An explicit model for ADvance

Student Project

Author

Frank Buchli

December 2002

Supervised by:

Prof. Dr. Stéphane Ducasse
Dr. Roel Wuyts

Institut für Informatik und angewandte Mathematik

The address of the author:

Frank Buchli
Benedikt Hugistrasse 13
CH-4500 Solothurn

or

Software Composition Group
University of Bern
Institute of Computer Science and Applied Mathematics
Neubrückstrasse 10
CH-3012 Bern
buchli@iam.unibe.ch
http://www.iam.unibe.ch/~buchli/

Abstract

ADvance is a powerful round-trip UML viewer integrated with the VisualWorks Smalltalk environment. Because ADvance implicitly uses Smalltalk as its model, it cannot be used for anything but Smalltalk code that is loaded in the image. This project removes this restriction by transparently introducing an explicit model for ADvance using Smalltalk's meta-programming facilities. This solution is validated by showing how it can be used to display ADvance diagrams on *Moose* models.

Contents

1 Introduction **4**

2 ADvance **6**
2.1 What is ADvance ? 6
2.1.1 Terminology 7
2.2 Technical view 7
2.2.1 System Extensions and Modification 7
2.2.2 Singleton Pattern 7
2.2.3 Integration with Smalltalk 7
2.2.4 Linking the classes with ADvance 8
2.2.5 Type Inference 8

3 Integrating an explicit model in ADvance **10**
3.1 The problem 10
3.2 Concept 10
3.3 The Class Simulator 11
3.4 Integrate the new model 12
3.5 Validation 14
3.5.1 Moose 14
3.5.2 Visualizing *Moose* models using ADvance 15
3.5.3 The LAN example 16

4 Conclusion **17**
4.1 Summary 17
4.2 Lessons Learned 17
4.3 Future Work 18
4.4 Acknowledgement 18

List of Figures

2.1 ADvance takes all the information from the class. 8

2.2 Overview of the Type Inference Engine classes. 9

3.1 Solution takes information from objects. 11

3.2 Simulating classes replacing real classes. 11

3.3 *Moose* Architecture. 14

3.4 The LAN Example. 16

List of Tables

3.1 Protocol for a class 12
3.2 IC&C-Protocol enhancements for a class 13
3.3 Protocol for a Metaclass 13

Chapter 1

Introduction

UML has now become an industrial standard for describing object oriented design. ADvance is a tool which shows source code from a VisualWorks [1] image. The problem is that it only *shows information from source in the image.*

The Unified Modeling Language (**UML**) [OBJE 99] is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for the modeling of businesses and other non-software systems. Therefore it proposes a modeling language (a notation) as well as a set of model types (information content) to describe different aspects of a software system. The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.

ADvance is a system round-trip engineering tool from IC&C [2]. It is a multidimensional OOAD-tool for supporting object-oriented analysis and design, reverse engineering and documentation. ADvance provides an always up-to-date graphical view on models. It is fully integrated into VisualWorks. Modifying a diagram means changing your *source code*; changing the source code has a direct impact on your diagram. Its visual interface helps you to quickly develop your object model and allows one to understand and/or modify the object design inherit in your code.

ADvance uses the underlying Smalltalk meta model: To store the needed information, it uses the source code from the Visual Works image itself. The information about a class is stored in this class itself. For finding out information for the diagram, ADvance directly talks to the Smalltalk meta model, for example: All the attributes of a certain class. It analysis the comments as well, for example to find out if the method is abstract or not.

The full integration into VisualWorks is besides all its advantages also a **problem**: The round-trip engineering facilities from ADvance cannot be used on Smalltalk code other than VisualWorks code from the current image, or even complete foreign code like Java.

To **solve** this problem, a new real model for ADvance has to be created. This model should have all the information ADvance needs. The new objects of this model have to be wrapped into Smalltalk meta model compliant objects.

