Enriching Reverse Engineering with Annotations

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Andrea Brühlmann

im April 2008

Leiter der Arbeit

Prof. Dr. Oscar Nierstrasz

Dr. Tudor Gîrba

Institut für Informatik und angewandte Mathematik

Abstract

Successful reverse engineering needs to take into account human knowledge about architecture, about features or even about validation of the results of automatic analyses. This knowledge should be linked to the automatically reverse engineered model and should be taken into account by analyses.

Typically, when we want to reason about data, we first encode an explicit meta-model and then express analyses at that level. However, human knowledge is often implicit and as a consequence it is not possible to describe it comprehensively upfront. In this dissertation we propose a generic approach to iteratively enrich the system model with external knowledge using annotations. Our mechanism allows the reverse engineer to iteratively describe and refine the annotations during the analysis process, instead of requiring the meta-model to be built upfront.

As a validation of the expressiveness of our framework, we show how we use it to support reverse engineering scenarios.

Contents

1 Introduction

The goal of reverse engineering is to understand an existing software system. Through activities like reading the code, talking with the developers or skimming the documentation, we gain knowledge about the system [Demeyer *et al.*, 2002]. Often, the way how we store this knowledge is undefined, because it is unstructured information. So often, a piece of paper or a simple text document is chosen to write down what we learn during reverse engineering.

At the same time, we have analysis tools that work with a model of the source code [Nierstrasz *et al.*, 2005; Marinescu *et al.*, 2005]. They usually have a meta-model which is applicable for many programming languages and can generate metrics, visualizations and different kinds of reports and automatic detections, and they provide the reverse engineer with many ways to navigate and explore a system under analysis.

However, until now there is a missing link. There is no way to link human knowledge to a system model in the reverse engineering tool. Additional information gleaned from discussions with developers and domain experts cannot be directly incorporated in the model and be used when generating reports about the system.

Other researchers have considered the importance of taking external knowledge about a system into consideration during analysis. For example, reflexion models have been proposed for architecture recovery by capturing developer knowledge and then manually mapping architectural entities to the source code [Koschke and Simon, 2003; Murphy *et al.*, 1995]. Another example is provided by Intentional Views which are rules that encode external constraints that are checked against the reality of source code [Mens *et al.*, 2006; Deissenboeck and Ratiu, 2006].

Storing externally gained knowledge about a system directly in the analyzed model would also improve the management of this knowledge, because it can be attached to the matching source components. There could also be possibilities of structuring this information gradually, as soon as some structure emerges. This is hardly possible to do when it is only stored in a text document or even on a sheet of paper.

1.1 Challenges of integrating human knowledge

If the external information should be used for reports, visualizations and other kinds of analyses, it needs to be structured somehow. However, one peculiarity of human knowledge is that often, it cannot be structured upfront or only partly and that the structure changes over time. We might at the beginning have information about some aspect of the system and later on find out that this information could be turned into an attribute that can be attached to certain objects, where for each of these objects we can add a value for it.

This requires that information can be attached to any object, to the whole system itself as well as to all its elements.

Typically, human beings would fill in some values and later find out that the attribute should be changed a little bit to fit better. We know this characteristic from database management and from object oriented programming. An example where such a flexibility is supported for databases is DabbleDB[1]. An analysis tool that wants to integrate human knowledge should take this particularly into account.

Another challenge is that the types of information are unlimited in a human mind. Because of that, not only a given set of data types should be available for the human information, but the reverse engineer should have the possibility to introduce new data types.

Moreover, an important consideration is that people will not be motivated to use an integration tool if it proves to be too complicated and laborious to use. This highlights the importance of providing tool support such as editors to enable them to manipulate units of information and their structure.

With these factors in mind, we argue that an integration of human knowledge into an analysis engine should meet the following demands:

- It should provide flexible (re-)structuring of the information
- It should have a flexible type management
- It should allow for annotating everything
- It should offer tool support such as GUI editing

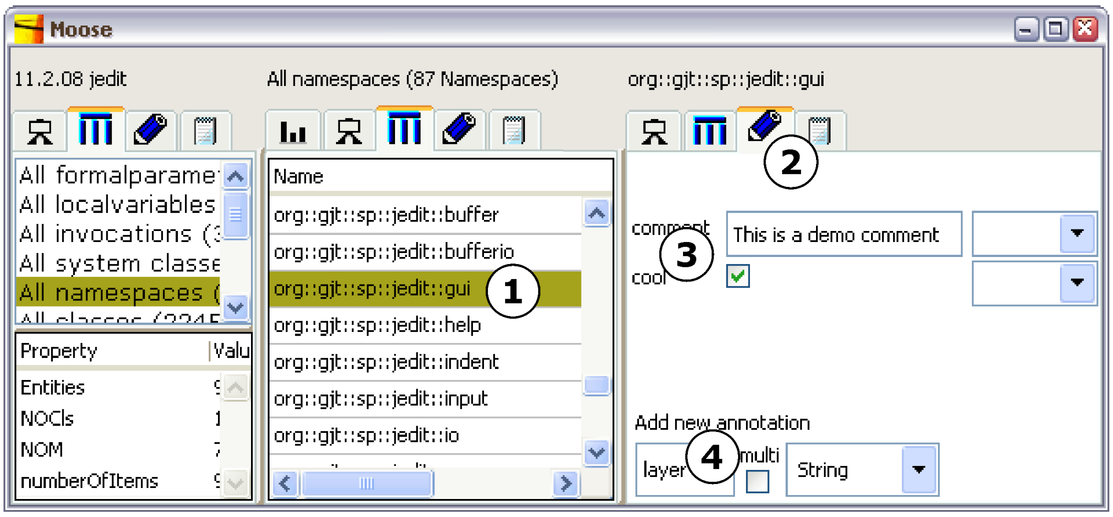
[1]http://dabbledb.com/

1.2 Our solution in a nutshell

In this dissertation we present an approach of integrating human knowledge into software analysis based on annotations. With annotations, information is attached to objects and the description of the structure of this information is stored in annotation descriptions.

As a proof of concept, we implemented an annotation framework which we call *Metanool* with the focus on flexibility and ease-of-use as it is especially needed for reverse engineering. Our implementation is fully integratable in the Moose analysis platform [Nierstrasz *et al.*, 2005].

Figure 1.1 demonstrates how annotating works with our solution. It shows the model extracted by parsing the source code of JEdit[2], after it has been loaded in the Moose Browser. The figure shows that we have selected the org.gjt.sp.jedit.gui package (or namespace) (1).



All formalparame
All localvariables
All invocations (3
All system classe
All namespaces (

Property	Valu
Entities	9
NOCls	1
NOM	7
numberOfItems	9

Name
org::gjt::sp::jedit::buffer
org::gjt::sp::jedit::bufferio
org::gjt::sp::jedit::gui
org::gjt::sp::jedit::help
org::gjt::sp::jedit::indent
org::gjt::sp::jedit::input
org::gjt::sp::jedit::io

comment: This is a demo comment
cool: ✓
layer · multi ☐ · String

Figure 1.1: Metanool annotations in Moose: (1) Selected Namespace, (2) annotations tab, (3) list of annotations, (4) adding a new annotation

On the right hand side, we have the annotations tab open for this package (2). It contains the list of annotations (3). There are the two annotations comment and cool which have been described for the Namespace class and whose values can be edited here for the selected namespace.

To create a new annotation, we enter a name and a type at the bottom of the annotations tab (4). In Figure 1.1 we entered the name layer and the type String.

[2]A programmer's text editor. http://www.jedit.org

As a result, a new annotation appears in the annotations list, where we can edit its value, as shown in Figure 1.2 (1).

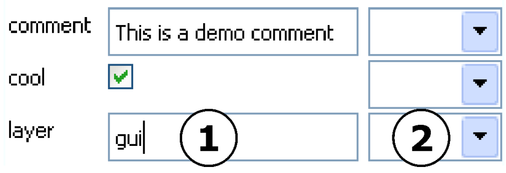


Figure 1.2: Editing the newly added annotation: (1) value editor, (2) options menu

On the right hand side of each annotation there is a menu (2) with options to remove values and to edit the description of the annotations. In Figure 1.3 we see the annotation description editor for the annotation layer that has been opened through the options menu. There, we can edit the name, the type and the multiplicity of the selected annotation description.

