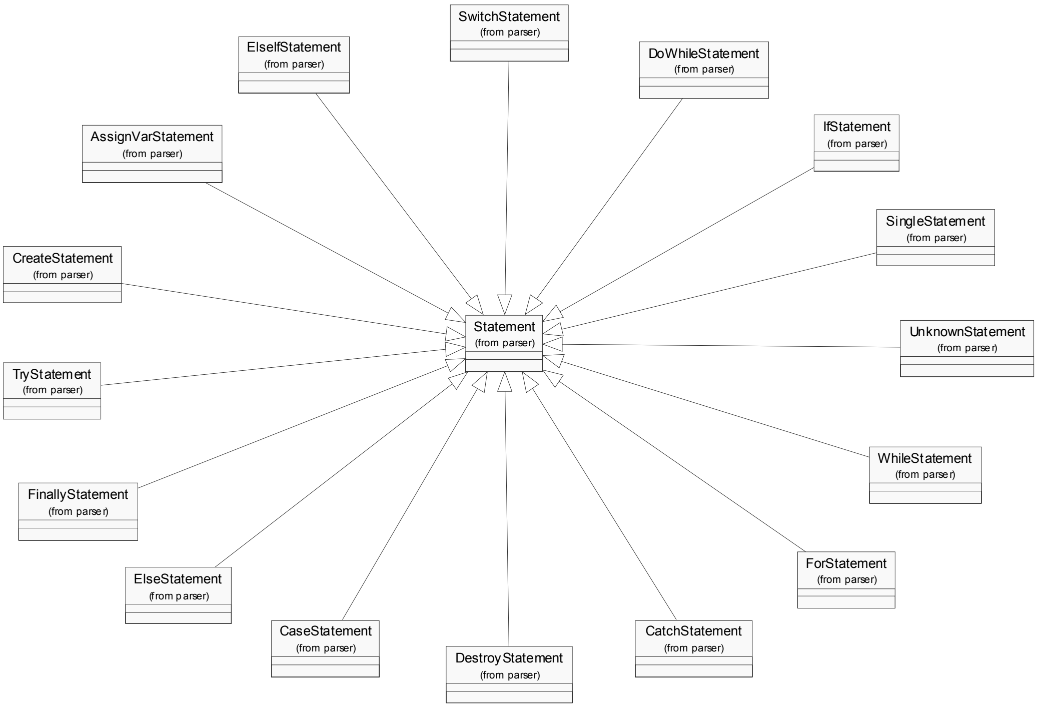


```
MainFrame
(from gui)

openWorkspace : boolean
messageList : java.util.ArrayList
htmlStart : String
htmlEnd : String
sourceFiles : java.util.Hashtable
pseudoCodeObjects : java.util.ArrayList
workingProgress : boolean

MainFrame()
initComponents() : void
main(args[] : String) : void
writeMessage(appendString : String) : void
setGUIState(setState : int) : void
stringReplace(original : String, replaceThis : char, replaceWith : String) : String
openAndDisplayFile(fileName : String, sourceFile : boolean) : void
openWorkspace() : String
replaceMessageString(originalString : String, replaceString : String) : void
updateMessagePane() : void
closeWorkspace() : void
setWorkingProgress(working : boolean) : void
reOpenFile(fullPath : String, fileName : String) : void
getWorkingProgress() : boolean
closeMenuItemActionPerformed(evt : java.awt.event.ActionEvent) : void
messageEditorPaneHyperlinkUpdate(evt : javax.swing.event.HyperlinkEvent) : void
seMenuItemItemStateChanged(evt : java.awt.event.ItemEvent) : void
enMenuItemItemStateChanged(evt : java.awt.event.ItemEvent) : void
formComponentResized(evt : java.awt.event.ComponentEvent) : void
aboutMenuItemActionPerformed(evt : java.awt.event.ActionEvent) : void
javaGenMenuItemActionPerformed(evt : java.awt.event.ActionEvent) : void
openMenuItemActionPerformed(evt : java.awt.event.ActionEvent) : void
exitMenuItemActionPerformed(evt : java.awt.event.ActionEvent) : void
exitForm(evt : java.awt.event.WindowEvent) : void
update(observable : java.util.Observable, obj : Object) : void
```

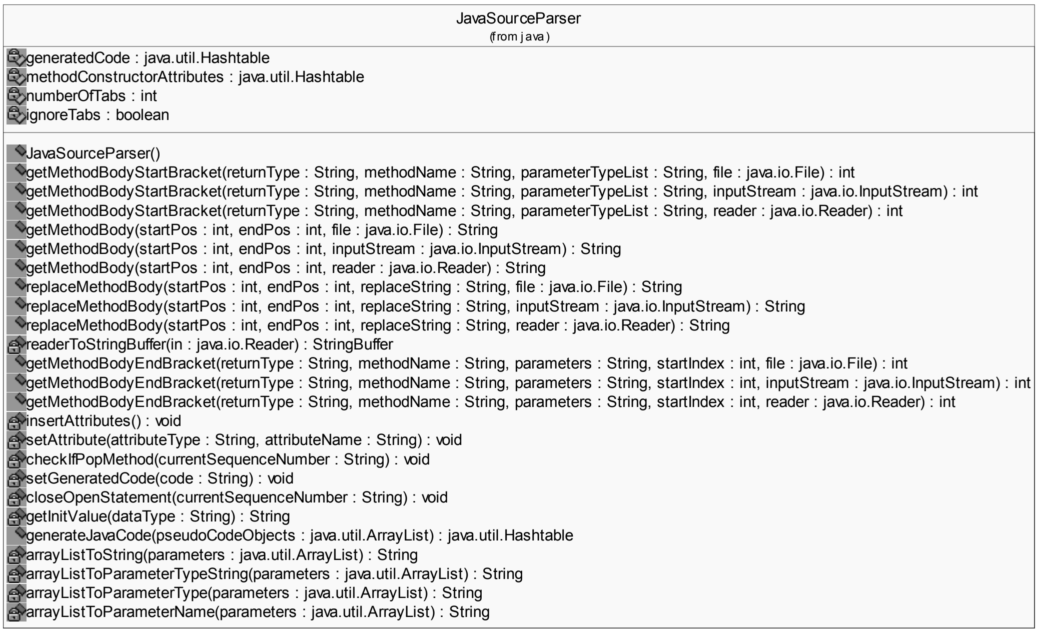
Java classes within the circleuml.logic package.


CodeGenerator
(from logic)
theCode : java.util.ArrayList
wildcardComparison(nonWCString : String, WCString : String) : boolean
CodeGenerator(guiUpdater : circleuml.shared.IGUIUpdater)
getCode() : java.util.ArrayList
setCode(theCode : java.util.ArrayList) : void
parseInteractionDiagramMessage(messages : java.util.ArrayList) : java.util.ArrayList
parseForStatement(forStatement : String, sequenceNumber : String, theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage) : void
parseWhileStatement(whileStatement : String, sequenceNumber : String, theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage) : void
parseDoWhileStatement(doWhileStatement : String, sequenceNumber : String, theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage) : void
parseIfStatement(ifStatement : String, sequenceNumber : String, theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage) : void
parseElseIfStatement(elseIfStatement : String, sequenceNumber : String, theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage) : void
getStringWhereWildcard(nonWCString : String, WCString : String) : String
convertParameterList(parameterList : String) : java.util.ArrayList
FileSearch
(from logic)
FileSearch(guiUpdater : circleuml.shared.IGUIUpdater)
recursiveSearch(path : String, filename : String) : String
getPathWithoutFilename(path : String) : String
<<singleton>>
LanguageObservable
(from logic)
LanguageObservable()
getString(key : String) : String
getInstance() : LanguageObservable
setLanguage(newLanguage : circleuml.logic.Language) : void
XmiFileFilter
(from logic)
XmiFileFilter()
getDescription() : String
accept(file : java.io.File) : boolean
-activeLanguage
JavaFileFilter
(from logic)
JavaFileFilter()
getDescription() : String
accept(file : java.io.File) : boolean
Language
(from logic)
langStrings : java.util.Hashtable = null
Language()
setString(key : String, value : String) : void
getString(key : String) : String
EnglishLanguage
(from logic)
EnglishLanguage()
SwedishLanguage
(from logic)
SwedishLanguage()

Java classes within the circleuml.logic.parser package.



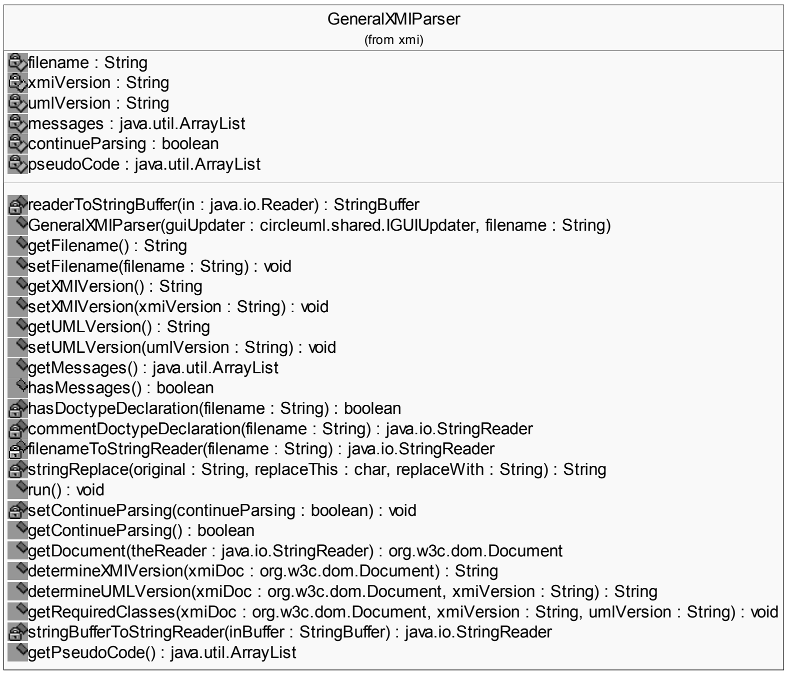
Java classes within the circleuml.logic.parser.java package.


JavaSourceParser
(from java)

generatedCode : java.util.Hashtable
methodConstructorAttributes : java.util.Hashtable
numberOfTabs : int
ignoreTabs : boolean

JavaSourceParser()
getMethodBodyStartBracket(returnType : String, methodName : String, parameterTypeList : String, file : java.io.File) : int
getMethodBodyStartBracket(returnType : String, methodName : String, parameterTypeList : String, inputStream : java.io.InputStream) : int
getMethodBodyStartBracket(returnType : String, methodName : String, parameterTypeList : String, reader : java.io.Reader) : int
getMethodBody(startPos : int, endPos : int, file : java.io.File) : String
getMethodBody(startPos : int, endPos : int, inputStream : java.io.InputStream) : String
getMethodBody(startPos : int, endPos : int, reader : java.io.Reader) : String
replaceMethodBody(startPos : int, endPos : int, replaceString : String, file : java.io.File) : String
replaceMethodBody(startPos : int, endPos : int, replaceString : String, inputStream : java.io.InputStream) : String
replaceMethodBody(startPos : int, endPos : int, replaceString : String, reader : java.io.Reader) : String
readerToStringBuffer(in : java.io.Reader) : StringBuffer
getMethodBodyEndBracket(returnType : String, methodName : String, parameters : String, startIndex : int, file : java.io.File) : int
getMethodBodyEndBracket(returnType : String, methodName : String, parameters : String, startIndex : int, inputStream : java.io.InputStream) : int
getMethodBodyEndBracket(returnType : String, methodName : String, parameters : String, startIndex : int, reader : java.io.Reader) : int
insertAttributes() : void
setAttribute(attributeType : String, attributeName : String) : void
checkIfPopMethod(currentSequenceNumber : String) : void
setGeneratedCode(code : String) : void
closeOpenStatement(currentSequenceNumber : String) : void
getInitValue(dataType : String) : String
generateJavaCode(pseudoCodeObjects : java.util.ArrayList) : java.util.Hashtable
arrayListToString(parameters : java.util.ArrayList) : String
arrayListToParameterTypeString(parameters : java.util.ArrayList) : String
arrayListToParameterType(parameters : java.util.ArrayList) : String
arrayListToParameterName(parameters : java.util.ArrayList) : String

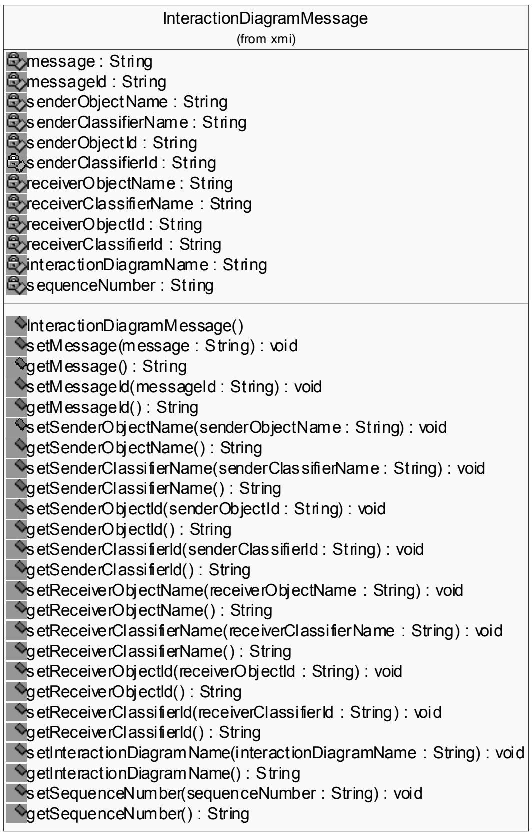
Java classes within the circleuml.logic.parser.xmi package.