[1] http://www.cincom.com/smalltalk
[2] http://www.io.com/∼icc/

In the next chapter ADvance will be explained in detail. Chapter 3 presents the new explicit model and its integration.

Chapter 2

ADvance

This chapter takes a closer look at ADvance, giving an overview of the features, explaining the terminology and presents the technical important aspects. This information is useful to understand the chosen solution, presented in the next chapter.

2.1 What is ADvance ?

With ADvance[IC 01] the structural, behavioral and architectural information contained in VisualWorks applications can be extracted, visualized and analyzed. Furthermore, it provides a powerful mechanism for supporting concurrent design documentation and construction of classes and relationships.

ADvance consists mainly of two components: a **forward engineering** part and a **reverse engineering** part.

The purpose of the forward engineering part is to support the engineer in the design and implementation work. This work can not be isolated, it is an iterative process. ADvance supports this iterative process: While you *draw* your design (class diagrams), the corresponding code is automatically generated. Changing the generated code directly manipulates your diagrams. The integrity of the design and the implementation is maintained from the design through the implementation.

The reverse engineering part helps you to understand code. There is no need that your code was developed with ADvance in order to visualize it, so it is possible to analyze foreign code to get a good overview. Changing the diagram means that you directly make the changes *into* the source code.

Additionally to this two parts, there is a documentation tool. The documentation facilities help you creating textual and graphical documentation – for methods, classes, interactions in use cases, subsystems, or complete applications. The tool also allows the user to view existing code as object design models. In this manner the model can be used to review existing OO constructs or as a design consistency check for the current iteration.

2.1.1 Terminology

A ***subject*** is a set of classes that you want to view with ADvance. The subject is also a collection of different views which represents a certain aspect of a (sub-)system. A subject typically contains 5-30 collaborating classes and often equates closely to a subsystem, business model structure or use case of your application.

A subject has ***diagrams*** which represent different graphical views on the subject. Diagrams let you choose the level of detail you want to use to look at; you can create overview diagrams with the class hierarchy only or more detailed diagrams showing attributes, services and/or relations. Furthermore, you can use diagrams to look at different aspects of a subject, *e.g.*, you can create structural diagrams with inheritance and relations or behavioral diagrams, which show message paths.

2.2 Technical view

The focus of this technical view is to give the reader as much information that he will understand the solution presented in the next chapter better.

2.2.1 System Extensions and Modification

Installation of the ADvance parcels will result in the extension of some existing system classes. All system extensions are installed under protocol names that reflect the owning module and the fact that the method is an extension. For example, the method `ApplicationModel>>iccEnable:group:` is installed under the protocol ICC-Common Classes. As shown in this example IC&C has chosen method name prefixes to avoid naming conflicts.

2.2.2 Singleton Pattern

It is important to know how an application has access to its subcomponents. IC&C uses almost all the time for their important classes the "Singleton Design Pattern" [GAMM 95]: Each class has a shared variable `Default` and a method `initialize`:

```
initialize
    Default := self new
```

At any time, you can get with the `default` accessor a reference to the object. This mechanism allows an easy replacement of certain subcomponents.

2.2.3 Integration with Smalltalk

As already mentioned ADvance stores all the necessary information in the source code of the class. Also type information (see below 2.2.5) is stored in the class. They use the method `ad2ClassInfo` on the class side. This method returns a string:

```
aClass>>ad2ClassInfo
```

```
^'
  Instance Variables:
    aName <String>
    aNumber <Integer>'
```

ADvance asks the classes with the Smalltalk meta-programming facilities about their attributes, methods, etc, see Figure 2.1.

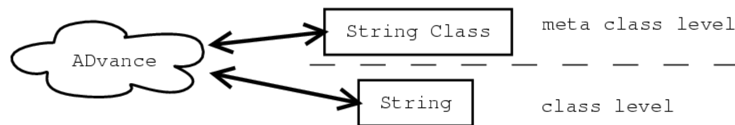

Figure 2.1: ADvance takes all the information from the class.