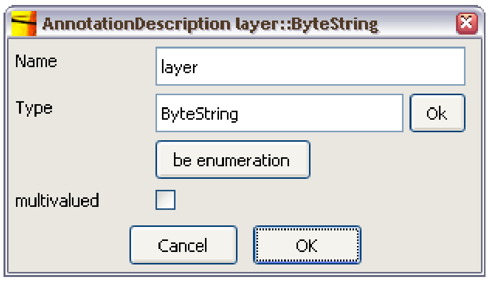


Figure 1.3: Editing an annotation description

Our framework can be used without the gui editor as well, by coding directly in its native programming language, Smalltalk. This is also very easy, because all we have to do is describe annotations and attach the descriptions to classes. Then, the values for all instances of these classes can be edited right away. The following lines of code show how the same annotation as before is added:

```
layer := (AnnotationDescription name: 'layer' type: String.).
FAMIXNamespace annotationDescriptions add: layer.

namespaceA annotations layer: 'gui'.
namespaceB annotations layer: 'model'.
```

Reading the values of the annotations can be done with the following code:

```
namespaceA annotations layer.
```

This is used every time we want to include the annotations in reports or further analysis.

1.3 Contributions

The contributions of this dissertation are:

- a generic approach for integrating external knowledge into the reverse engineering process,
- an implementation of this approach providing an extensible annotation model and editors for managing the annotations, and
- a set of examples of how our mechanism can be used to support existent reverse engineering approaches that take external knowledge into account.

1.4 Document structure

In **Chapter 2** (p.7) we present the state of the art in including external knowledge in reverse engineering.

In **Chapter 3** (p.13) we show how annotating works with our solution named *Metanool*.

In **Chapter 4** (p.25) we describe the implementation of *Metanool*.

In **Chapter 5** (p.29) we validate our approach by implementing various existing analysis techniques using *Metanool*.

In **Chapter 6** (p.43) we conclude.

In **Appendix A** (p.45) we explain how *Metanool* can be installed.

2 State of the art

There has been much research in automatic techniques for reverse engineering of software systems, but only few researchers have worked on capturing human knowledge in the various automatic approaches during the process of reverse engineering.

We first review the literature that deals with automatic reverse engineering techniques and we argue for the need to complement them with external knowledge. We then present some approaches that take specific external information into account but do it in a hard-coded and dedicated way. Last, we argue for the need to offer a generic mechanism for capturing external knowledge and we review some approaches that can offer such a mechanism.

2.1 The need to consider external knowledge in the reverse engineering process

Formal Concept Analysis. Formal Concept Analysis (FCA) allows us to identify concepts of elements and properties. It has been used by various researchers to mine abstractions from code, for example to maintain, detect and understand inconsistencies in class hierarchies [Godin *et al.*, 1998; Snelting and Tip, 1998; Huchard *et al.*, 2000]. Various approaches use FCA for detecting design patterns [Tonella and Antoniol, 1999], implicit architectural constraints and conventions [Arévalo *et al.*, 2004] and the structure and collaborations inside classes [Dekel, 2003; Arévalo *et al.*, 2003]. These approaches help in detecting concepts by considering the formal structure of elements but require further examination.

Crosscutting concerns. The detection of crosscutting concerns has been done by studying which parts of a program change at the same time [Breu and Zimmermann, 2006]. Another approach has been presented where commonalities in dynamic traces are identified [Breu and Krinke, 2004]. Gîrba *et al.* proposed detecting crosscutting concerns by analyzing how parts of the system change at the same time using FCA [Gîrba *et al.*, 2007]. FCA was also used to identify traits candidates in object-oriented code [Lienhard *et al.*, 2005]. Marin *et al.* have introduced a technique for identifying aspects by using a fan-in analysis [Marin *et al.*, 2007].

Feature identification. Feature identification determines which parts of the source code are executed for which feature [Wilde and Scully, 1995]. Eisenbarth *et al.* worked on feature identification based on a semi-automatic technique using static and dynamic analysis and FCA [Eisenbarth *et al.*, 2003]. The authors emphasize the need for the human input to refine the findings. A more fine-grained approach works at the level of statements instead of methods [Koschke and Quante, 2005]. Greevy also identified features with dynamic analysis and introduced a feature affinity measurement which automatically calculates which parts of the source code are used in how many features using dynamic analysis [Greevy, 2007]. The author also states that as feature definitions are fuzzy by nature, the results of the automatic feature identification need to be verified manually.

Group memory. Čubranić *et al.* developed a "group memory" in the form of a searchable database with artifacts related to a software system [Cubranic and Murphy, 2003]. They link source code with bug reports, news messages and external documentation with a structured meta-model. With this, the reverse engineer is automatically provided with the most relevant artifact for a specific task. Depending on the size and complexity of the system, the number of the proposed artifacts might however be immense. Human input about the suitability of artifacts to certain tasks could improve this approach.

Latent Semantic Indexing. Latent Semantic Indexing (LSI) is a technique to extract linguistic topics from a set of documents. It was used for software analysis first by Maletic and Marcus to categorize the source code files of the Mosaic web browser [Maletic and Marcus, 2000]. Other researchers then continued working with LSI to detect high-level conceptual clones [Marcus and Maletic, 2001] and to find concepts in the code [Marcus *et al.*, 2004; Kuhn *et al.*, 2007]. LSI was also used to detect links between source code and documentation [Marcus and Maletic, 2003] and to compute the class cohesion as a result of the semantic similarity of the methods of a class [Marcus and Poshyvanyk, 2005]. When we identified concepts, still some classes were misplaced. Providing a possibility to integrate human knowledge would increase the quality of LSI.

Design flaws detection. One approach to identify design quality problems was through metrics [Lanza and Marinescu, 2006]. The authors explicitly mention that the results of automatic detection reveals structures that are *suspected* of being flawed, and that manual inspection is required to confirm the results.

Automatic reverse engineering techniques are a good way to quickly get much information about a system and to support the reverse engineer with hints of possible concepts, design flaws *etc.*. However, the results need to be checked and sometimes altered by a reverse engineer to be more precise. Often, there is no possibility to add knowledge of a developer to the automatic techniques. A generic approach of adding external information to any kind of analyzed objects and results would complement all these automatic analyses and drastically enhance the applicability of their results.

2.2 Approaches that include external knowledge in the analysis

Ontologies. Raţiu and Deissenboeck proposed an approach to link source code with ontologies. They build the ontology and the mapping iteratively and with human intervention. This mechanism has been used for detection of semantic defects like conceptual duplications [Raţiu and Deissenboeck, 2006] or mismatches between described concepts and their implementation [Raţiu and Juerjens, 2007]. It has been extended later on to take into account the diffusion of concepts in the source code [Raţiu and Deissenboeck, 2007]. Linking ontologies to source code is an excellent way of integrating domain knowledge into the analysis and structuring it iteratively.

Reflexion models. Reflexion models are an approach to recovering a system's architecture [Murphy *et al.*, 1995; Koschke and Simon, 2003]. The developer knowledge is encoded in a model and then manually mapped to the source code and iteratively refined. Automatic clustering has been proposed recently which complements the approach by automatically detecting candidate components in the source code [Christl *et al.*, 2005].

The approach of reflexion models lives from the interaction of computation and human knowledge. The approach emphasizes that both parts are needed to not only complement each other but also to enhance one another.

Dedicated queries. A query-based approach to recover architectural elements was proposed by Pinzger *et al.* [Pinzger *et al.*, 2002]. The reverse engineer is required to define the rules of interest in queries. Mens *et al.* proposed a similar approach called Intensional Views, where Prolog-like queries are encoded by the reverse engineer to link external information to the source code [Mens *et al.*, 2002]. Both approaches take a dedicated kind of human knowledge into account for the analysis.

All approaches in this section include external knowledge into the analysis, but they do it only in a hard-coded and dedicated way that does not provide generic possibilities of integrating human knowledge in the reverse engineering process.

2.3 General approaches to encode external knowledge

Comments. Every programming language offers textual comments in the source code to support the encoding of external knowledge directly in the system [Gosling *et al.*, 2005; Stroustrup and Ellis, 1990]. They serve as documentation and should be visible to the analysis tool as they are an additional source of information. One drawback is, however, that they are unstructured and cannot be taken into account for further analysis. Another issue concerning reverse engineering is that we might not want to change the original source code to encode our knowledge about it. Depending on the analysis tool this is not even possible because analysis environments usually work with a model of the system, not with the system itself.

Annotations. A more advanced approach of annotating source code are Java Annotations[1]. They are meta-described by AnnotationTypes which allows for a structured handling. However, as Java Annotations are source code based, they cannot annotate objects as would be necessary when working with models.

As with comments, the source code itself needs to be changed if a class or any other code artifact is annotated. Every time an annotation description is changed, it has to be compiled and all annotations described by this AnnotationType have to be changed manually to conform to it again. Java Annotations also do not provide editors for annotating and for editing the metadescriptions.