GeneralXMIParser
(from xmi)

filename : String
xmiVersion : String
umlVersion : String
messages : java.util.ArrayList
continueParsing : boolean
pseudoCode : java.util.ArrayList

readerToStringBuffer(in : java.io.Reader) : StringBuffer
GeneralXMIParser(guiUpdater : circleuml.shared.IGUIUpdater, filename : String)
getFilename() : String
setFilename(filename : String) : void
getXMIVersion() : String
setXMIVersion(xmiVersion : String) : void
getUMLVersion() : String
setUMLVersion(umlVersion : String) : void
getMessages() : java.util.ArrayList
hasMessages() : boolean
hasDoctypeDeclaration(filename : String) : boolean
commentDoctypeDeclaration(filename : String) : java.io.StringReader
filenameToStringReader(filename : String) : java.io.StringReader
stringReplace(original : String, replaceThis : char, replaceWith : String) : String
run() : void
setContinueParsing(continueParsing : boolean) : void
getContinueParsing() : boolean
getDocument(theReader : java.io.StringReader) : org.w3c.dom.Document
determineXMIVersion(xmiDoc : org.w3c.dom.Document) : String
determineUMLVersion(xmiDoc : org.w3c.dom.Document, xmiVersion : String) : String
getRequiredClasses(xmiDoc : org.w3c.dom.Document, xmiVersion : String, umlVersion : String) : void
stringBufferToStringReader(inBuffer : StringBuffer) : java.io.StringReader
getPseudoCode() : java.util.ArrayList

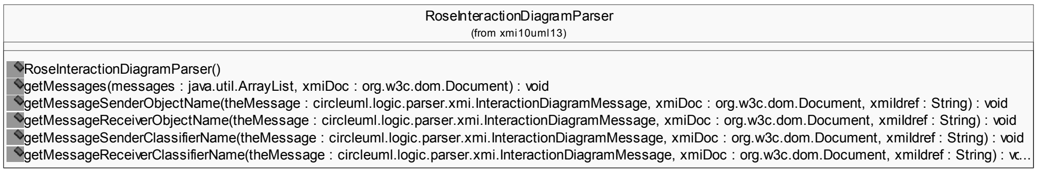


InteractionDiagramMessage
(from xmi)

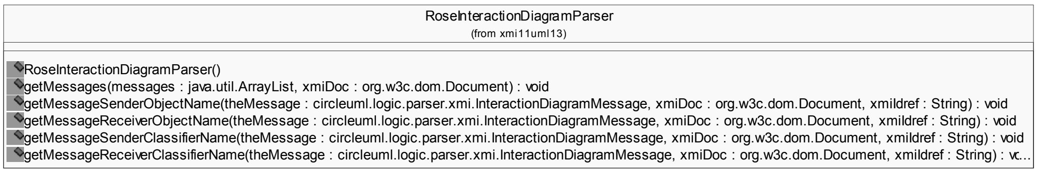
message : String
messageId : String
senderObjectName : String
senderClassifierName : String
senderObjectId : String
senderClassifierId : String
receiverObjectName : String
receiverClassifierName : String
receiverObjectId : String
receiverClassifierId : String
interactionDiagramName : String
sequenceNumber : String

InteractionDiagramMessage()
setMessage(message : String) : void
getMessage() : String
setMessageId(messageId : String) : void
getMessageId() : String
setSenderObjectName(senderObjectName : String) : void
getSenderObjectName() : String
setSenderClassifierName(senderClassifierName : String) : void
getSenderClassifierName() : String
setSenderObjectId(senderObjectId : String) : void
getSenderObjectId() : String
setSenderClassifierId(senderClassifierId : String) : void
getSenderClassifierId() : String
setReceiverObjectName(receiverObjectName : String) : void
getReceiverObjectName() : String
setReceiverClassifierName(receiverClassifierName : String) : void
getReceiverClassifierName() : String
setReceiverObjectId(receiverObjectId : String) : void
getReceiverObjectId() : String
setReceiverClassifierId(receiverClassifierId : String) : void
getReceiverClassifierId() : String
setInteractionDiagramName(interactionDiagramName : String) : void
getInteractionDiagramName() : String
setSequenceNumber(sequenceNumber : String) : void
getSequenceNumber() : String

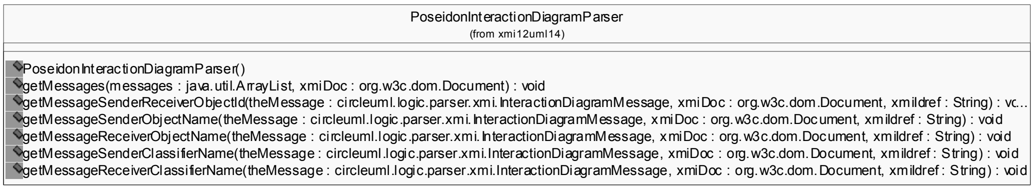
Java classes within the circleuml.logic.parser.xmi.xmi10uml13 package.


RoseInteractionDiagramParser
(from xmi10uml13)
RoseInteractionDiagramParser()
getMessages(messages : java.util.ArrayList, xmiDoc : org.w3c.dom.Document) : void
getMessageSenderObjectName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageReceiverObjectName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageSenderClassifierName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageReceiverClassifierName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : vo...

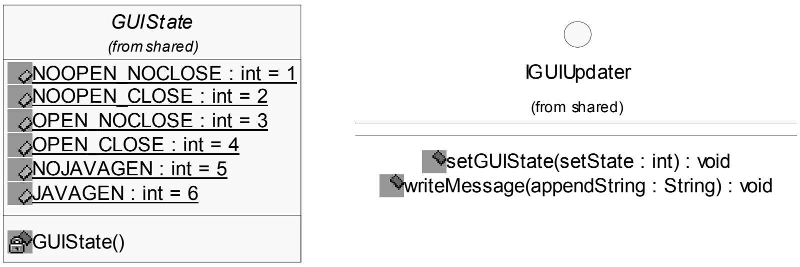
Java classes within the circleuml.logic.parser.xmi.xmi11uml13 package.


RoseInteractionDiagramParser
(from xmi11uml13)
RoseInteractionDiagramParser()
getMessages(messages : java.util.ArrayList, xmiDoc : org.w3c.dom.Document) : void
getMessageSenderObjectName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageReceiverObjectName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageSenderClassifierName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageReceiverClassifierName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : vo...

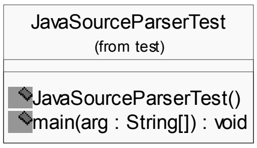
Java classes within the circleuml.logic.parser.xmi.xmi12uml14 package.


PoseidonInteractionDiagramParser
(from xmi12uml14)
PoseidonInteractionDiagramParser()
getMessages(messages : java.util.ArrayList, xmiDoc : org.w3c.dom.Document) : void
getMessageSenderReceiverObjectId(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : vo...
getMessageSenderObjectName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageReceiverObjectName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageSenderClassifierName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void
getMessageReceiverClassifierName(theMessage : circleuml.logic.parser.xmi.InteractionDiagramMessage, xmiDoc : org.w3c.dom.Document, xmiIdref : String) : void

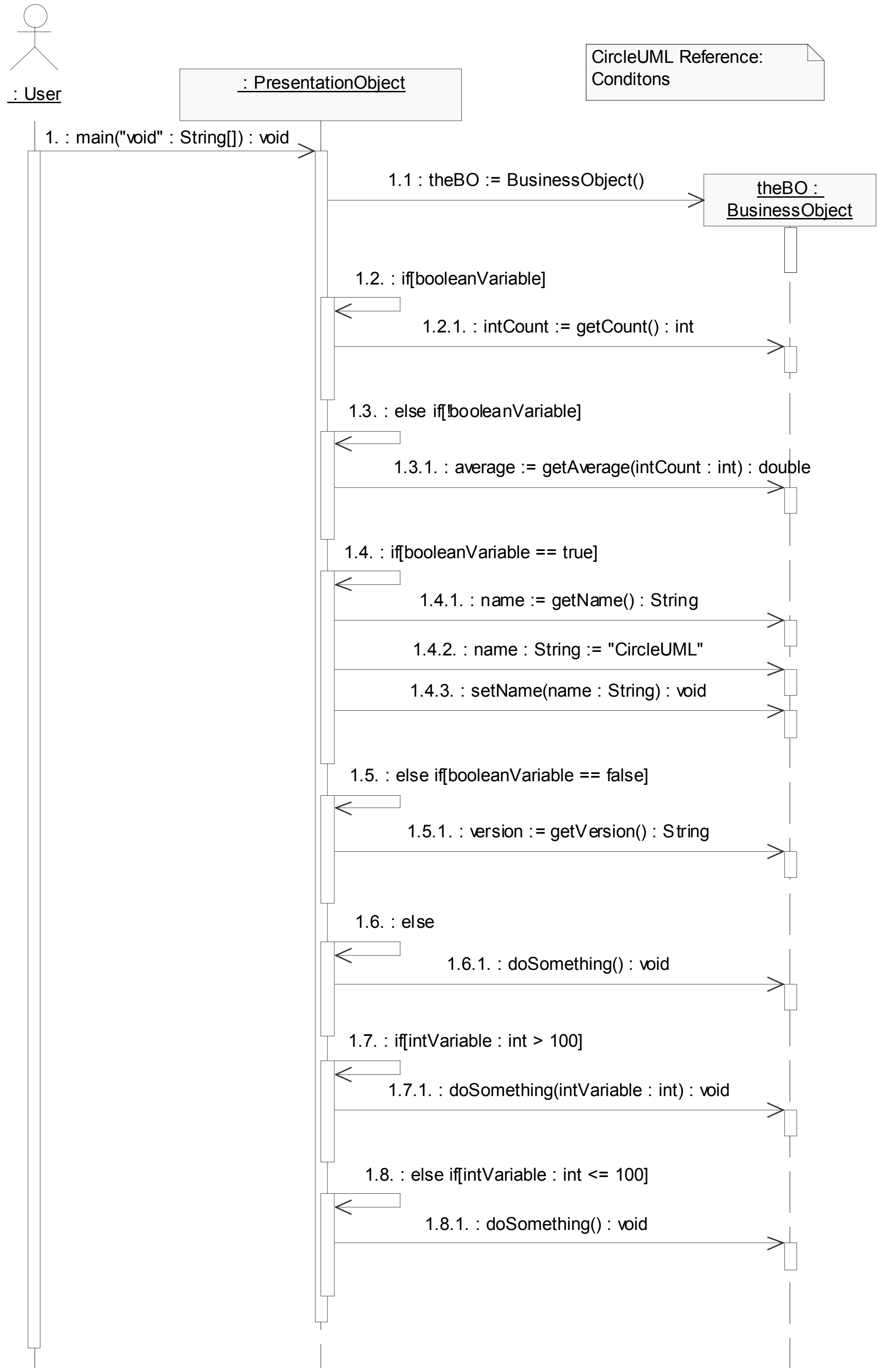
Java classes and interfaces within the circleuml.shared package.

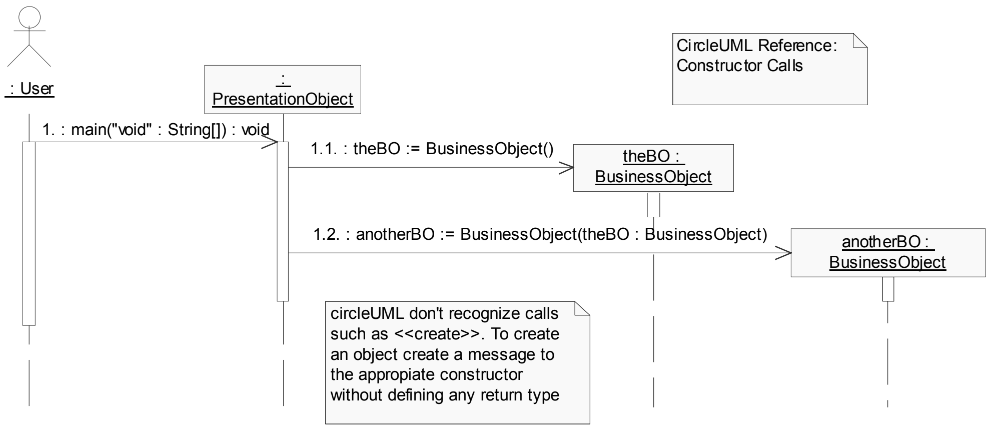


Java classes within the circleuml.test package.