All the information is taken from the source code and is stored in the source code. This is the implicit model of ADvance.

2.2.4 Linking the classes with ADvance

Interesting for this project is how does ADvance access to the classes? Fortunately there is one single hook to do so. It is the class `AD2SystemEnvironment`. In the protocol *enumerating* their is the only hook to the system classes:

```
AD2SystemEnvironment>>allClassesDo: aBlock
    Kernel.SystemUtils allClassesDo: aBlock
```

2.2.5 Type Inference

An UML diagram displays also type information. The problem is that Smalltalk is a typeless language. ADvance cannot ask a class what types their instance variables are. ADvance tries to elaborate the type of an attribute or return value.

IC&C has therefore written a type inference engine (See Figure 2.2 for an overview), which is not explained here any deeper.

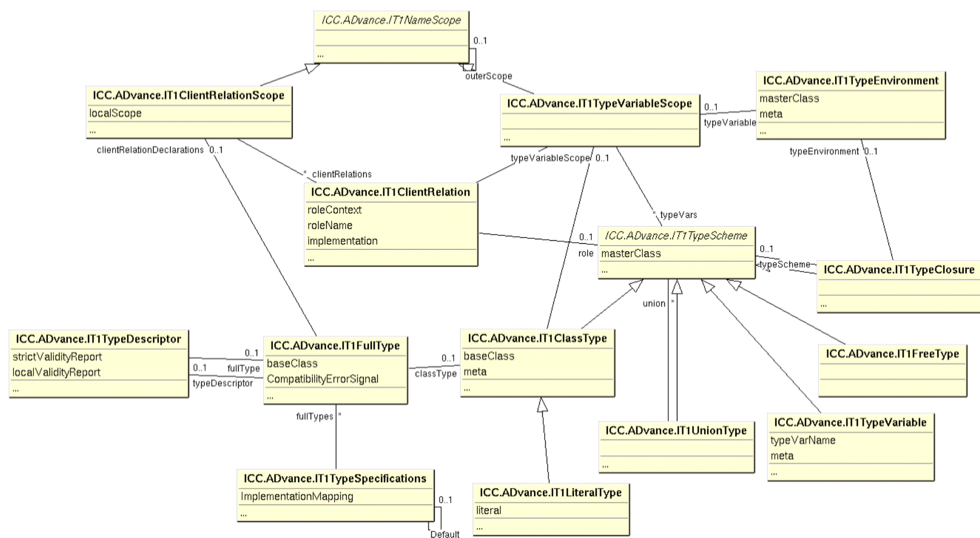

Figure 2.2: Overview of the Type Inference Engine classes.

Chapter 3

Integrating an explicit model in ADvance

The previous chapter showed the power of ADvance and how it is technical realised. The strong integration of ADvance into VisualWorks brings besides all its advantages also some problems, which are discussed in the first section. A solution is shown in the next three sections. As a proof of concept ADvance takes with the presented bridge information from another tool and shows it as an UML diagram. The last sections shows possible future work to accomplish.

3.1 The problem

Because ADvance implicitly uses Smalltalk as its model, it cannot be used for anything but Smalltalk code that is loaded in the image. All the needed information is taken from the classes itself. Additional information is stored in the classes as well, for example in the comment or in special methods. This is the reason for the need of a new explicit model. All the communication with the classes have to been redirected to this new model.

3.2 Concept

The realisation consists of two main steps:

1. Construct an explicit model for ADvance.
2. Let ADvance use this new model.

The new model should be an object which has the same protocol as a class: This new object *pretends to be a class* and are called *Class Simulator*. ADvance does not deal anymore with the class side, now it asks normal objects about attributes, methods, etc, see Figure 3.1 and compare it with Figure 2.1. This solution is transparent: ADvance is not aware that it talks to objects instead of classes.