[1]http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

Adaptive models. Adaptive Object-Models (AOM) [Yoder *et al.*, 2001; Yoder and Johnson, 2002] encode business entities in metadata instead of classes. Thus, whenever a business change is needed, the metadata is changed which is then immediately reflected in the running code. AOMs have only been proposed to solve changing business models in applications, but they could be applicable for modeling external knowledge during reverse engineering. This would solve the problem of inflexibility we have with Java annotations. Our solution is inspired by AOM in the way that not only the values but also the structure of our annotations can be changed at any time.

3 Annotating with Metanool

In the introduction we identified the requirements that an annotation framework for reverse engineering should support. We implemented an annotation framework, *Metanool*, that shows how annotating models of software systems improves the reverse engineering process. In this chapter we present *Metanool* with some examples of how a reverse engineer would work with it.

Metanool is implemented in Smalltalk [Goldberg and Robson, 1983] and has full support for being integrated in the Moose analysis platform [Nierstrasz *et al.*, 2005; Ducasse *et al.*, 2005]. Moose offers several analysis mechanisms and can be easily extended with custom plugins.

Figure 3.1 shows how the Moose browser works. When in a browsing area (1) one or more items are selected, another sector is opened next to it (2) with the available tabs for this selection, as for example a browsing tab, a source code tab or a Mondrian pictures tab. In this figure, the model of JEdit[1] has been loaded. First, "All system classes" is chosen and then the class org.gjt.sp.jedit.bsh.Parser is selected. For this class, the blueprint tab (3) is visible.

In this case, the blueprint [Ducasse and Lanza, 2005] reveals to the reverse engineer that the selected class is rather complex. Furthermore, we might be a bit surprised that a parser is in the bsh package. We are interested in this class now and do not want to forget to have a look at it later on. For that purpose, we want to attach an annotation to that class to remember it.

3.1 Creating our first annotation

Figure 3.2 shows *Metanool* as an annotation tab (1) in the Moose Browser. It contains the list of annotations (2) for the selected element. In this picture, the list already contains the annotation comment. We will show in this section how an annotation named interesting is added to remember the class org.gjt.sp.jedit.bsh.Parser.

[1]A programmer's text editor (`http://www.jedit.org`)

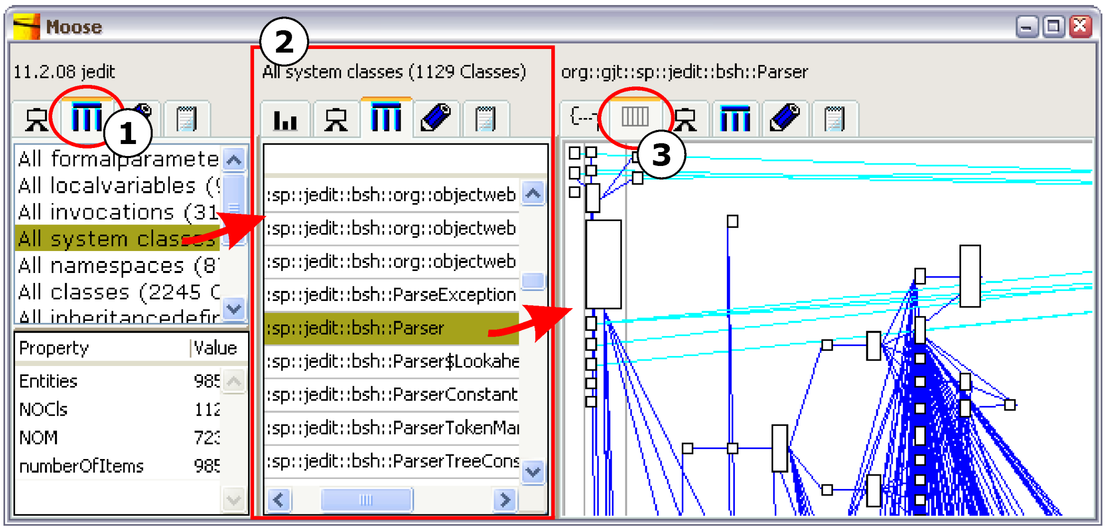

Moose
11.2.08 jedit
All system classes (1129 Classes)
org::gjt::sp::jedit::bsh::Parser
1
2
3
All formalparamete
All localvariables (
All invocations (31
All system classes
All namespaces (8
All classes (2245 C
All inheritancedefir
Property | Value
Entities 985
NOCls 112
NOM 723
numberOfItems 985
:sp::jedit::bsh::org::objectweb
:sp::jedit::bsh::org::objectweb
:sp::jedit::bsh::org::objectweb
:sp::jedit::bsh::ParseException
:sp::jedit::bsh::Parser
:sp::jedit::bsh::Parser$Lookahe
:sp::jedit::bsh::ParserConstant
:sp::jedit::bsh::ParserTokenMa
:sp::jedit::bsh::ParserTreeCons

Figure 3.1: Moose: (1) browsing meta elements, (2) browsing classes, (3) blueprint of selected class

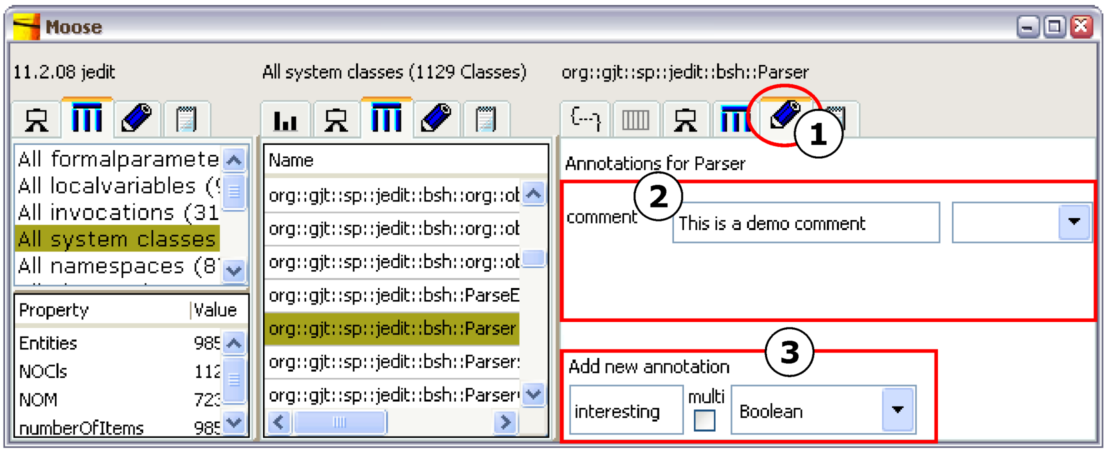
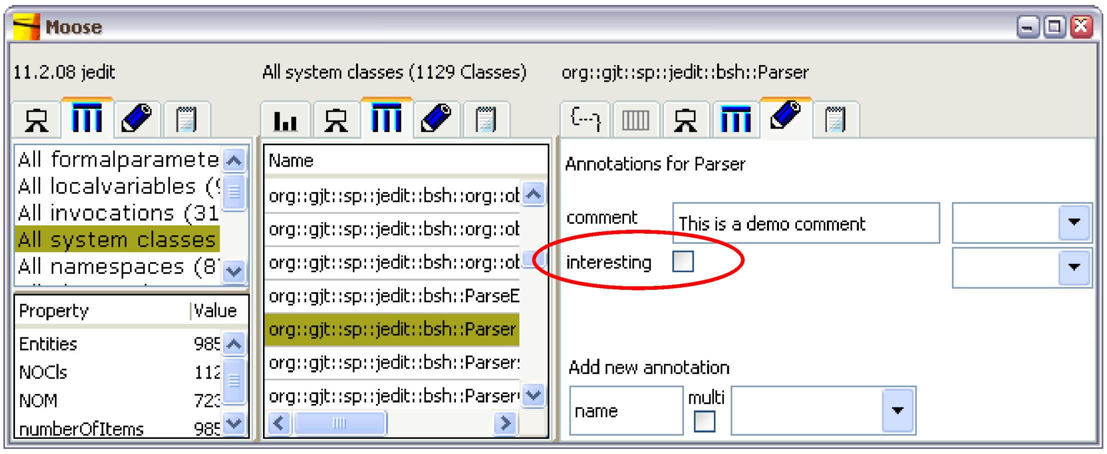

Moose
11.2.08 jedit
All system classes (1129 Classes)
org::gjt::sp::jedit::bsh::Parser
1
2
3
All formalparamete
All localvariables (
All invocations (31
All system classes
All namespaces (8
Property | Value
Entities 985
NOCls 112
NOM 723
numberOfItems 985
Name
org::gjt::sp::jedit::bsh::org::ol
org::gjt::sp::jedit::bsh::org::ol
org::gjt::sp::jedit::bsh::org::ol
org::gjt::sp::jedit::bsh::ParseE
org::gjt::sp::jedit::bsh::Parser
org::gjt::sp::jedit::bsh::Parser
org::gjt::sp::jedit::bsh::Parser
Annotations for Parser
comment
This is a demo comment
Add new annotation
interesting
multi
Boolean

Figure 3.2: Metanool annotations in Moose: (1) annotations tab, (2) list of annotations, (3) adding a new annotation

To create a new annotation for the selected class, we enter a name and a type at the bottom of the annotations tab (3). In this case, we entered the name interesting and the type Boolean. The new annotation then appears in the annotations list, as shown in Figure 3.3. There we can set interesting to true or false with a check box. The annotation is not only available for org.gjt.sp.jedit.bsh.Parser but for all classes now, because the description has been added to the metadescription of classes in Moose, namely FAMIXClass. Like this, we automatically have a check box for each class to say if it is an interesting class.