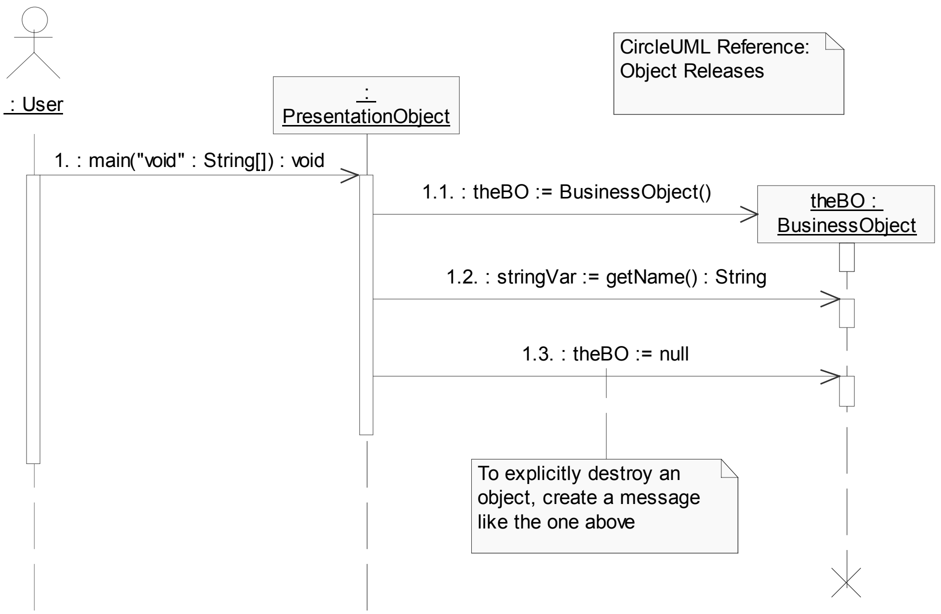
Appendix E
CircleUML reference expressed in Rational Rose sequence diagrams


: User
: PresentationObject
CircleUML Reference:
Conditons
1. : main("void" : String[]) : void
1.1 : theBO := BusinessObject()
theBO : BusinessObject
1.2. : if[booleanVariable]
1.2.1. : intCount := getCount() : int
1.3. : else if[!booleanVariable]
1.3.1. : average := getAverage(intCount : int) : double
1.4. : if[booleanVariable == true]
1.4.1. : name := getName() : String
1.4.2. : name : String := "CircleUML"
1.4.3. : setName(name : String) : void
1.5. : else if[booleanVariable == false]
1.5.1. : version := getVersion() : String
1.6. : else
1.6.1. : doSomething() : void
1.7. : if[intVariable : int > 100]
1.7.1. : doSomething(intVariable : int) : void
1.8. : else if[intVariable : int <= 100]
1.8.1. : doSomething() : void


: User
:
PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO :
BusinessObject
1.2. : anotherBO := BusinessObject(theBO : BusinessObject)
anotherBO :
BusinessObject

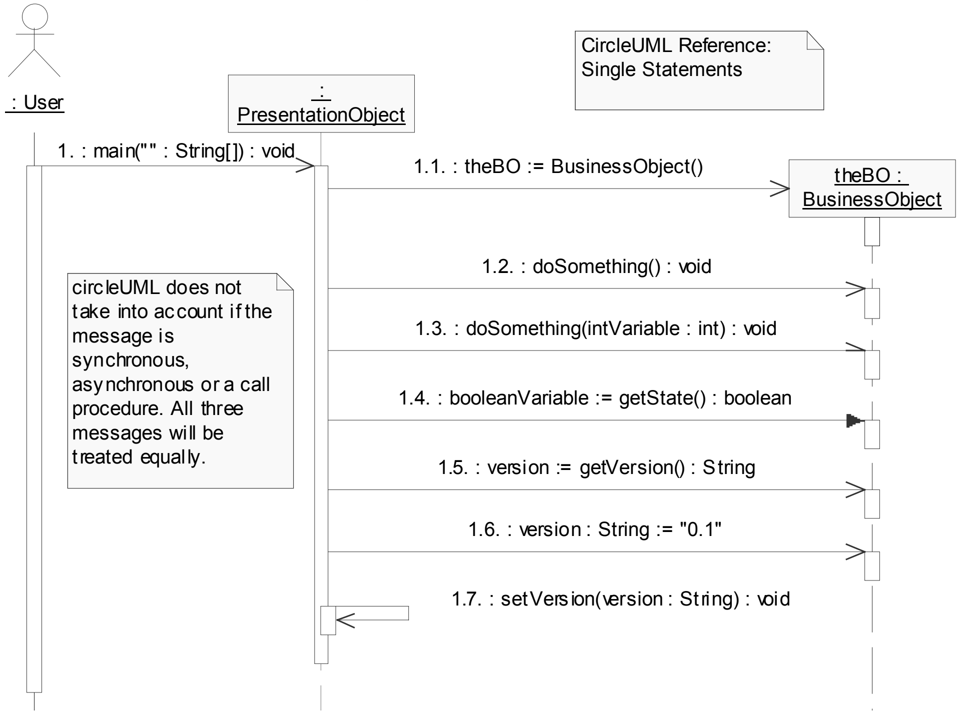
CircleUML Reference:
Constructor Calls

circleUML don't recognize calls such as <<create>>. To create an object create a message to the appropiate constructor without defining any return type


: User
:
PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO :
BusinessObject
1.2. : stringVar := getName() : String
1.3. : theBO := null

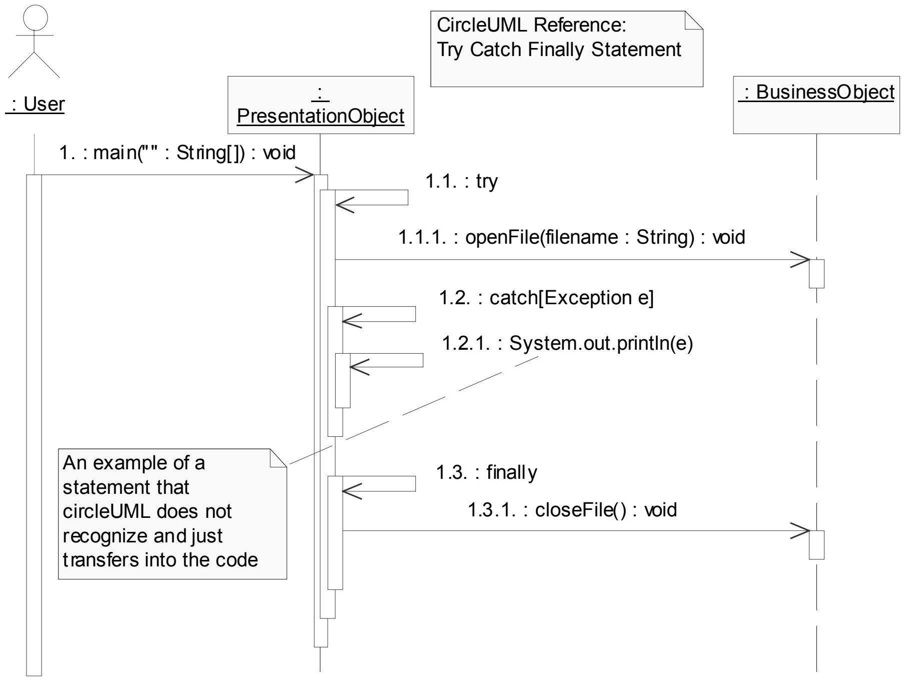
CircleUML Reference:
Object Releases

To explicitly destroy an object, create a message like the one above


: User
:
PresentationObject
1. : main("" : String[]) : void
1.1. : theBO := BusinessObject()
theBO :
BusinessObject
1.2. : doSomething() : void
1.3. : doSomething(intVariable : int) : void
1.4. : booleanVariable := getState() : boolean
1.5. : version := getVersion() : String
1.6. : version : String := "0.1"
1.7. : setVersion(version : String) : void

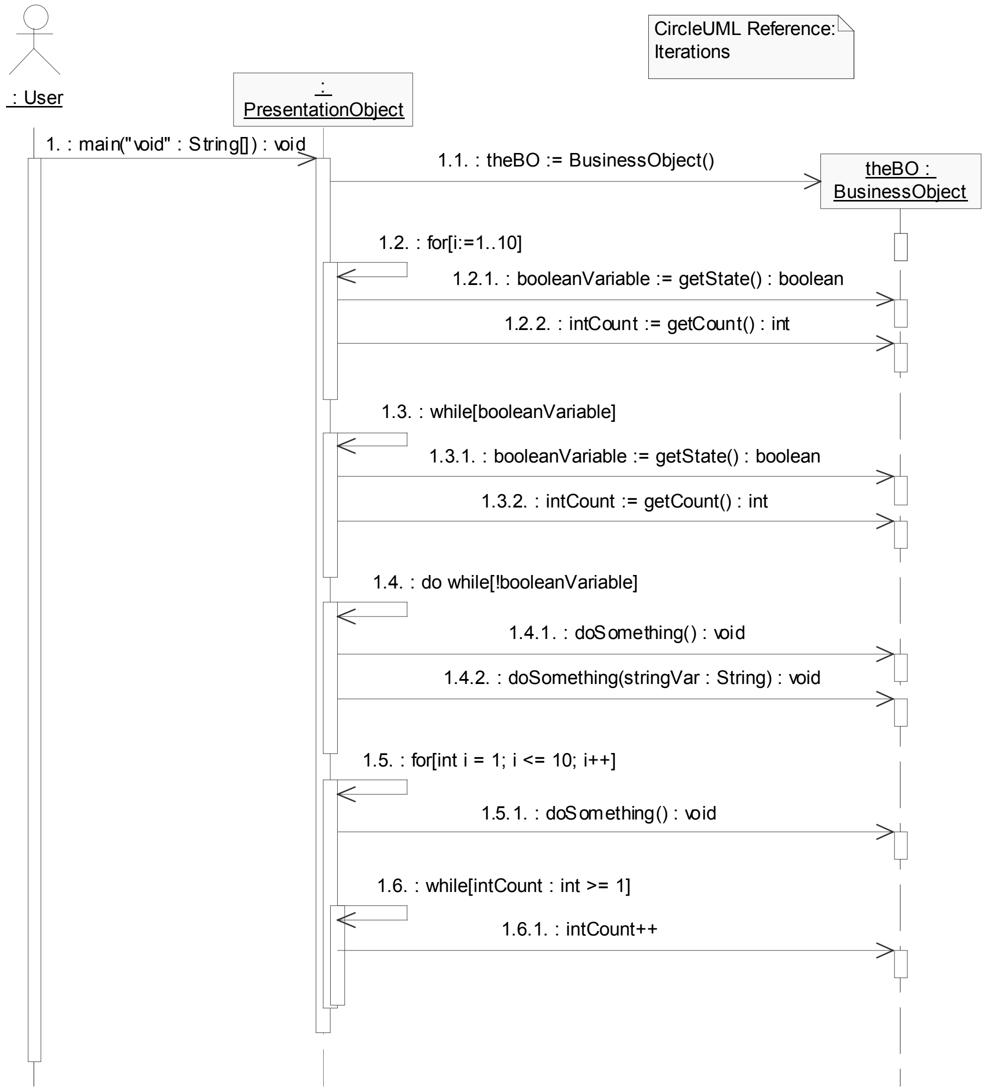
CircleUML Reference:
Single Statements

circleUML does not take into account if the message is synchronous, asynchronous or a call procedure. All three messages will be treated equally.

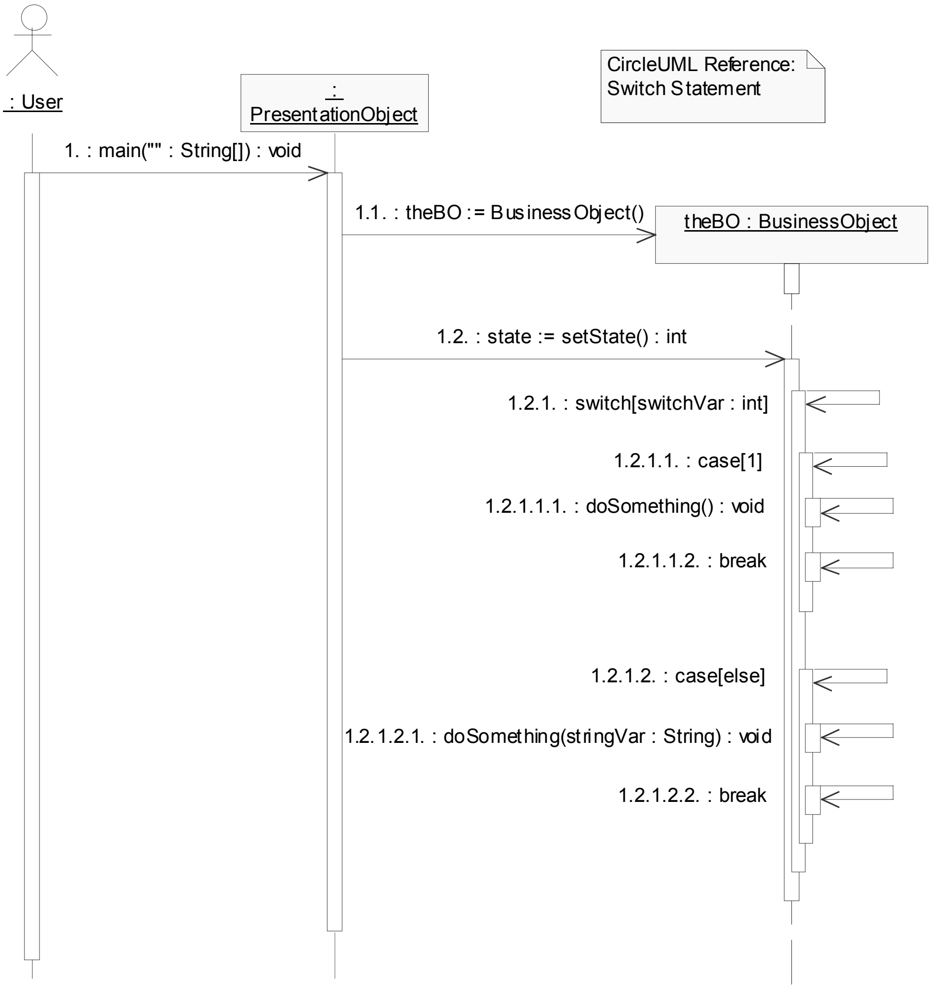

: User
: PresentationObject
: BusinessObject
1. : main("" : String[]) : void
1.1. : try
1.1.1. : openFile(filename : String) : void
1.2. : catch[Exception e]
1.2.1. : System.out.println(e)
1.3. : finally
1.3.1. : closeFile() : void

CircleUML Reference:
Try Catch Finally Statement

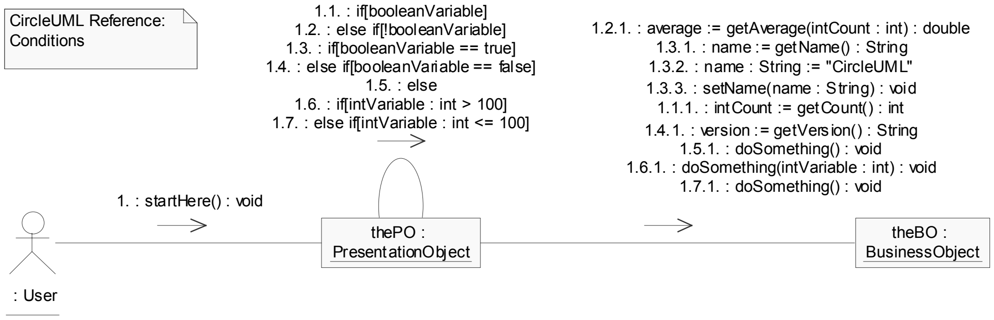
An example of a statement that circleUML does not recognize and just transfers into the code