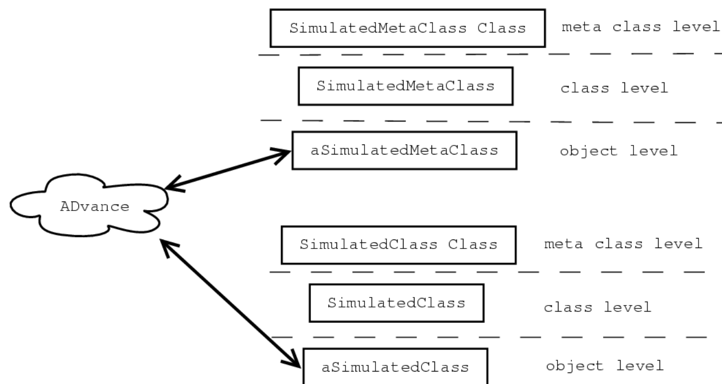

Figure 3.1: Solution takes information from objects.

To ensure that ADvance uses this new objects, each time ADvance has access to a class, we have to give the reference to the new objects.

3.3 The Class Simulator

As ADvance wants to have Smalltalk classes, we have to make an object which looks *like* a class. This new object is *simulating a class*, see Figure 3.2.

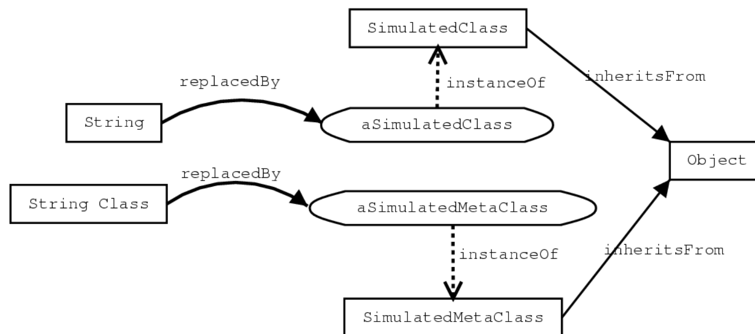

Figure 3.2: Simulating classes replacing real classes.

Also the class side has to be simulated. We use therefor a `MetaClassSimulator`. Asking an object from `ClassSimulator` about its class, it returns either a `MetaClassSimulator` or a meta class from Smalltalk. The `MetaClassSimulator` is returned when ADvance is calling the method. The reason for this mechanism is to keep the system stable. Without returning Smalltalk Metaclasses you cannot inspect, debug, etc your simulation anymore.

The whole protocol ADvance needs you find in Table 3.1. IC&C made some enhancements to the class `Class`, we also have to simulate them, see Table 3.2. The protocol

allInstVarNames	Returns an Array of the names of the receiver's instance variables. (including superclasses)
class	Returns the Smalltalk Metaclass, unless ADvance is calling it, then it returns an instance of MetaClassSimulator.
comment	Returns the comment of a class.
compiledMethodAt: selector	Returns the compiled method associated with the message selector in the receiver's method dictionary. If the selector is not in the dictionary, create an error notification.
fullName	Returns the name of the receiver, fully qualified.
includesBehavior: aClass	¡returns whether the argument, aClass, is equal to self or is on the receiver's superclass chain.
includesSelector:aSymbol	Returns whether the message whose selector is the argument is in the method dictionary of the receiver's class.
instVarNames	Returns an Array of the names of the receiver's instance variables.
isMeta	Returns whether the receiver is a meta class.
isOverride	Returns true if the receiver represents an Overridden object.
organization	Returns the instance of ClassOrganizer that represents the organization of the messages of the receiver. (ADvance uses this information to assume certain things like if a an attribute is private.)
prequisitesForLoading	List those classes that must be filed, BOSSed or load into the system before I can be loaded.
subclasses	All Subclasses as a set.
superclass	Superclass or nil.
whichClassIncludesSelector aSymbol	Returns the class on the receiver's superclass chain where the argument, aSymbol (a message selector), will be found.
whichSelectorReferTo: literal	Returns a collection of selectors whose methods access the argument as a literal.