Moose

11.2.08 jedit

All formalparamete
All localvariables (
All invocations (31
All system classes
All namespaces (8

Property	Value
Entities	985
NOCls	112
NOM	723
numberOfItems	985

All system classes (1129 Classes)

Name
org::gjt::sp::jedit::bsh::org::ol
org::gjt::sp::jedit::bsh::org::ol
org::gjt::sp::jedit::bsh::org::ol
org::gjt::sp::jedit::bsh::ParseE
org::gjt::sp::jedit::bsh::Parser
org::gjt::sp::jedit::bsh::Parser
org::gjt::sp::jedit::bsh::Parser

org::gjt::sp::jedit::bsh::Parser

Annotations for Parser

comment: This is a demo comment
interesting ☐

Add new annotation
name
multi ☐

Figure 3.3: Editing the newly added annotation

The same can be done without the annotations editor as well, by using the native programming language of *Metanool*, Smalltalk. This is also very straightforward, because all we have to do is describe annotations and attach the descriptions to classes. Then, the annotations for all instances of these classes can be edited right away. The following lines of code show how the same annotation as in the gui editor is added:

```
interesting := (AnnotationDescription name: #interesting type: Boolean).
FAMIXClass annotationDescriptions add: interesting.
```

The values are set with the following line of code:

```
aFAMIXClass annotations interesting: true.
```

3.2 Managing annotation types

For annotations to be edited in gui editors, each annotation has a certain type. Boolean annotations then have a check box, String annotations an input field etc. Furthermore, annotation types permit in the first place the reuse of annotations in the analysis because they define what kind of values are valid. An annotation type is therefore responsible for two things:

- The type should provide a gui element for editing a value
- The type must know if a value is allowed for it or not

Our approach to typing the annotations is flexible and we are not restricted to only traditional types such as Boolean as used in our example above. In our implementation we also accept types corresponding to any Smalltalk class, as for example Color.

Furthermore in *Metanool*, not only classes can be types. For example, enumerations can be types as well. In this case, the type will be a collection of values. Then, not a class but a given collection is the type. An enumeration validates if a value corresponds to it or not by checking if the value is included in its elements. The gui editor is implemented on the instance side instead of the class side as for class types and consists of a drop down menu with the enumerated items.

As an example, we could use an enumeration to create a layer annotation which tells for a package which architectural layer it belongs to (see Section 5.1 (p.29)). In Figure 3.4 we see such an annotation with the type #('ui' 'model' 'persistence'), a Smalltalk collection of three Strings. The enumeration could also contain elements with distinct types because type checking is done by the enumeration as stated above.

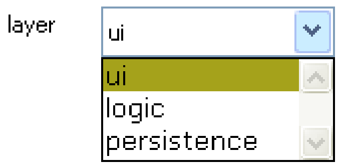
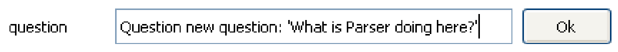


Figure 3.4: Enumeration

Introducing a new type. For the class org.gjt.sp.jedit.bsh.Parser, we not only want to remember that it is interesting, but we might also want to write down a question that came to our mind when we discovered it. This is motivated by the reverse engineering pattern “Tie code and questions” [Demeyer *et al.*, 2002], where the reverse engineer is encouraged to annotate the code with what he does not understand. So then, we need an annotation that handles questions and answers. In this section, we show how a new annotation type for questions is introduced.

We create a class Question with the String attributes question, answer and a Boolean done. This class is now already usable as a type. In the following picture we can see an annotation with the type Question, with no gui editor specified yet. In the input field, we can write the code to create a Question object, as seen in Figure 3.5.



```
Question new question: 'What is Parser doing here?'
```

Figure 3.5: Generic editor

If we want to tailor an own editor for the class Question, we need to implement the method mnEditPane. The sourcecode of a possible editor is shown below. It assembles two text input fields with the labels 'Q' and 'A' for the question and the answer and a check box labeled 'ok'.

```
Question > mnEditPane

   | form label inputFieldD inputFieldM inputFieldY |
   form := Widgetry.Form new.


   label := Widgetry.DisplayLabel string: 'Q'.
   label frame: ((Widgetry.FractionalFrame fractionLeft:0 right:0 top:0 bottom:1)
      rightOffset:10).
   form addComponent: label.

   inputFieldD := String mnEditPaneFor: self question.
   inputFieldD frame: ((Widgetry.FractionalFrame fractionLeft:0 right:0.45 top:0
      bottom:1) leftOffset:11).
   inputFieldD when: ValueChanged do: [:change |
      form announce: (ValueChanged
         from: self
         to: (self question: (String mnEditReturnValue: (change newValue)))) ].
   form addComponent: inputFieldD.


   label := Widgetry.DisplayLabel string: 'A'.
   label frame: ((Widgetry.FractionalFrame fractionLeft:0.45 right:0.45 top:0
        bottom:1) leftOffset:2; rightOffset: 12).
    form addComponent: label.

    inputFieldM := String mnEditPaneFor: self answer.
    inputFieldM when: ValueChanged do: [:change |
        form announce: (ValueChanged
            from: self
            to: (self answer: (String mnEditReturnValue: (change newValue)))) ].
    inputFieldM frame: ((Widgetry.FractionalFrame fractionLeft:0.45 right:1 top:0
        bottom:1) leftOffset:13; rightOffset: -31).
    form addComponent: inputFieldM.


    label := Widgetry.DisplayLabel string: 'ok'.
    label frame: ((Widgetry.FractionalFrame fractionLeft:1 right:1 top:0 bottom:1)
        leftOffset:-30; rightOffset: -15).
    form addComponent: label.

    inputFieldY := Boolean mnEditPaneFor: self isOk.
    inputFieldY when: ValueChanged do: [:change |
        form announce: (ValueChanged
            from: self
            to: (self isOk: (Boolean mnEditReturnValue: change newValue))) ].
    inputFieldY frame: ((Widgetry.FractionalFrame fractionLeft:1 right:1 top:0
        bottom:1) leftOffset:-15).
    form addComponent: inputFieldY.
    ^form
```

The following method defines what the editor should like if no value has been specified yet. In this case, a new object with empty values is asked for its gui editor.

```
Question class > mnEditPaneForNil
    ^(self new question:''; answer:''; isOk:false) mnEditPane
```

In Figure 3.6 we see the result of the above code, a special Question - editor.