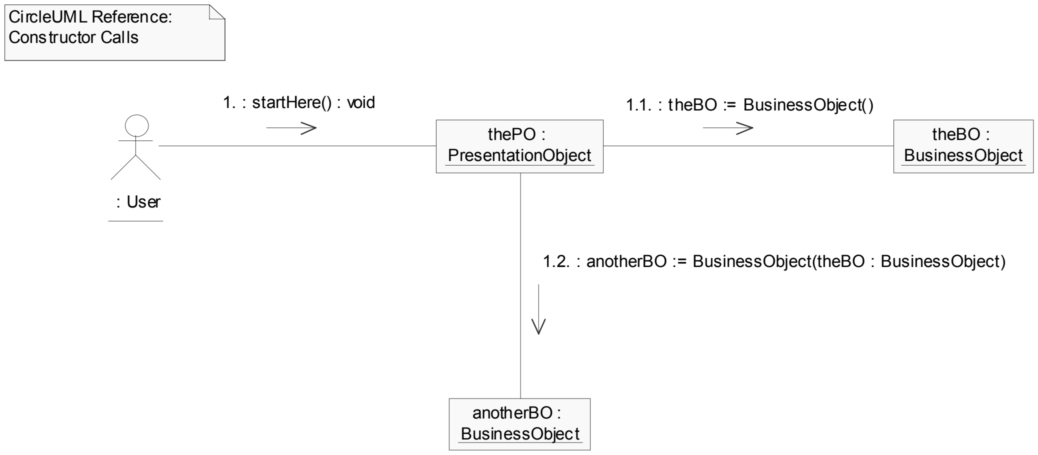

: User
: PresentationObject
theBO : BusinessObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
1.2. : for[i:=1..10]
1.2.1. : booleanVariable := getState() : boolean
1.2.2. : intCount := getCount() : int
1.3. : while[booleanVariable]
1.3.1. : booleanVariable := getState() : boolean
1.3.2. : intCount := getCount() : int
1.4. : do while[!booleanVariable]
1.4.1. : doSomething() : void
1.4.2. : doSomething(stringVar : String) : void
1.5. : for[int i = 1; i <= 10; i++]
1.5.1. : doSomething() : void
1.6. : while[intCount : int >= 1]
1.6.1. : intCount++

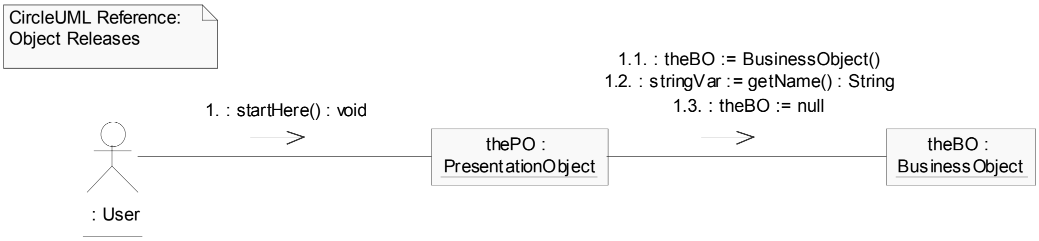
CircleUML Reference:
Iterations

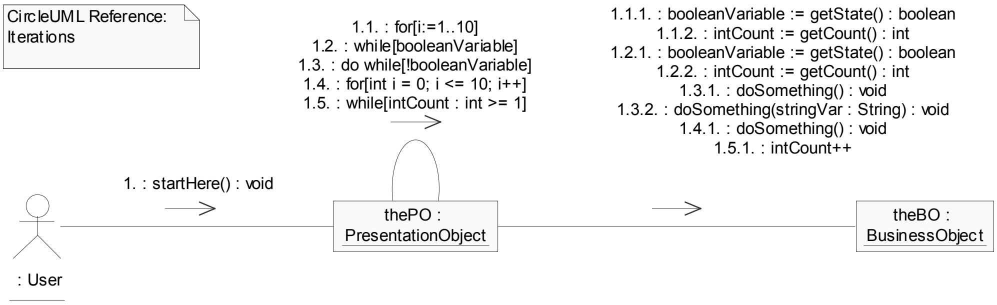

: User
:
PresentationObject
CircleUML Reference:
Switch Statement
1. : main("" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : state := setState() : int
1.2.1. : switch[switchVar : int]
1.2.1.1. : case[1]
1.2.1.1.1. : doSomething() : void
1.2.1.1.2. : break
1.2.1.2. : case[else]
1.2.1.2.1. : doSomething(stringVar : String) : void
1.2.1.2.2. : break

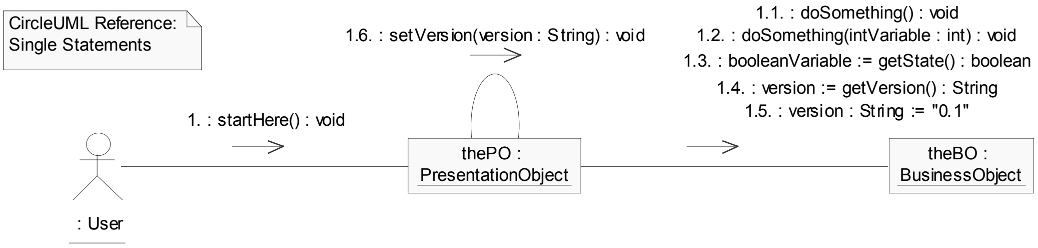
Appendix F
CircleUML reference expressed in Rational Rose collaboration diagrams

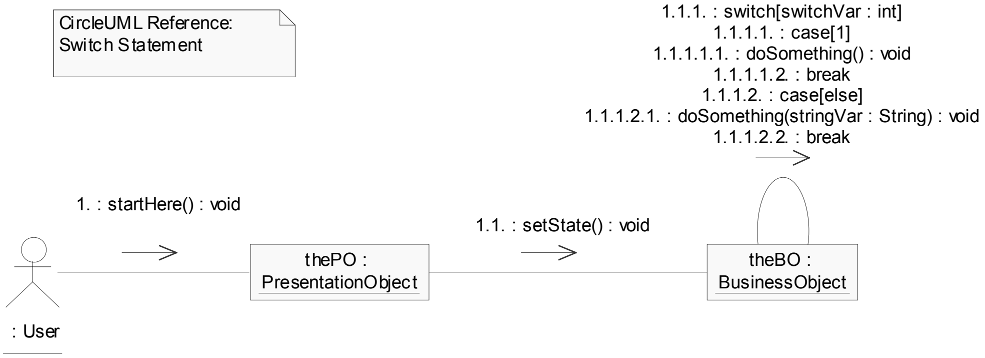

CircleUML Reference:
Conditions
1.1. : if[booleanVariable]
1.2. : else if[!booleanVariable]
1.3. : if[booleanVariable == true]
1.4. : else if[booleanVariable == false]
1.5. : else
1.6. : if[intVariable : int > 100]
1.7. : else if[intVariable : int <= 100]
1.2.1. : average := getAverage(intCount : int) : double
1.3.1. : name := getName() : String
1.3.2. : name : String := "CircleUML"
1.3.3. : setName(name : String) : void
1.1.1. : intCount := getCount() : int
1.4.1. : version := getVersion() : String
1.5.1. : doSomething() : void
1.6.1. : doSomething(intVariable : int) : void
1.7.1. : doSomething() : void
1. : startHere() : void
thePO :
PresentationObject
theBO :
BusinessObject
: User

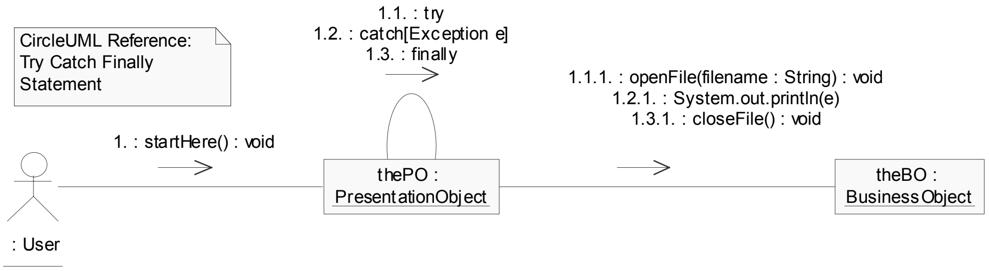




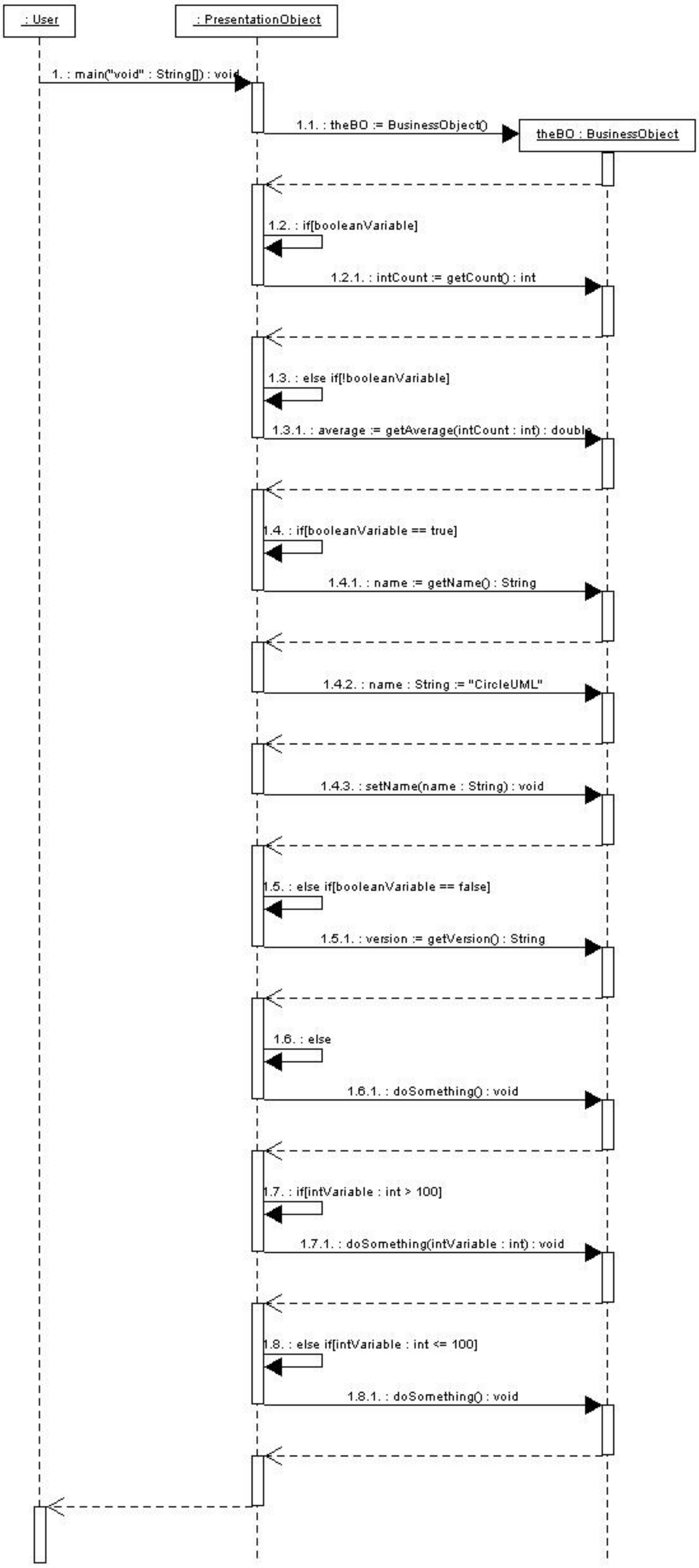

CircleUML Reference:
Iterations
1.1. : for[i:=1..10]
1.2. : while[booleanVariable]
1.3. : do while[!booleanVariable]
1.4. : for[int i = 0; i <= 10; i++]
1.5. : while[intCount : int >= 1]
1.1.1. : booleanVariable := getState() : boolean
1.1.2. : intCount := getCount() : int
1.2.1. : booleanVariable := getState() : boolean
1.2.2. : intCount := getCount() : int
1.3.1. : doSomething() : void
1.3.2. : doSomething(stringVar : String) : void
1.4.1. : doSomething() : void
1.5.1. : intCount++
1. : startHere() : void
thePO :
PresentationObject
theBO :
BusinessObject
: User

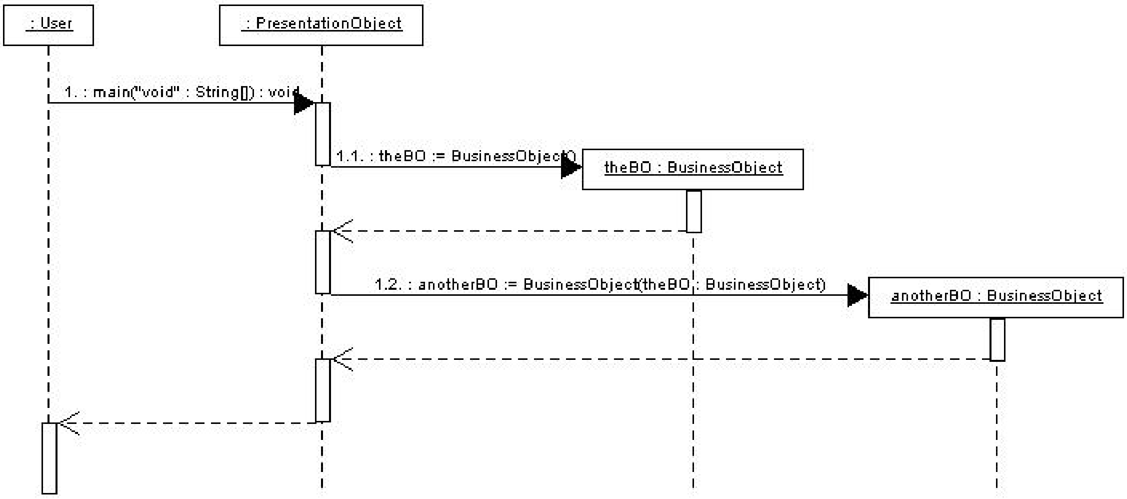

CircleUML Reference:
Single Statements
1.6. : setVersion(version : String) : void
1.1. : doSomething() : void
1.2. : doSomething(intVariable : int) : void
1.3. : booleanVariable := getState() : boolean
1.4. : version := getVersion() : String
1.5. : version : String := "0.1"
1. : startHere() : void
thePO :
PresentationObject
theBO :
BusinessObject
: User