Table 3.1: Protocol for a class

for the Metaclass (Table 3.3 is almost the same, but smaller.)

3.4 Integrate the new model

As wee have seen in Section 2.2.4 we are in a lucky situation: Their is just one hook. Therefore we just have to override the `allClassesDo:` method from `AD2SystemEnvironment`:

```
SystemEnvironmentSimulatingClass>>ClassallClassesDo: aBlock

    "this is for backwards compatibility"
    super allClassesDo: aBlock.
```

clientRelations	Return an empty collection of relations of the classRef class with other classes. To get this information, Advance normally uses the type-checking mechanism to deduce the relations. By returning an empty collection for this class, we effectively shortcut this system. If this were the bridge towards Moose, then we should return a collection of relations as found in Moose in a format that Advance understands. See Section 2.2.5
clientRelationDeclarations	Elaborates the types is only started on demand. See Section 2.2.5
fullType	Returns a type based on a class. See Section 2.2.5
infoString	Return the string ADvance will extract type information from. Depending on the settings use either the comment or the method given by the user in the preferences.
isADvanceSubject	True when the class is used to store a *subject*2.1.1
typeVarDeclarations	Returns `TTypeVariableScope` See section 2.2.5

Table 3.2: IC&C-Protocol enhancements for a class

allInstVarNames	Returns an Array of the names of the receiver's instance variables. (including superclasses)
includesSelector:aSymbol	Answer whether the message whose selector is the argument is in the method dictionary of the receiver's class.
instVarNames	Returns an Array of the names of the receiver's instance variables.
organization	Returns the instance of ClassOrganizer that represents the organization of the messages of the receiver.
whichSelectorReferTo: literal	Returns a collection of selectors whose methods access the argument as a literal.

Table 3.3: Protocol for a Metaclass

```
"new things"
ClassSimulator allInstances copy do:
    [:aSimulatedClass | aBlock value: aSimulatedClass]
```

Calling just once the `initialization` method from the new `SystemEnvironmentSimulatingClass` class (see Section 2.2.2), has the effect that everywhere the simulated classes are used.

3.5 Validation

As a proof of concept, *Moose* is the explicit model which is shown as UML diagram in ADvance. The *LAN example* is used to show how a concrete diagram looks like.

3.5.1 Moose

In the past few years the Software Composition Group (SCG) at the University of Bern has been involved in a number of research projects in the field of software re- and reverse engineering. In the FAMOOS[1] project leading European partners came together to build a number of tool prototypes to support object oriented reengineering.

To avoid equipping the tool prototypes with parsing technology for different programming languages, a common information exchange model with language specific extensions is specified. This model is named FAMIX (**FAM***OOS* **I***nformation* **EX***change Model*).

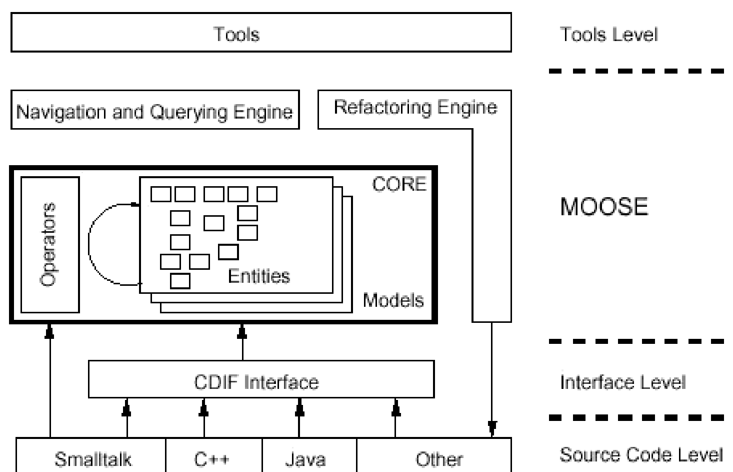

Figure 3.3: *Moose* Architecture.