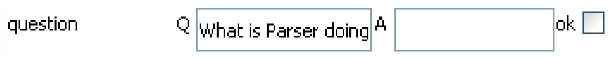
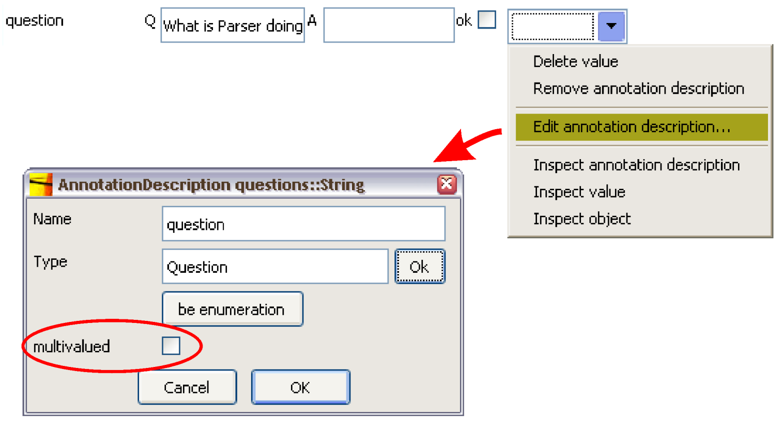


Figure 3.6: Question editor

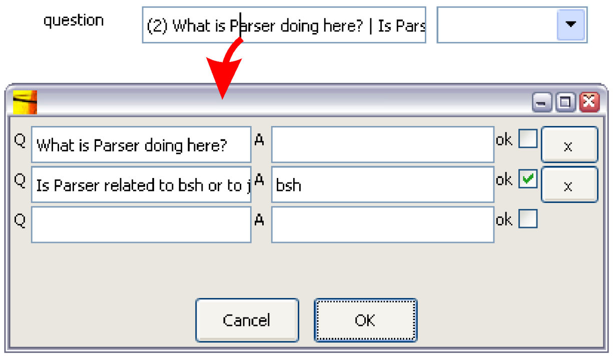
Making an annotation multivalued. In the case of a question annotation, the number of questions per object should not be limited to one. This requires that we can have multivalued annotations. In the annotation editor, we open the menu at the right hand side and choose “edit annotation description” as seen in Figure 3.7. This opens the annotation description editor, where we can select the check box for being multivalued.


question
Q What is Parser doing A ok
Delete value
Remove annotation description
Edit annotation description...
Inspect annotation description
Inspect value
Inspect object
AnnotationDescription questions::String
Name: question
Type: Question

multivalued: ☐

Figure 3.7: Making an annotation multivalued

Then the input field for this annotation opens a multi value editor when we click on it. There we can add as many questions and answers as we want to. As soon as the last line is filled with a value, a new line is added below (Figure 3.8).


question (2) What is Parser doing here? | Is Pars

Q	A	ok	
What is Parser doing here?		☐	x
Is Parser related to bsh or to j	bsh	✓	x
		☐	

Figure 3.8: Multivalued annotation

The Smalltalk code for transforming an annotation description to be multivalued is:

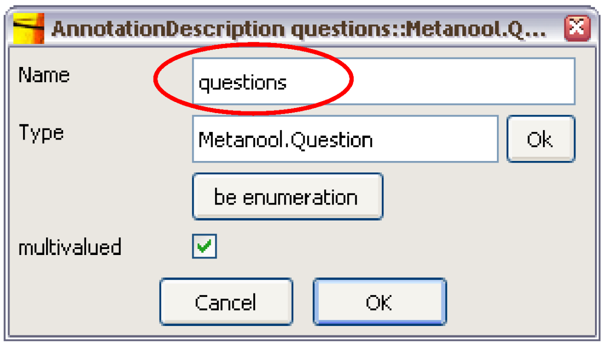
```
anAnnotationDescription beMultiValued.
```

The other direction is also possible, but going from multi-valued to single-valued naturally results in loss of information as the multiple values need to be dropped. In *Metanool*, the user is prompted for every object with more than one value to select which value he wants to keep.

3.3 Restructuring annotations

Renaming annotations. After having changed the multiplicity of our question annotation, its name is not exactly appropriate anymore. It should now be called questions. We can easily change an annotation description name in the editor as seen in Figure 3.9 or by executing:

```
(AnnotationDescription named:'question') name: 'questions'.
```



Name: questions
Type: Metanool.Question
multivalued: ✓

Figure 3.9: Renaming annotations

Collecting annotation values into a drop down list. Sometimes we notice during the process of annotating that the annotations should be a bit different than what they are. For example, we find out that we have to type the same values a lot of times and that we would prefer to select them from a drop down menu. Such a change requires two things: (1) The type must be changed while already specified values must be kept, (2) all existing values must be collected to create an enumeration of them. *Metanool* supports automatic transformation into an enumeration and back. In Figure 3.10 we show what such a transformation looks like.

Turning an enumeration type automatically back into a “normal” type requires knowing what the new type should be. *Metanool* selects the most specific common class or superclass of the elements in the enumeration. This enables the user to

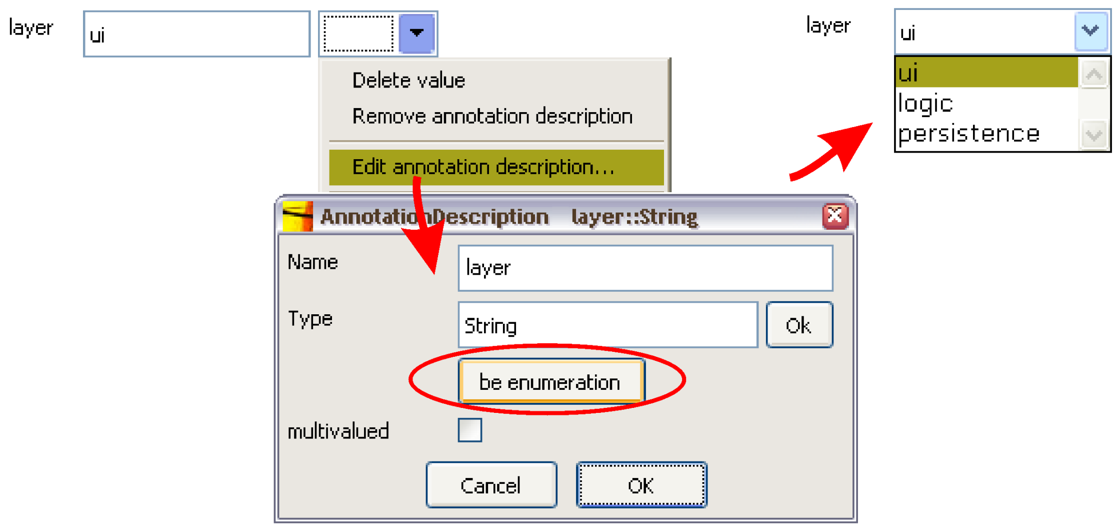


Figure 3.10: Collecting existing values

transform annotations without having to think about types more than absolutely necessary. If he wants to, he may choose another type though as we will see in the next section.

The Smalltalk code for turning an annotation type into an enumeration and back is the following:

```
anAnnotationDescription changeToEnumeration.
anAnnotationDescription changeToNonEnumeration.
```

Type transformation. As we have seen, an annotation description has a type which the actual annotation values must conform to. During the reverse engineering process, we might want to change the type of an annotation because we know in more detail what an annotation should look like as time goes on. For example in the case of a questions annotation as shown in Section 3.2 (p.16), the user might not have had the Question type from the outset but started with a String annotation.

Suddenly, he realizes that it would be better to have a Question type to be able to specify questions and answers and to tell if it is still an open issue. So he creates the new type as described in Section 3.2 (p.16). The next step is then to change the type of the question annotation from String to Question. The values that already exist in the system should not be thrown away but be transformed as well as possible into questions.

For the existing types in *Metanool*, transformation strategies have been specified. As the reverse engineer introduces a new type, he can add transformation strategies for this type where it is needed. In the following lines we show the method that allows changing the type from String to Question.

```
Question class mnFromString: aString
   ^self new question: aString
```

In Figure 3.11 we show the modification of an annotation description from the type String to Question. This corresponds to:

```
questionAnnotationDescription transformToType: Question.
```

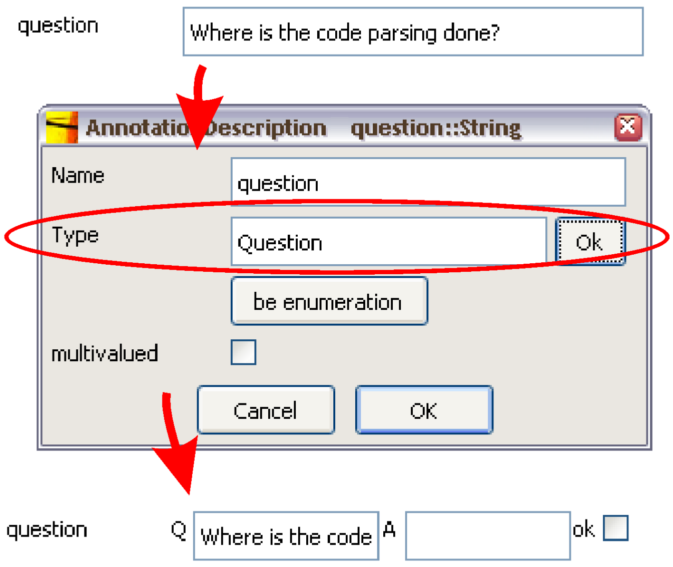


Figure 3.11: Type transformation from String to Question

To enable transformations from Question to other types, we need a base method mnFromQuestion: that is implemented in Object and that is overridden in all types that should be able to import questions. In Question, we need the method mnBe: that calls mnFromQuestion: from the target type. Here we show the needed code to permit the transformation from a Question into a String.