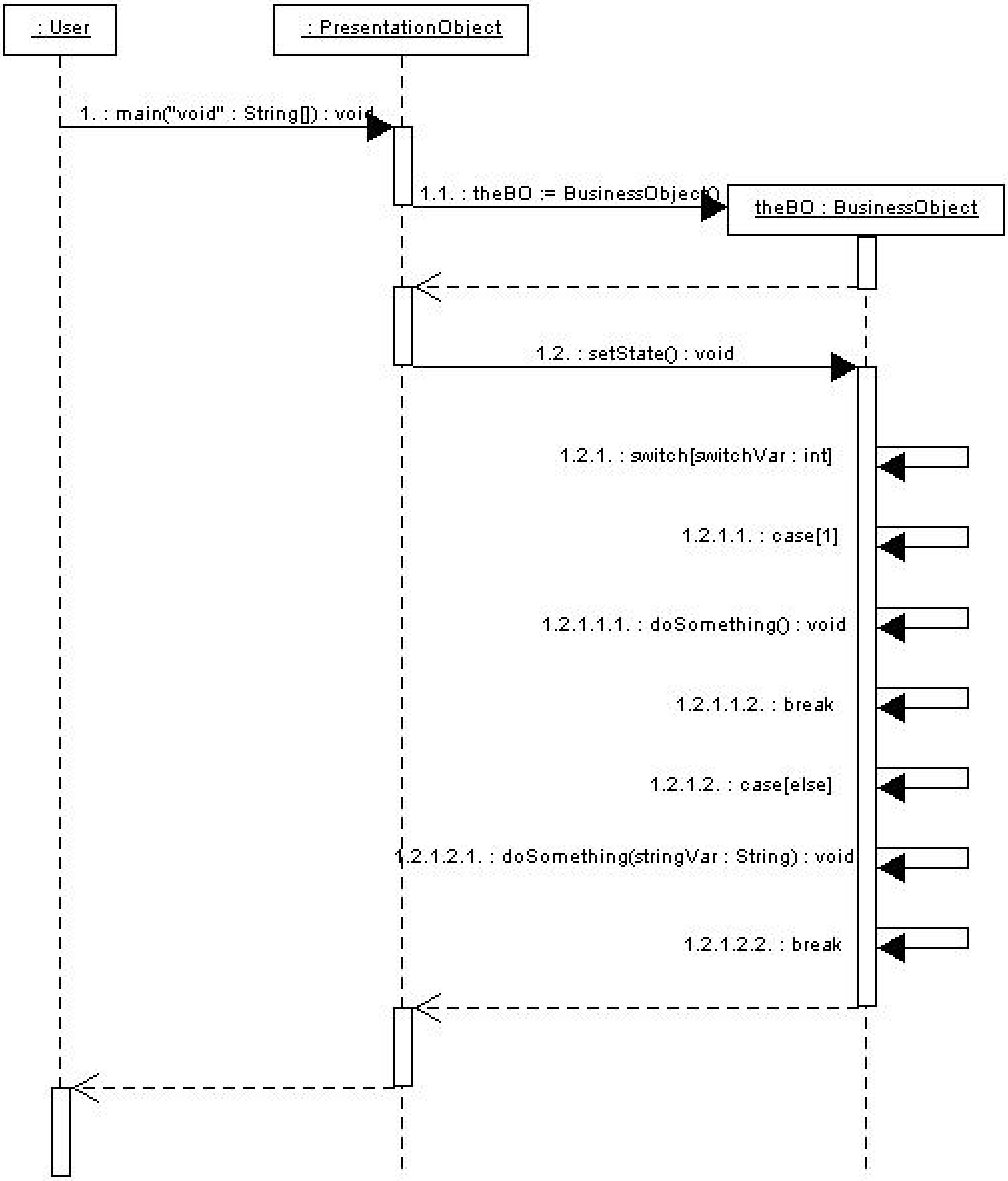

CircleUML Reference:
Switch Statement
1.1.1. : switch[switchVar : int]
1.1.1.1. : case[1]
1.1.1.1.1. : doSomething() : void
1.1.1.1.2. : break
1.1.1.2. : case[else]
1.1.1.2.1. : doSomething(stringVar : String) : void
1.1.1.2.2. : break
1. : startHere() : void
1.1. : setState() : void
thePO :
PresentationObject
theBO :
BusinessObject
: User

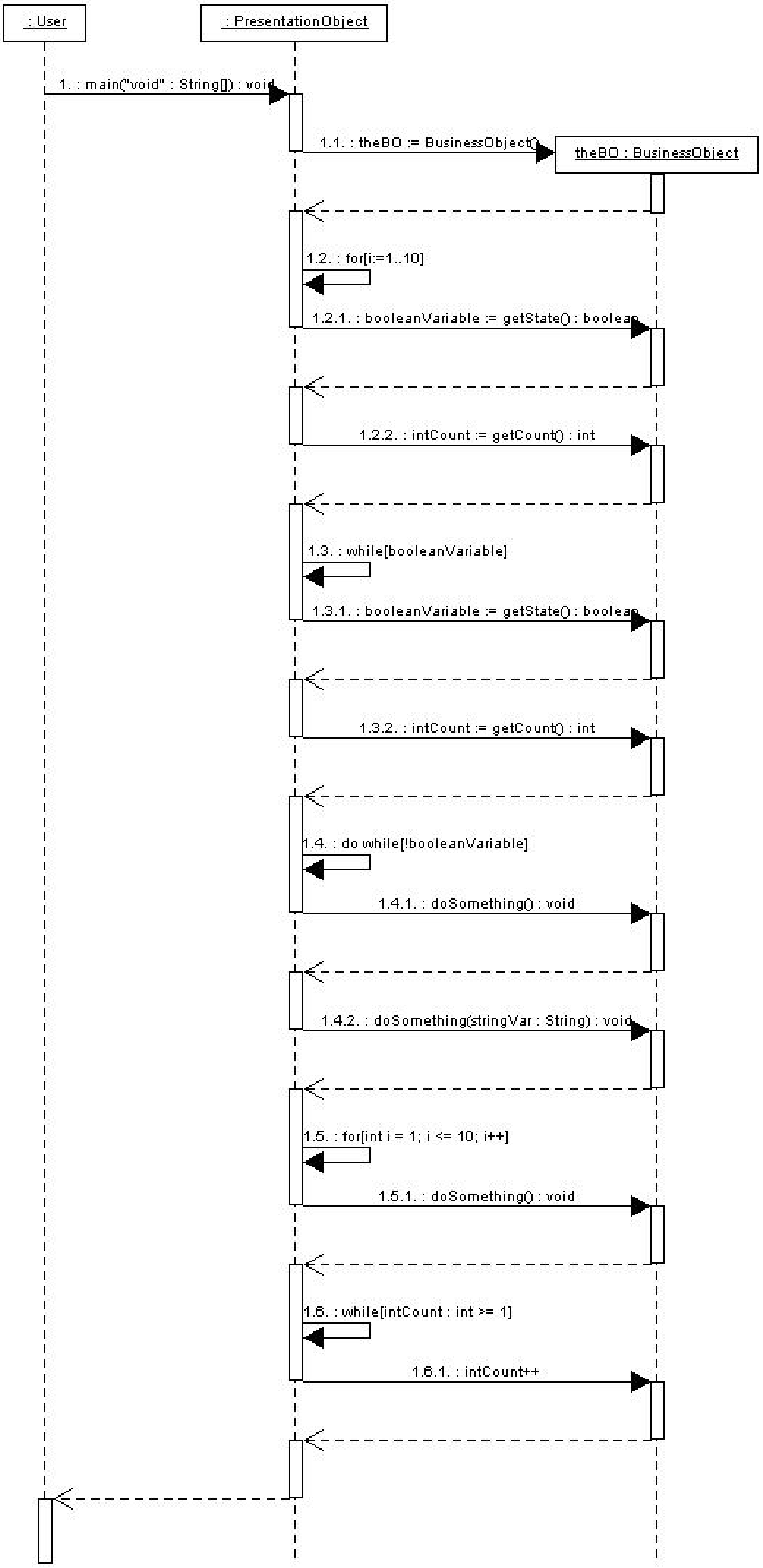

CircleUML Reference:
Try Catch Finally
Statement
1.1. : try
1.2. : catch[Exception e]
1.3. : finally
1.1.1. : openFile(filename : String) : void
1.2.1. : System.out.println(e)
1.3.1. : closeFile() : void
1. : startHere() : void
thePO :
PresentationObject
theBO :
BusinessObject
: User

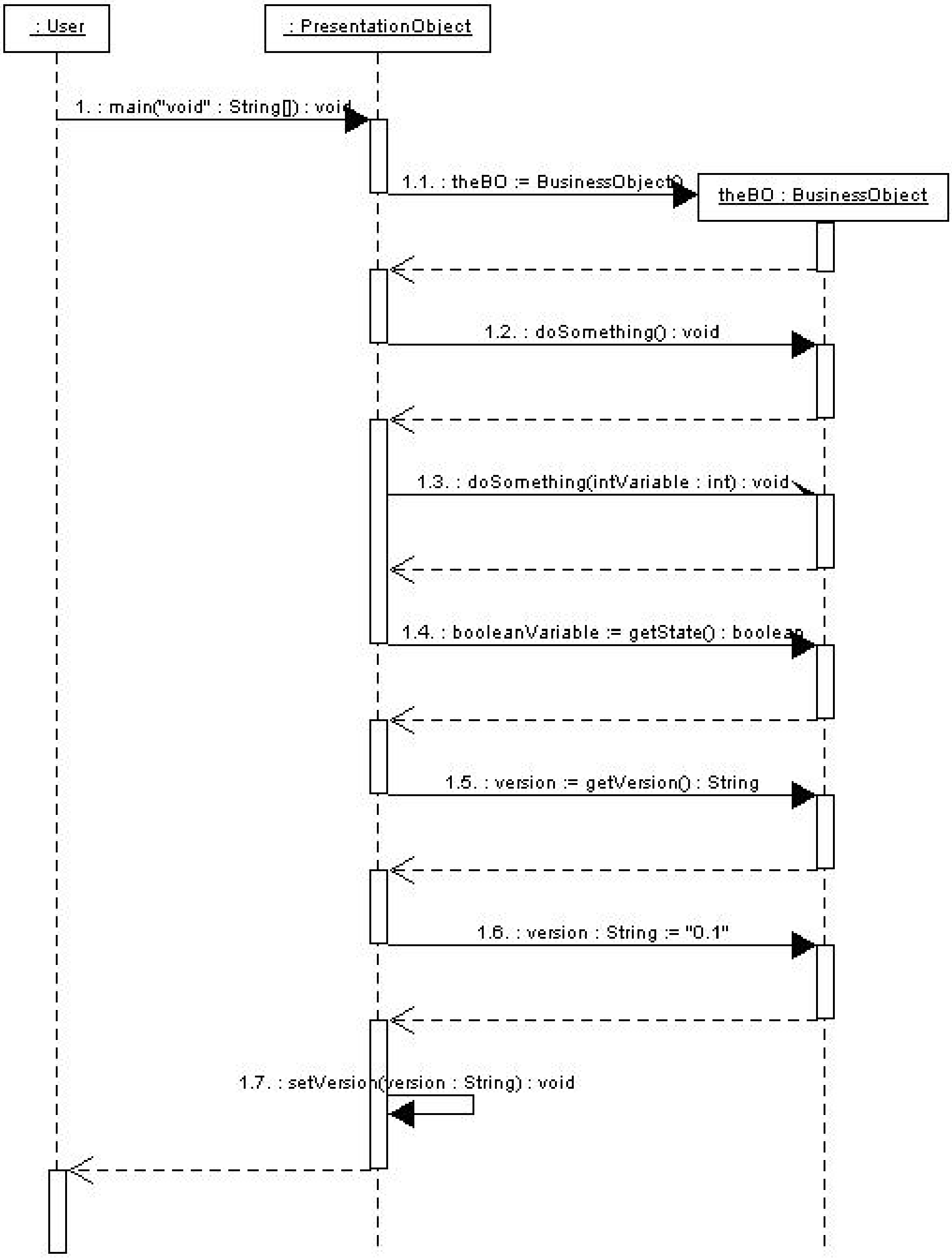
Appendix G
CircleUML reference expressed in Poseidon for UML sequence diagrams