Moose is the reengineering research platform implemented in VisualWorks Smalltalk [DUCA 00] [DUCA 01] [TICH 01]. It has been developed during the FAMOOS project to reverse engineer and re-engineer object-oriented systems. It consists of a repository

[1] http://www.iam.unibe.ch/~famoos/

to store models of source code. The models are stored based on the entities defined in FAMIX. The software analysis functionality of *Moose* is language independent. The FAMIX models can be loaded from and stored to files. Apart from the repository, there are other features implemented to support reverse engineering activities:

- a parser for Smalltalk code
- an interface to load and store information exchange files
- a software metrics calculation engine
- an interface for additional tools to browse and visualize stored entities

The *Moose* tool is lacking the possibility to generate an overview of the class hierarchy as it would be possible with an UML diagram.

3.5.2 Visualizing *Moose* models using ADvance

The goal is to use ADvance to display diagrams from the current *Moose model*. A model in *Moose* is like a subject (see Section 2.1.1) in ADvance: It consists of meta program elements (classes and methods) and relation between those elements.

A menu item "*UML-Diagram*" calls a little script which takes all the classes from the current *Moose* model into a ADvance subject and shows them.

Here is the pseudo code of this script:

1. Clear all cached information in ADvance.
2. Fetch all classes from the *Moose* model.
3. Fill the gained information into `ClassSimulator` classes.
4. Import this simulated class to a ADvance subject.
5. Open the diagram painter on this subject.

The important steps are the second and third: Here the information from *Moose* goes to the new `ClassSimulator`:

```
MooseToAdvance>>elaborateClasses

   mooseClasses do:
      [:each |
          | tempClass |
          tempClass := ClassSimulator withName: each name.

          each methodsDo:
             [:eachM |
             | methName |
             methName := eachM name asSymbol.
             tempClass addMethod: methName.
             eachM isAbstract
```

```
    ifTrue:
       [(tempClass methodAt: methName) isAbstract: true.]].

 each attributesDo:
    [:eachA | tempClass addInstVar: eachA name].

 each inheritsAsSubclassIsEmpty
    ifFalse:
       [tempClass simulatedSuperclass: (ClassSimulator withName:
          eachinheritsAsSubclass first superclass name)].

 advClasses add: tempClass]
```

3.5.3 The LAN example

The LAN is an often used application for demonstration purposes. It simulates a little computer network. There are general nodes. Subclasses of this nodes are *Fileserver*, *Printer*, etc.

After importing to LAN into a *Moose* model you have a result as shown in Figure 3.4.