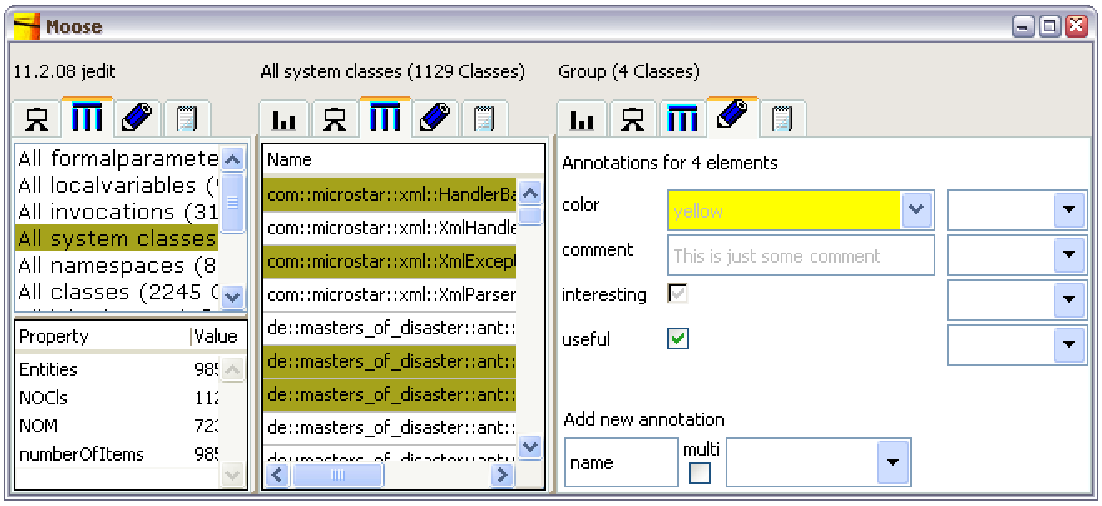
```
Question mnBe: aType
  ^aType mnFromQuestion: self

Object class > mnFromQuestion: aQuestion
  (self mnCanBe: aQuestion) ifTrue: [^aQuestion] ifFalse: [^nil]
```

```
String class mnFromQuestion: aQuestion
   ^aQuestion mnPrintString
```

3.4 Editing groups

Often, there is a need to set certain annotation values for a whole group of elements. *Metanool* offers editing annotations for groups in the same way as for single objects. The additional challenge for group editing is that the user should be able to know if all elements have the same value for a certain annotation or if they have distinct values. We did not want to show all distinct values in the group editor because this would have required a lot more space at the expense of clarity, but we considered it helpful to show at least one value for each annotation. If this value is common to all group elements, it is displayed as in the normal editor, and if it's not, it is painted pale-gray. In Figure 3.12, four classes are selected that have unequal values for the annotations color, comment and interesting. For the annotation useful, they all have the value true. If the user edits the value in the group editor, it is set for all selected elements.


Moose

11.2.08 jedit

All formalparamete
All localvariables ('
All invocations (31
All system classes
All namespaces (8
All classes (2245 C

Property	Value
Entities	985
NOCls	112
NOM	723
numberOfItems	985

All system classes (1129 Classes)

Name
com::microstar::xml::HandlerBa
com::microstar::xml::XmlHandle
com::microstar::xml::XmlExcept
com::microstar::xml::XmlParser
de::masters_of_disaster::ant::
de::masters_of_disaster::ant::
de::masters_of_disaster::ant::
de::masters_of_disaster::ant::

Group (4 Classes)

Annotations for 4 elements

color	yellow
comment	This is just some comment
interesting	✓
useful	✓

Add new annotation

name multi

Figure 3.12: Group editing

4 Metanool internals

This chapter provides a more detailed description of the *Metanool* internals.

Moose works with the EMOF meta-meta-model [Ducasse *et al.*, 2008], which can be used for meta-models as for example FAMIX. Each element of the meta-model is described by an instance of EMOF.Class. For example when working with the FAMIX meta-model, the instances of FAMIXNamespace model the source code namespaces. FAMIXNamespace is described by an EMOF.Class named 'FAMIXNamespace'. Each instance of EMOF.Class can have multiple instances of EMOF.Property which describe its attributes. For FAMIXNamespace, some of the properties are: classes, belongsTo, LOC (lines of code).

4.1 Annotations and their descriptions

The term "annotation" is a bit ambiguous: it can mean both the value that is attached to an object or the name for which an object can have a certain value. In *Metanool*, the latter is called "annotation description" and the values are simply called "values" (or "annotation values"). Where we do not need to distinguish between the values and the descriptions, we sometimes just say "annotation" in an informal way.

Figure 4.1 shows how annotation values and annotation descriptions are attached to objects. We attach them at the meta-level so that annotation descriptions are properties of an EMOF.Class and are available for all instances of the corresponding class. In this figure we describe the case where FAMIXNamespace objects are annotated, hence the annotation description layer is added to the EMOF.Class named 'FAMIXNamespace'.

To store the actual annotation values, we use the MetanoolRegistry. In this case, the namespace #jedit.gui has the value 'gui' for the annotation layer.