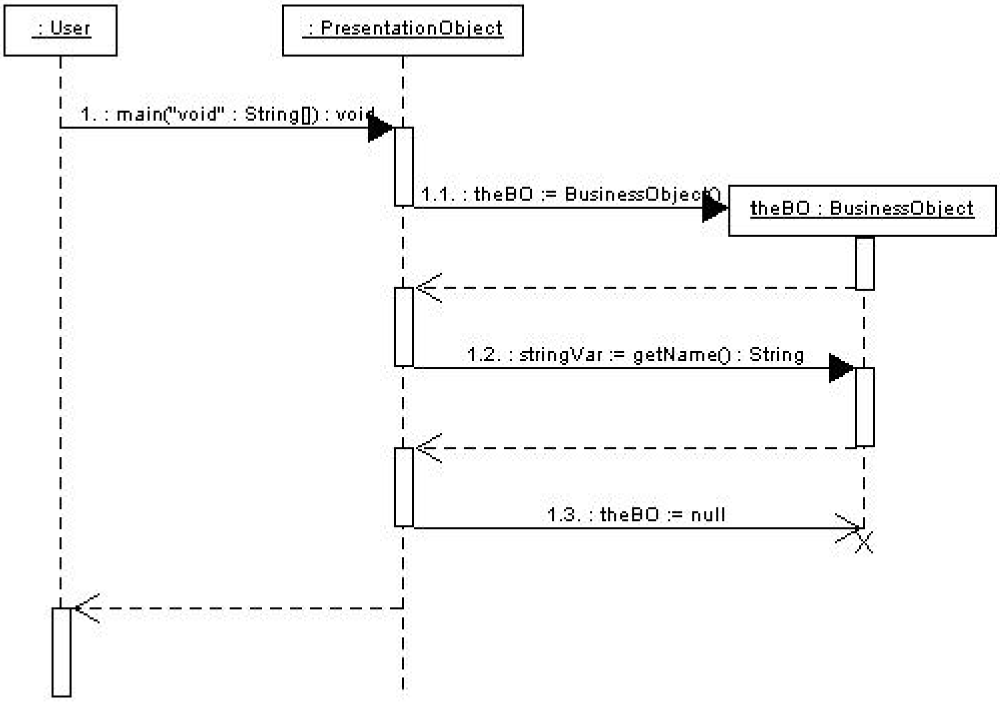

: User
: PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : if[booleanVariable]
1.2.1. : intCount := getCount() : int
1.3. : else if[!booleanVariable]
1.3.1. : average := getAverage(intCount : int) : double
1.4. : if[booleanVariable == true]
1.4.1. : name := getName() : String
1.4.2. : name : String := "CircleUML"
1.4.3. : setName(name : String) : void
1.5. : else if[booleanVariable == false]
1.5.1. : version := getVersion() : String
1.6. : else
1.6.1. : doSomething() : void
1.7. : if[intVariable : int > 100]
1.7.1. : doSomething(intVariable : int) : void
1.8. : else if[intVariable : int <= 100]
1.8.1. : doSomething() : void

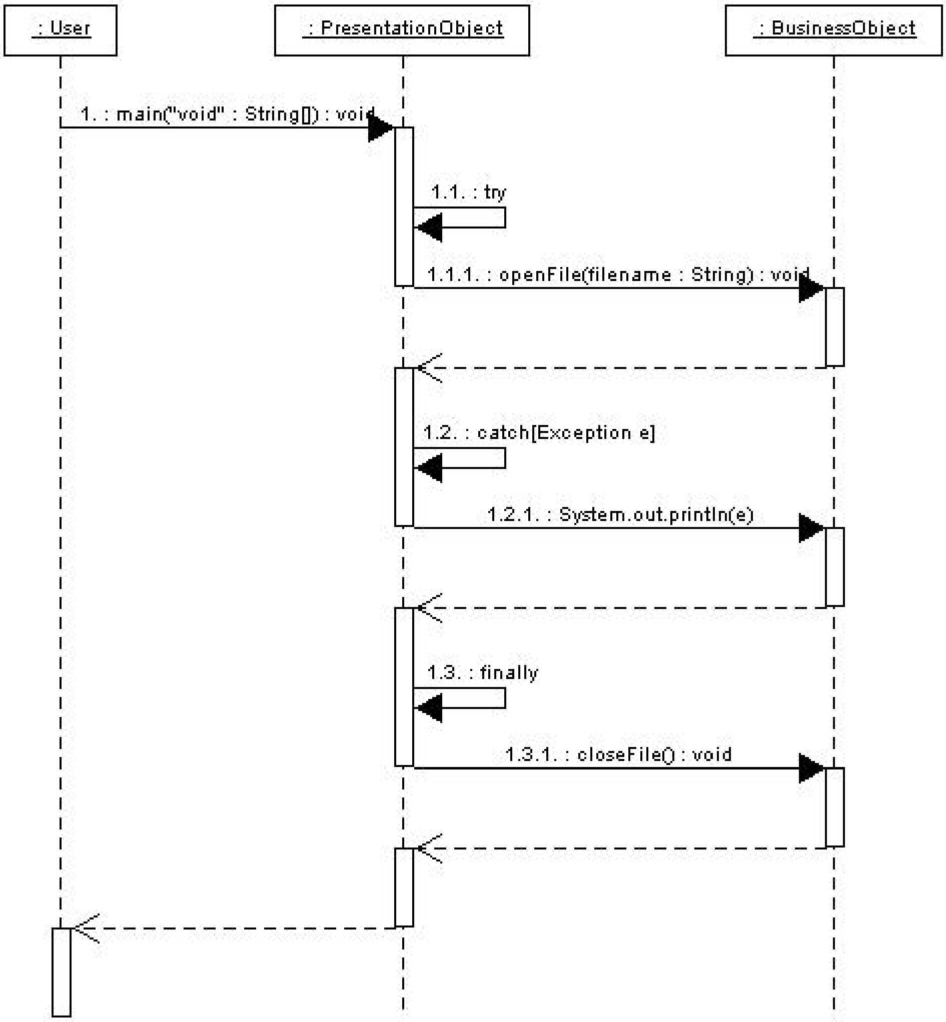

: User
: PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : anotherBO := BusinessObject(theBO : BusinessObject)
anotherBO : BusinessObject


: User
: PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : setState() : void
1.2.1. : switch[switchVar : int]
1.2.1.1. : case[1]
1.2.1.1.1. : doSomething() : void
1.2.1.1.2. : break
1.2.1.2. : case[else]
1.2.1.2.1. : doSomething(stringVar : String) : void
1.2.1.2.2. : break

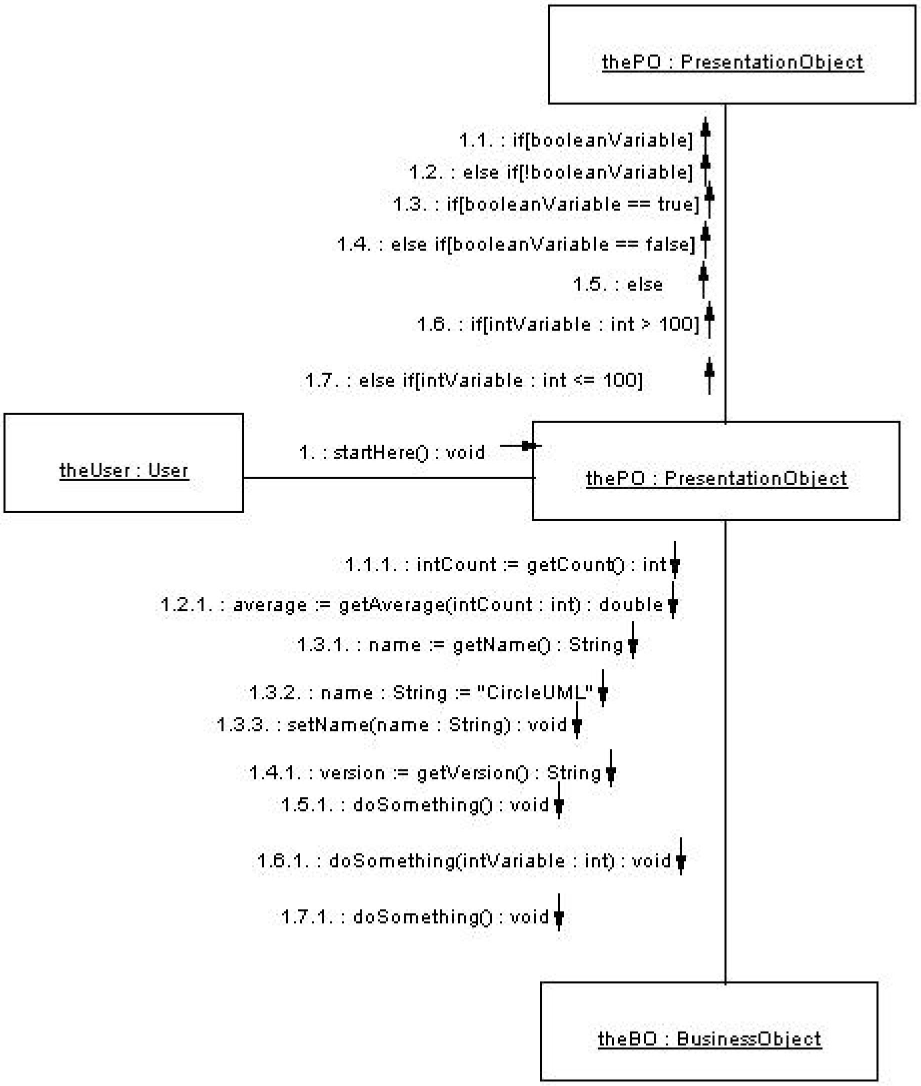

: User
: PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : for[i:=1..10]
1.2.1. : booleanVariable := getState() : boolean
1.2.2. : intCount := getCount() : int
1.3. : while[booleanVariable]
1.3.1. : booleanVariable := getState() : boolean
1.3.2. : intCount := getCount() : int
1.4. : do while[!booleanVariable]
1.4.1. : doSomething() : void
1.4.2. : doSomething(stringVar : String) : void
1.5. : for[int i = 1; i <= 10; i++]
1.5.1. : doSomething() : void
1.6. : while[intCount : int >= 1]
1.6.1. : intCount++

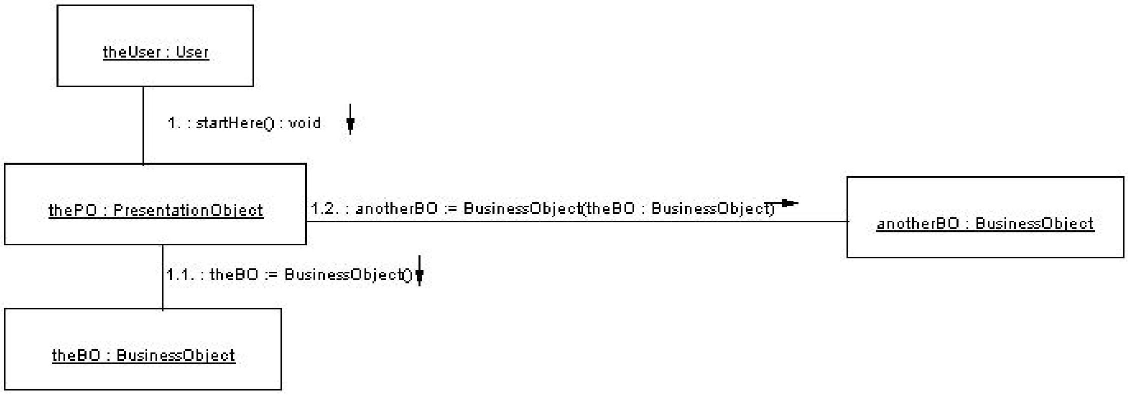

: User
: PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : doSomething() : void
1.3. : doSomething(intVariable : int) : void
1.4. : booleanVariable := getState() : boolean
1.5. : version := getVersion() : String
1.6. : version : String := "0.1"
1.7. : setVersion(version : String) : void

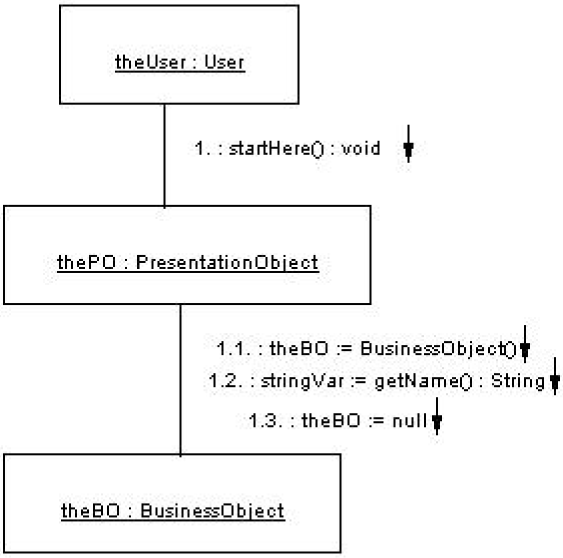

: User
: PresentationObject
1. : main("void" : String[]) : void
1.1. : theBO := BusinessObject()
theBO : BusinessObject
1.2. : stringVar := getName() : String
1.3. : theBO := null