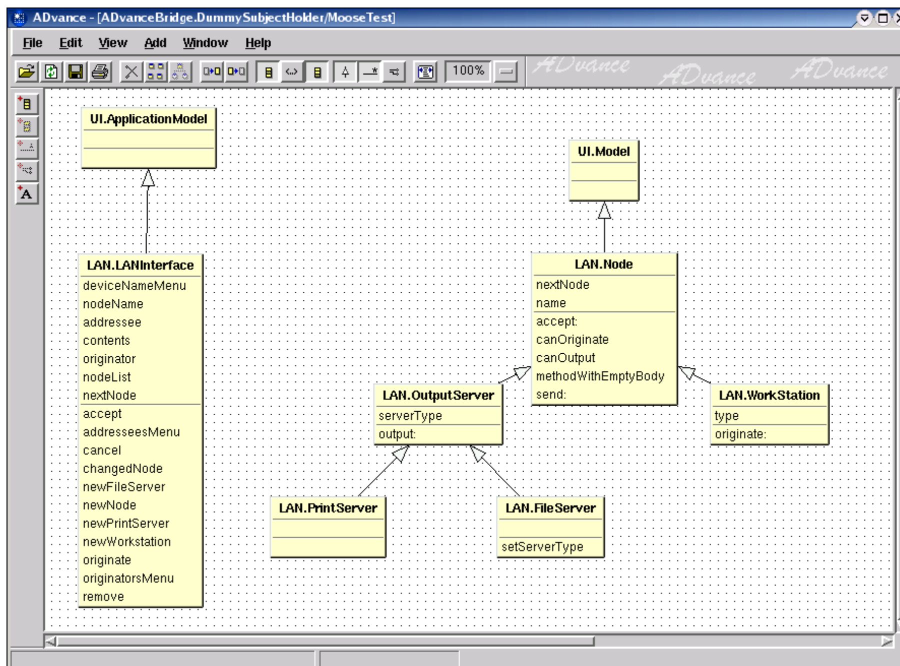

Figure 3.4: The LAN Example.

Chapter 4

Conclusion

This chapter is structured as follows: After a short summary, the three most important things *I* have learned from this project are listed. The next section shows possible future work to accomplish. Last but not least I have to thank two people in the last section.

4.1 Summary

ADvance is enriched by an explicit model. It is possible to take foreign code and let ADvance draw UML class diagrams. Objects from the *Class Simulator* pretend on an object level to be real Smalltalk classes. The bridges ensures that ADvance communicates with this new objects besides the real existing Smalltalk classes.

In the proof of concept *Moose* plays the role of the underlying model. The LAN example is first loaded into the *Moose* meta model. From there the new bridge imports on the fly the information via the *Class Simulator*. Finally ADvance draws the LAN classes.

4.2 Lessons Learned

- The fact that IC&C uses the *Singelton Pattern* (see Section 2.2.2) and the protocol consists of just one single hook, made the realisation much easier than it would have been with with a strongly linked protocol.
- Without the generous *meta-programming facilities of Smalltalk* the solution would have been much harder to realize. It would have been impossible to let the source code of ADvance untouched.
- To find out the protocol of an object (all the messages an object receives), one useful way is the following: Define a new class which inherits from *nil*. Overwrite the `doesNotUnderstand:` method. This new method redirects all the received messages to the observed object and logs their requests. The log contains then all the public methods called during runtime.

4.3 Future Work

The solution with *Moose* is just a prototype. There are still a lot of things to do:

- *Broken GUI*. Because ADvance assumes that their is still real Smalltalk classes behind the diagram, some of the options are broken. Example: If you try to open the class from the context menu, an exception is raised.
- *Smalltalk restrictions*. ADvance was designed to show Smalltalk architecture. Loading with *Moose* a meta model from an other programming language might cause some problems. Two examples:
  1. Multiple inheritance. Smalltalk classes can just inherit from one class. The whole routine where ADvance arranges the classes in a good way would have to be refactored. So far the algorithm just put the parent class in the middle above all the subclasses.
  2. Other concepts like Java Interfaces is not known by ADvance. As Smalltalk does not have the Interface concept from for example Java, ADvance does not know the inheritance flash.

4.4 Acknowlodgement

Last but not least I want to say thank you to my two supervisors: *Prof. Dr. Stéphane Ducasse* and *Dr. Roel Wuyts*. Stéphane for leading my through the project and giving a lot of conceptual inputs and Roel for his pair-programming sessions which enriched my Smalltalk knowledge enormously!

Bibliography

[Duc 99] S. Ducasse and S. Demeyer, editors. The FAMOOS Object-Oriented Reengineering Handbook. University of Bern, October 1999.

[IC 01] IC & C GmBH Software Foundations, Papenhoehe 14, D-25335 Elmshorn/Hamburg, Germany. *ADvance User's Guide*, August 2001. (p 6)

[DUCA 00] S. Ducasse, M. Lanza, and S. Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000. (p 14)

[DUCA 01] S. Ducasse, M. Lanza, and S. Tichelaar. *The Moose Reengineering Environment*. Smalltalk Chronicles, August 2001. (p 14)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison Wesley, Reading, Mass., 1995. (p 7)

[OBJE 99] Object Management Group. *Unified Modeling Language (version 1.3)*. Research report, Object Management Group, June 1999. (p 4)

[TICH 01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001. (p 14)