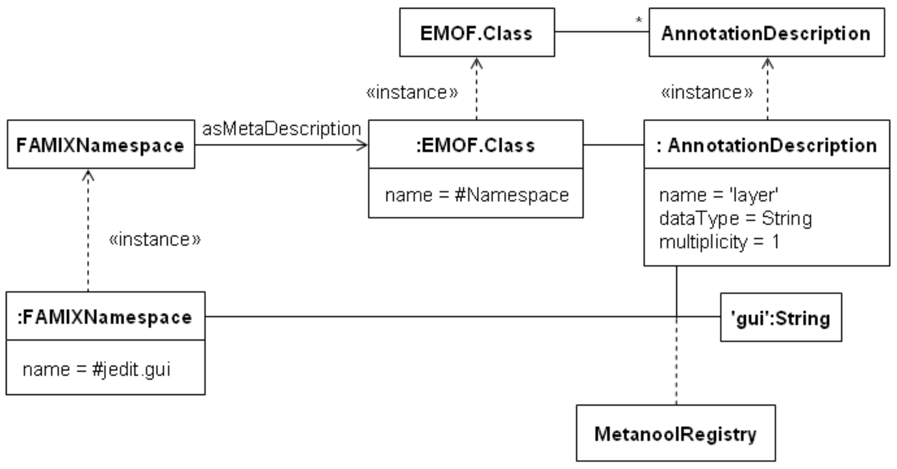


Figure 4.1: Annotation descriptions and values

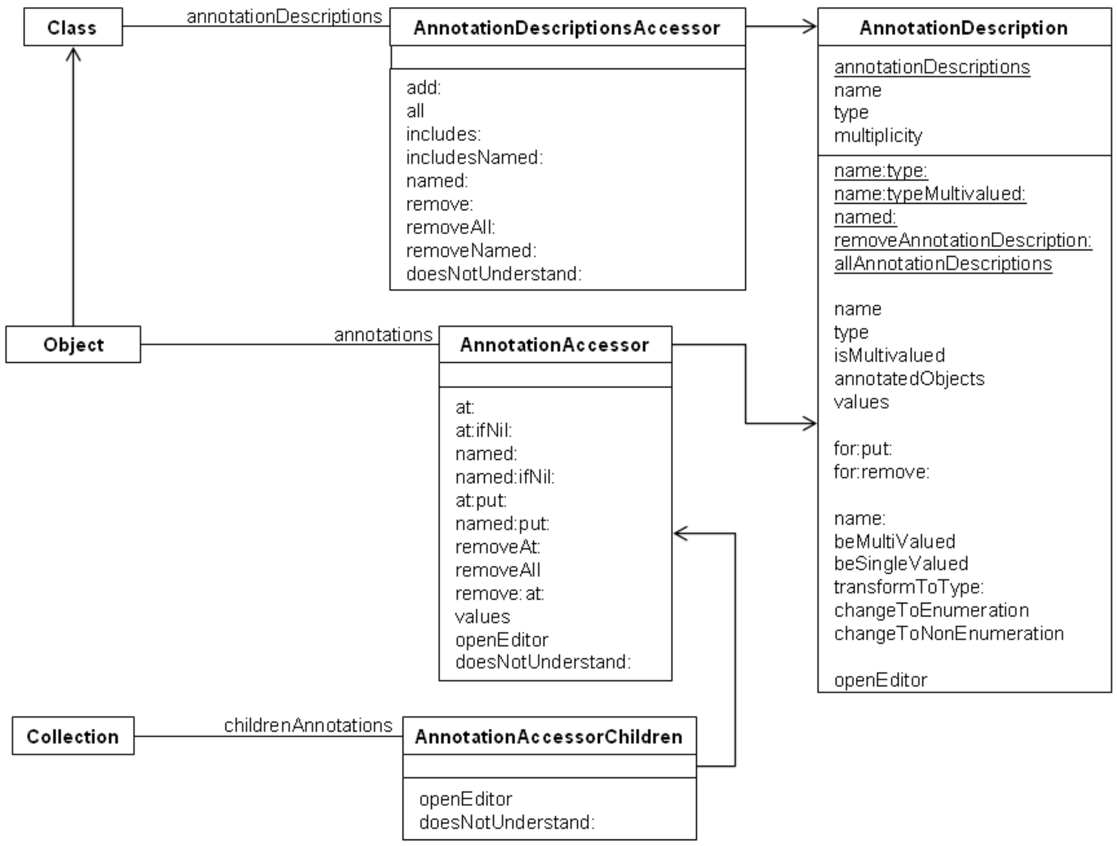


Figure 4.2: The public interface of Metanool

4.2 The annotation interface

In Figure 4.2 we see the design of the public interface of *Metanool*. To avoid conflicts in the interface of Object and to provide a simple way of accessing annotations, we created accessor classes for annotation descriptions and values. The AnnotationDescriptionsAccessor manages the annotation descriptions for a class, the AnnotationAccessor the values for an annotated object. AnnotationAccessor has not only annotation accessors but also two more methods: (1) openEditor opens the annotation editor which is the same as is integrated in the Moose browser but is a standalone editor; (2) doesNotUnderstand: transforms all method calls that consist of a valid annotation name into getters and setters for the annotation values. This allows us to access the annotation value in two different ways:

```
aNamespace annotations named: 'layer'    ''equals:''
aNamespace annotations layer

aNamespace annotations named: 'layer' put: 'gui'    ''equals:''
aNamespace annotations layer: 'gui'
```

The same is done in AnnotationDescription and in AnnotationDescriptionAccesser to get annotation descriptions in a quick way.

```
AnnotationDescription named: 'layer'
AnnotationDescription layer
```

AnnotationDescription basically has a name, a type and a multiplicity, and on the class side it has a list of all existing annotation descriptions in the system. It offers various transforming methods to change its name, type or multiplicity as described in the previous sections of this chapter.

At the bottom of Figure 4.2 we see that Collection has an additional accessor, AnnotationAccessorChildren. This accessor delegates all messages to the accessors of each of its element. Like this, the annotations of the children are edited by executing for example:

```
aNamespaceGroup childrenAnnotations layer:'model'
```

If for a collection, aCollection annotations is called instead of aCollection childrenAnnotations, the annotations for the collection object itself are edited. With openEditor, AnnotationAccessorChildren opens the group annotation editor.

4.3 The type interface

By calling anObject annotations openEditor, a MetanoolEditor is opened. The same editor is integrated in Moose. This editor shows a list of all available annotations for this object, where all values can be directly edited. The editors for each annotation are not created by the MetanoolEditor itself but by the types of the annotations. This makes annotation editing generic, as *Metanool* can be extended with new types without changing the core editor but by providing some gui methods directly in the type definition. In Figure 4.3 we see the methods that are responsible for type behavior (as explained in Section 3.2 (p.16)). There are methods for the gui editors, for type transformations and for type checking. All of them are prefixed by "mn" according to the Smalltalk convention to avoid conflicts in the Object interface. mnCanBe:aValue usually evaluates if the value is an instance of this class or of a subclass, but it can be overridden to provide a specific type check, for example for enumerations.

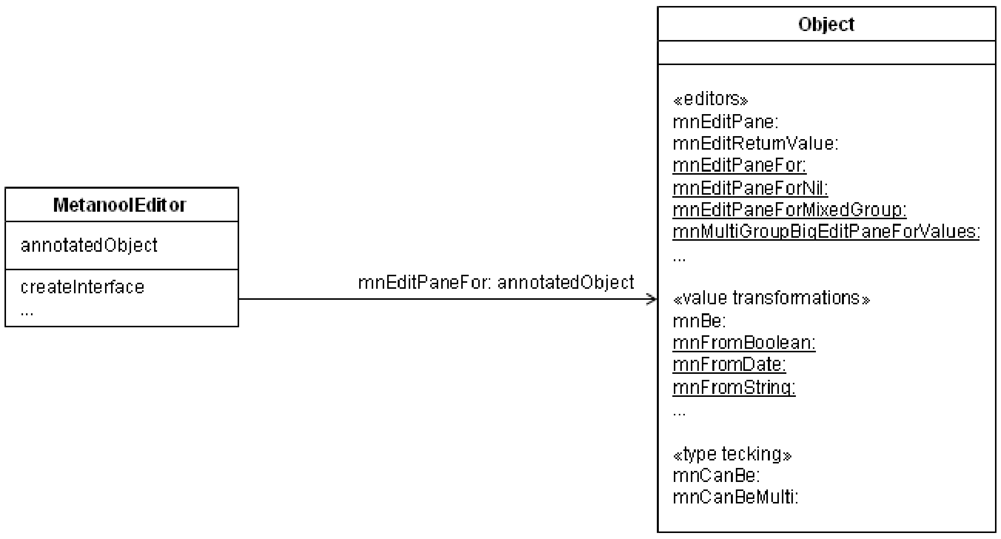


Figure 4.3: Methods related to types

5 Metanool Use Cases

To give an impression of the usefulness of our annotations approach, we describe some typical use cases from the world of reverse engineering.

The first use case examines an architecture recovery of a layered system which reveals violations of architectural restrictions. The second use case we describe is a task list implementation, which lets a programmer assign tasks to code artifacts and shows a list of all tasks inside a system. Then we show how automatic design flaw detection (*e.g.,* god classes, data classes, feature envy...) is improved by human validation. The final example is how to use our annotations to perform and validate the results of automatic feature analysis based on dynamic analysis of traces [Greevy and Ducasse, 2005b].

All of these use cases have been performed in Moose [Nierstrasz *et al.*, 2005] and the visualizations are painted with Mondrian [Meyer *et al.*, 2006].

5.1 Architecture Recovery

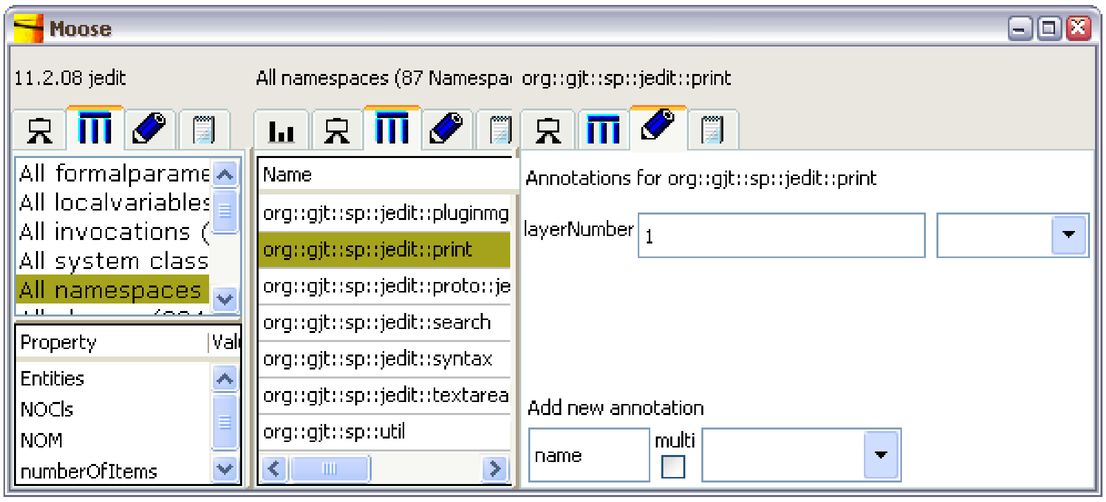
Many software companies define a standard architecture for a set of applications. When new applications are being developed, the software architects need to monitor the compliance of the software development with the architectural rules (see reflexion models [Murphy *et al.*, 1995; Koschke and Simon, 2003]). Architectures are typically not explicit in the source code. Packages are usually used to group classes according to their particular role or function in the system. However there may be multiple packages associated with an architectural layer, or a package may be associated with multiple layers. In the absence of a language construct to reflect architectural layering, our annotations provide an ideal way during analysis of a system to associate packages with layers and then to perform an analysis of the invocations to ensure that the architectural boundaries are being adhered to. A similar architecture analysis can be done with Sotoarc[1], a commercial software architecture analysis tool specialized on layer modelling.