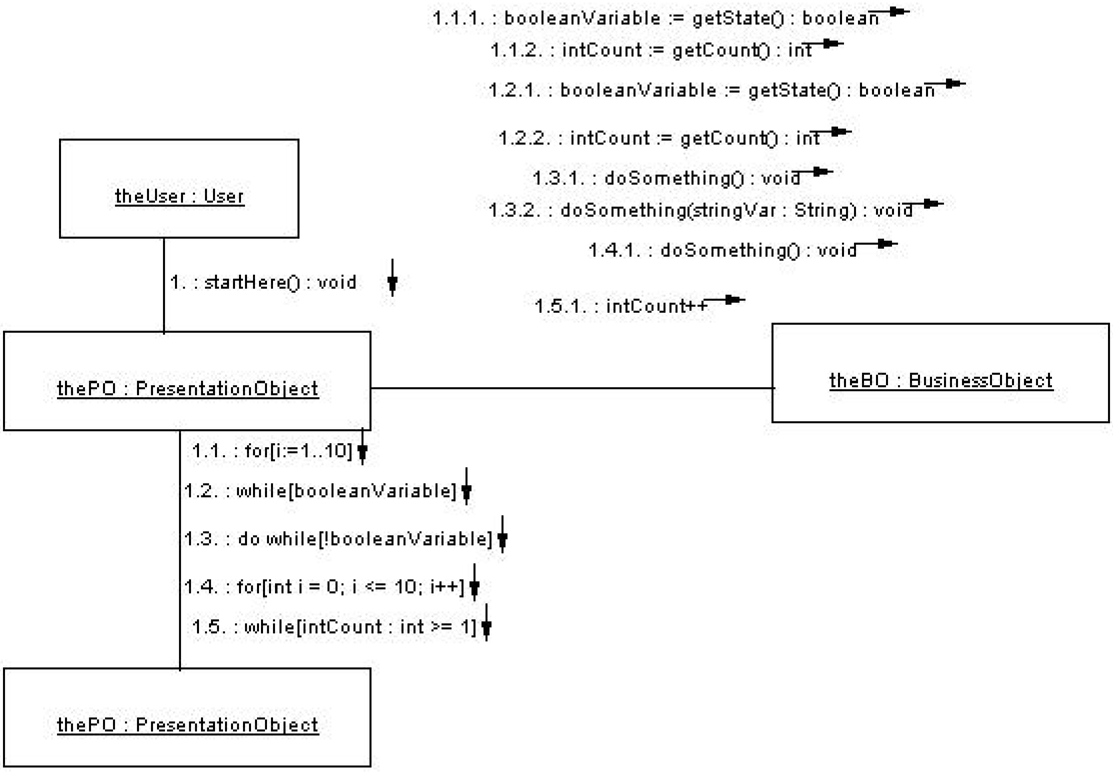

: User
: PresentationObject
: BusinessObject
1. : main("void" : String[]) : void
1.1. : try
1.1.1. : openFile(filename : String) : void
1.2. : catch[Exception e]
1.2.1. : System.out.println(e)
1.3. : finally
1.3.1. : closeFile() : void

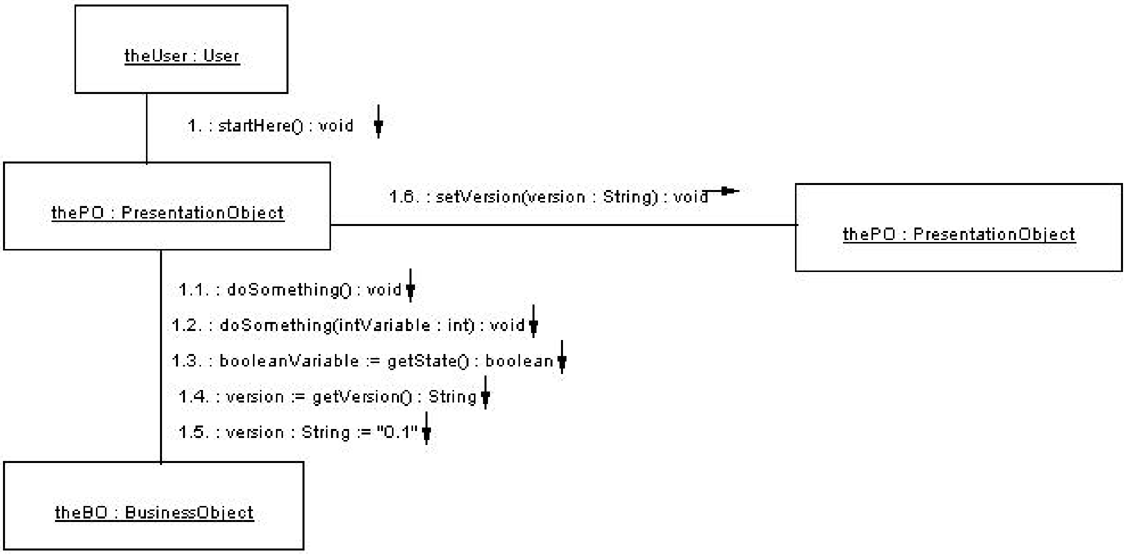
Appendix H
CircleUML reference expressed in Poseidon for UML collaboration diagrams

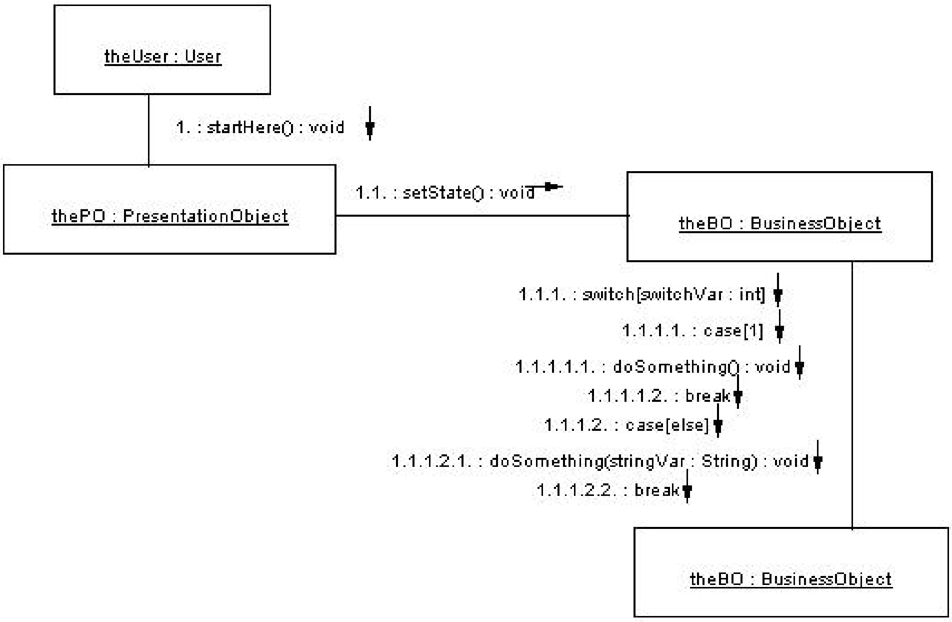

thePO : PresentationObject
1.1. : if[booleanVariable]
1.2. : else if[!booleanVariable]
1.3. : if[booleanVariable == true]
1.4. : else if[booleanVariable == false]
1.5. : else
1.6. : if[intVariable : int > 100]
1.7. : else if[intVariable : int <= 100]
theUser : User
1. : startHere() : void
thePO : PresentationObject
1.1.1. : intCount := getCount() : int
1.2.1. : average := getAverage(intCount : int) : double
1.3.1. : name := getName() : String
1.3.2. : name : String := "CircleUML"
1.3.3. : setName(name : String) : void
1.4.1. : version := getVersion() : String
1.5.1. : doSomething() : void
1.6.1. : doSomething(intVariable : int) : void
1.7.1. : doSomething() : void
theBO : BusinessObject

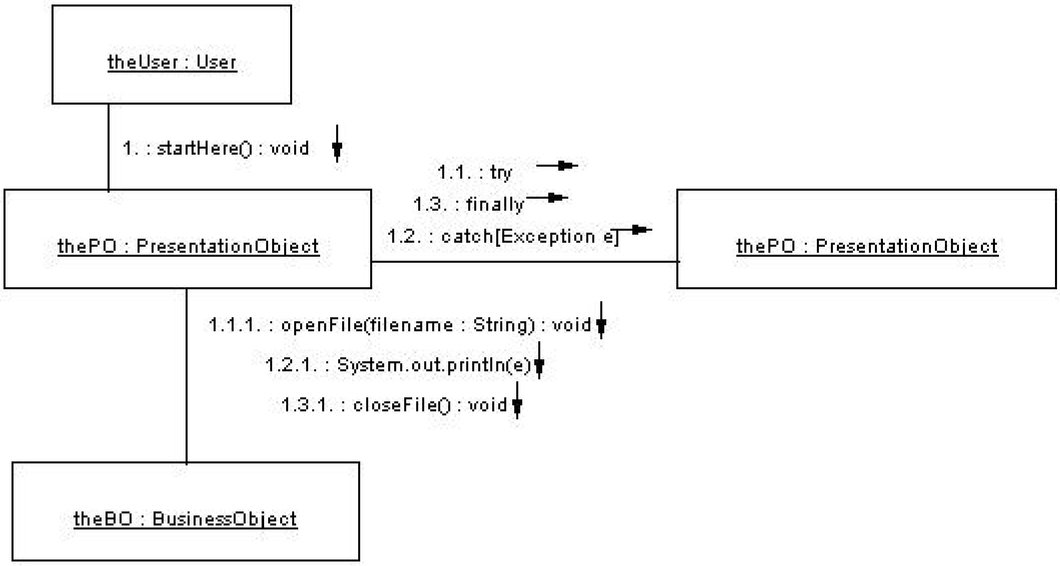

theUser : User
1. : startHere() : void
thePO : PresentationObject
1.2. : anotherBO := BusinessObject(theBO : BusinessObject)
anotherBO : BusinessObject
1.1. : theBO := BusinessObject()
theBO : BusinessObject


theUser : User
1. : startHere() : void
thePO : PresentationObject
1.1. : theBO := BusinessObject()
1.2. : stringVar := getName() : String
1.3. : theBO := null
theBO : BusinessObject

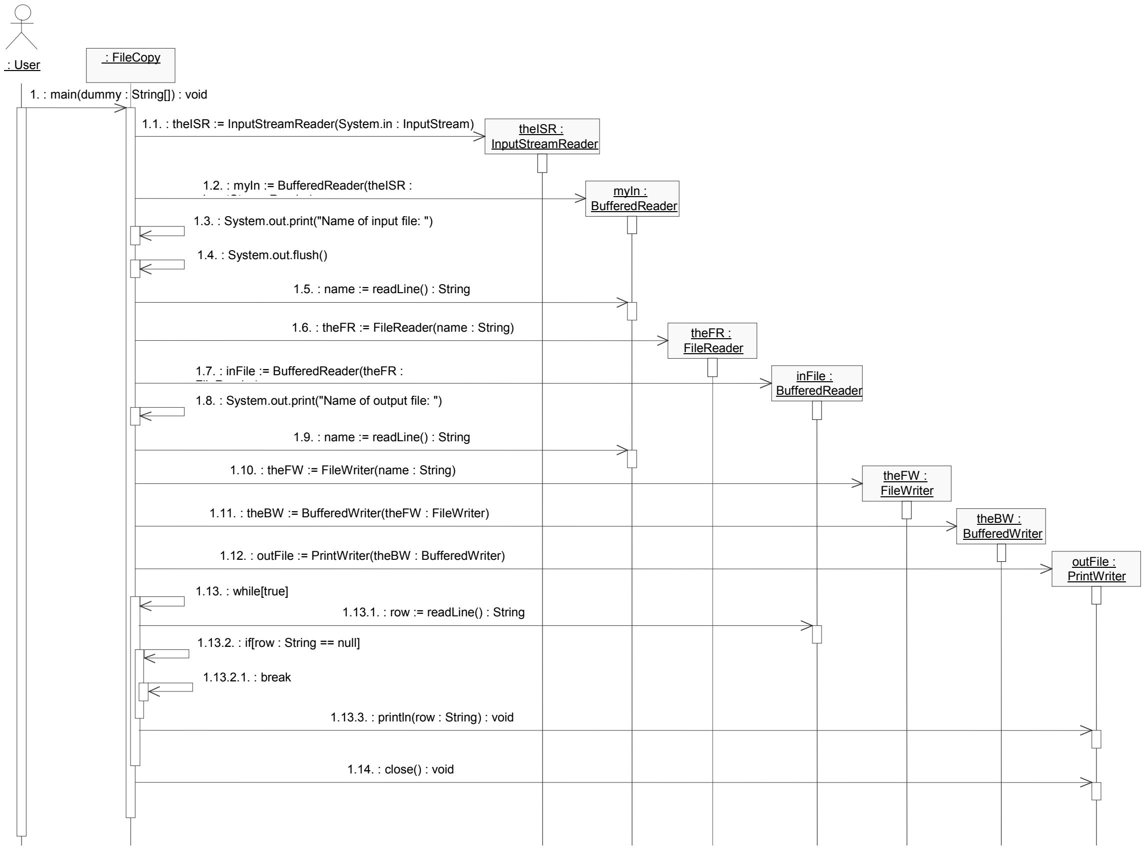

1.1.1. : booleanVariable := getState() : boolean
1.1.2. : intCount := getCount() : int
1.2.1. : booleanVariable := getState() : boolean
1.2.2. : intCount := getCount() : int
1.3.1. : doSomething() : void
1.3.2. : doSomething(stringVar : String) : void
1.4.1. : doSomething() : void
1.5.1. : intCount++
theUser : User
1. : startHere() : void
thePO : PresentationObject
theBO : BusinessObject
1.1. : for[i:=1..10]
1.2. : while[booleanVariable]
1.3. : do while[!booleanVariable]
1.4. : for[int i = 0; i <= 10; i++]
1.5. : while[intCount : int >= 1]
thePO : PresentationObject


theUser : User
1. : startHere() : void
thePO : PresentationObject
1.6. : setVersion(version : String) : void
thePO : PresentationObject
1.1. : doSomething() : void
1.2. : doSomething(intVariable : int) : void
1.3. : booleanVariable := getState() : boolean
1.4. : version := getVersion() : String
1.5. : version : String := "0.1"
theBO : BusinessObject

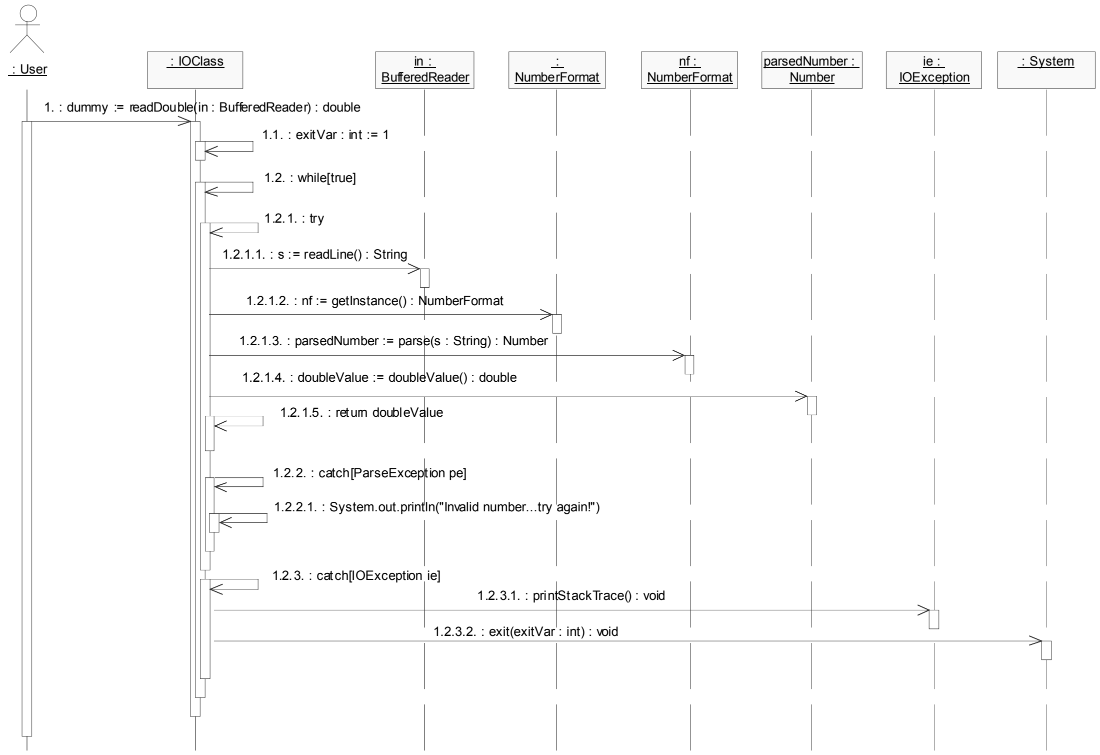

theUser : User
1. : startHere() : void
thePO : PresentationObject
1.1. : setState() : void
theBO : BusinessObject
1.1.1. : switch[switchVar : int]
1.1.1.1. : case[1]
1.1.1.1.1. : doSomething() : void
1.1.1.1.2. : break
1.1.1.2. : case[else]
1.1.1.2.1. : doSomething(stringVar : String) : void
1.1.1.2.2. : break
theBO : BusinessObject


theUser : User
1. : startHere() : void
1.1. : try
1.3. : finally
1.2. : catch[Exception e]
thePO : PresentationObject
thePO : PresentationObject
1.1.1. : openFile(filename : String) : void
1.2.1. : System.out.println(e)
1.3.1. : closeFile() : void
theBO : BusinessObject

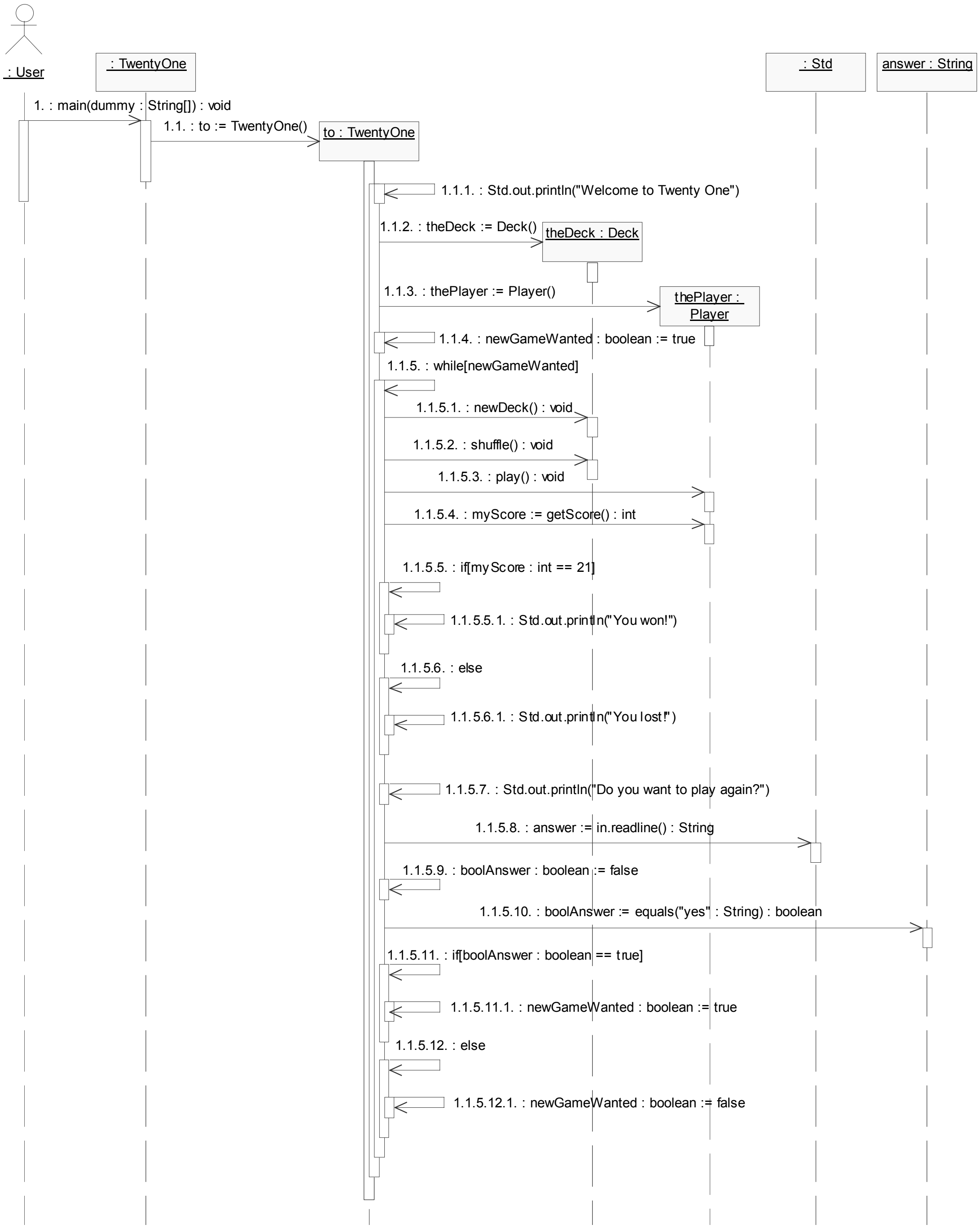
Appendix I
Figure 21 presented as a whole


: User
: FileCopy
1. : main(dummy : String[]) : void
1.1. : theISR := InputStreamReader(System.in : InputStream)
theISR : InputStreamReader
1.2. : myIn := BufferedReader(theISR :
myIn : BufferedReader
1.3. : System.out.print("Name of input file: ")
1.4. : System.out.flush()
1.5. : name := readLine() : String
1.6. : theFR := FileReader(name : String)
theFR : FileReader
1.7. : inFile := BufferedReader(theFR :
inFile : BufferedReader
1.8. : System.out.print("Name of output file: ")
1.9. : name := readLine() : String
1.10. : theFW := FileWriter(name : String)
theFW : FileWriter
1.11. : theBW := BufferedWriter(theFW : FileWriter)
theBW : BufferedWriter
1.12. : outFile := PrintWriter(theBW : BufferedWriter)
outFile : PrintWriter
1.13. : while[true]
1.13.1. : row := readLine() : String
1.13.2. : if[row : String == null]
1.13.2.1. : break
1.13.3. : println(row : String) : void
1.14. : close() : void

Appendix J
Figure 23 presented as a whole


: User
: IOClass
in : BufferedReader
: NumberFormat
nf : NumberFormat
parsedNumber : Number
ie : IOException
: System
1. : dummy := readDouble(in : BufferedReader) : double
1.1. : exitVar : int := 1
1.2. : while[true]
1.2.1. : try
1.2.1.1. : s := readLine() : String
1.2.1.2. : nf := getInstance() : NumberFormat
1.2.1.3. : parsedNumber := parse(s : String) : Number
1.2.1.4. : doubleValue := doubleValue() : double
1.2.1.5. : return doubleValue
1.2.2. : catch[ParseException pe]
1.2.2.1. : System.out.println("Invalid number...try again!")
1.2.3. : catch[IOException ie]
1.2.3.1. : printStackTrace() : void
1.2.3.2. : exit(exitVar : int) : void

Appendix K
Figure 24 presented as a whole


: User
: TwentyOne
: Std
answer : String
1. : main(dummy : String[]) : void
1.1. : to := TwentyOne()
to : TwentyOne
1.1.1. : Std.out.println("Welcome to Twenty One")
1.1.2. : theDeck := Deck()
theDeck : Deck
1.1.3. : thePlayer := Player()
thePlayer : Player
1.1.4. : newGameWanted : boolean := true
1.1.5. : while[newGameWanted]
1.1.5.1. : newDeck() : void
1.1.5.2. : shuffle() : void
1.1.5.3. : play() : void
1.1.5.4. : myScore := getScore() : int
1.1.5.5. : if[myScore : int == 21]
1.1.5.5.1. : Std.out.println("You won!")
1.1.5.6. : else
1.1.5.6.1. : Std.out.println("You lost!")
1.1.5.7. : Std.out.println("Do you want to play again?")
1.1.5.8. : answer := in.readline() : String
1.1.5.9. : boolAnswer : boolean := false
1.1.5.10. : boolAnswer := equals("yes" : String) : boolean
1.1.5.11. : if[boolAnswer : boolean == true]
1.1.5.11.1. : newGameWanted : boolean := true
1.1.5.12. : else
1.1.5.12.1. : newGameWanted : boolean := false

Appendix L
Description of appendix material contained on CD-ROM

- Root of the CD-ROM
    - CIRCLEUML_LICENSE file
        - GNU General Public License information for CircleUML.
    - CircleUML_v0.1.jar file
        - CircleUML executable file.
    - JAVA_GRAPHICS_LICENSE file
        - License that must be distributed with applications that uses icons or graphics from the Java look and feel graphics repository.
    - README file
        - Usage instructions for CircleUML.
    - xalan.jar file
        - Jar file including XPath implementation needed by CircleUML
    - XALAN_LICENSE
        - License that must be distributed with the xalan.jar file.
- CircleUML directory
    - SourceCode directory
        - Contains the complete Java source code for CircleUML
    - UML directory
        - Contains Rational Rose file with UML diagrams for CircleUML
- InteractionDiagramExamples directory
    - Java-files directory
        - Contains Java source code files that are needed when generating source code from the interaction diagram examples.
    - Poseidon directory
        - Diagrams directory
            - Contains sequence and collaboration diagrams for all interaction diagram examples.
        - XMI-files directory
            - Contains XMI files generated from all interaction diagram examples, both from sequence and collaboration diagrams.
    - RationalRose directory
        - Diagrams directory
            - Contains sequence and collaboration diagrams for all interaction diagram examples.
        - XMI-files directory
            - Contains XMI files for version 1.0 and 1.1 generated from all interaction diagram examples, both from sequence and collaboration diagrams.
- ResultDiagrams directory
    - Diagrams directory
        - Contains complete Rational Rose diagrams for all interaction diagrams presented in the Results chapter.
    - Java-files directory
        - Contains Java source code files that are needed when generating source code from the interaction diagrams in the results chapter.
    - XMI-files directory
        - Contains XMI files for version 1.0 and 1.1 generated from the interaction diagrams in the results chapter.

Appendix M
CircleUML requirements

The functional requirements of CircleUML are concerned with what the user can do with CircleUML. The main functional requirements are expressed from a user perspective. These main functional requirements are then expressed in detail from a system perspective.

Functional requirements from a user perspective:

1. The user shall be able to change the language of menus and log messages in the GUI.
2. The user shall be able to open XMI files and have CircleUML parse the content.
3. The user shall be able to have CircleUML generate source code from the opened XMI file.

Functional requirements from a system perspective:

1.1. CircleUML shall provide a choice of supported languages in a menu in the GUI.
1.2. CircleUML shall provide functionality for changing the text for menus and captions of graphical objects into the chosen language.
1.3. CircleUML shall provide a log message area in the GUI and provide functionality for writing necessary information to that log message area.

2.1. CircleUML shall provide an open file dialog.
2.2. CircleUML shall provide an open file filter for XML files.
2.3. CircleUML shall provide functionality for locating needed source code files recursively.
2.4. CircleUML shall provide functionality for determining UML and XMI versions of XMI files.
2.5. CircleUML shall provide functionality for parsing interaction diagram messages expressed in an XMI file.

3.1. CircleUML shall provide functionality for determining the beginning and end of a method body in a Java source code file.
3.2. CircleUML shall provide functionality for translating parsed interaction messages into Java source code.
3.3. CircleUML shall provide functionality for inserting generated code into method bodies in Java source code files.
3.4. CircleUML shall provide functionality for presenting the files with inserted code.

Non-functional requirements

The non-functional requirements of CircleUML are divided into three parts. Where the first part is focused on execution issues, the second part is focused on the targeting platforms for CircleUML and the third part is focused on deployment issues.

Overview of non-functional requirements:

1. CircleUML shall execute demanding tasks on separate threads leaving the main thread free of such tasks.
2. CircleUML shall be able to be run on different processor architectures and operating systems.
3. CircleUML shall be packaged for easy distribution and simple execution of the application.

Detailed non-functional requirements:

1.1. CircleUML shall execute the task of parsing the interaction messages on a separate thread.
1.2. CircleUML shall execute the task of recursively locating needed files on a separate thread.

2.1. CircleUML shall be developed for a platform that targets multiple operating systems.
2.2. The platform shall preferably include some kind of virtual machine or virtual execution system.

3.1. CircleUML shall be packaged in a way that makes distribution simple.
3.2. CircleUML shall be packaged in such a way that execution of the application becomes as simple as running a single command.