[1] http://www.software-tomography.de/html/sotoarc.html

In this section we present two ways of using annotations for the package-level definition of layer membership. Both will annotate the packages with a layer annotation and then create a visualization of the layers and accesses between the layers. They will make visible the structure of the system and point out violations of the layered architecture. The model that we will use in this section is again the model of JEdit.

5.1.1 Way one: Layer numbers

The layered architecture of JEdit had been analyzed before by Patel *et al.* [Patel *et al.*, 2003]. The authors found 14 layers and wrote down for each layer which packages belong to it. What we did then was to include their knowledge into our tool by annotating the packages with their layer number. The annotation description had the name layerNumber, the type was Integer and the multiplicity was 1. Then each package received a layer number, according to the number that we found in the report. Figure 5.1 shows the annotation for the package org.gjt.sp.jedit.print that belongs to layer number one which is the topmost layer.



Moose

11.2.08 jedit

All namespaces (87 Namespa org::gjt::sp::jedit::print

All formalparame
All localvariables
All invocations (
All system class
All namespaces

Property	Val
Entities	
NOCls	
NOM	
numberOfItems	

Name
org::gjt::sp::jedit::pluginmg
org::gjt::sp::jedit::print
org::gjt::sp::jedit::proto::je
org::gjt::sp::jedit::search
org::gjt::sp::jedit::syntax
org::gjt::sp::jedit::textarea
org::gjt::sp::util

Annotations for org::gjt::sp::jedit::print

layerNumber 1

Add new annotation

name multi

Figure 5.1: Annotating packages with a layer number

The visualization seen in Figure 5.2 was painted with Mondrian based on the layer numbers. The small squares are the packages and the big rectangles are the layers. The packages that did not exist in the report because the authors used a different version of JEdit could not be annotated; all these are collected in the “rest” group. The visualization considers not only the external knowledge about which package belongs into which layer, but also architectural constraints about accesses between layers. One of these constrains is that accesses between layers should never go from a lower to a higher layer, *e.g.,* a class in the persistence layer should not call a gui class. We painted all accesses that go from bottom up in red to show that they are a violation of the architecture. Accesses from top to bottom are blue because they are allowed. Grey lines are drawn for all packages that could not be assigned to a layer.

Another constraint prohibits jumping over a layer, but as we can see in this picture, layers are jumped over very often and we doubt if this architectural rule can be applied to this classification of fourteen layers. Usually the number of layers is about three or four. But even though the model is not necessarily correct, the exercise was about being able to encode the external information.

Package names are not visible in Figure 5.2 because they would have needed a lot of space. However, when watching the visualization in the Mondrian view pane, package names are interactively shown when pointing at them with the mouse.

5.1.2 Way two: allowed / forbidden

Adding layer names. Another approach to model knowledge about layers in annotations is to add a String annotation to the packages that contain the name of their layer, *e.g.,* gui, model, data *etc.*. The following code does exactly this.

```
layer := (AnnotationDescription name: #layer type: String.).
FAMIXNamespace annotationDescriptions add: layer.

namespaceA annotations layer: 'gui'.
namespaceB annotations layer: 'model'.
aGroupOfNamespaces childrenAnnotations layer: 'data'.
```

All this could of course also have been done in the editor. Besides, this is a good candidate for changing the annotation type into an enumeration as described in Section 3.3 (p.20).

Adding constraints. Next, we define for each layer which layers are allowed or forbidden to access. These relations between layers are also modeled with annotations. For this, we need an annotation allowed and an annotation forbidden that are both attached to the class String. We will directly annotate the layer names which are instances of String. The type of the two annotations is also String, because we want to interrelate String objects. To be able to define multiple

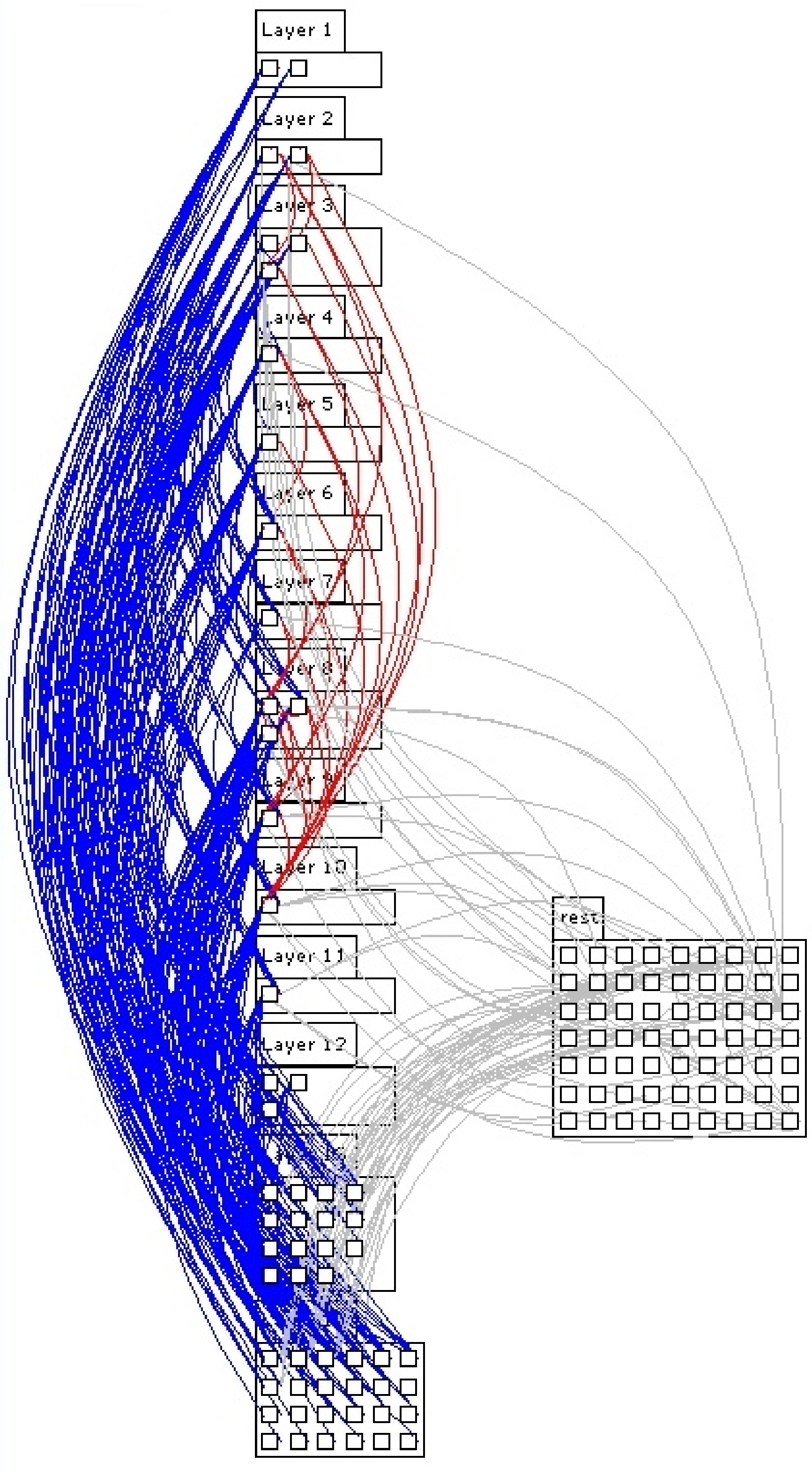


Figure 5.2: JEdit in 14 layers and their access violations (red)

forbidden and allowed layers for each layer, we make the annotations multi-valued as shown in the following code segment.

```
String addAnnotationType:
    (AnnotationDescription name: #allowed mutivaluedType: String)
String addAnnotationType:
    (AnnotationDescription name: #forbidden mutivaluedType: String)
```

Then the layer names are annotated like this:

```
'ui' annotations allowed: 'model'
'persistence' annotations forbidden: 'model'
'persistence' annotations forbidden: 'ui'
```

Figure 5.3 gives an overview of the process of annotating the layer annotations. The picture shows three packages that are annotated with a layer annotation; two of them belong to the "gui" layer and one of them to the "persistence" layer. The "persistence" layer is in turn annotated with the value "gui" for forbidden.

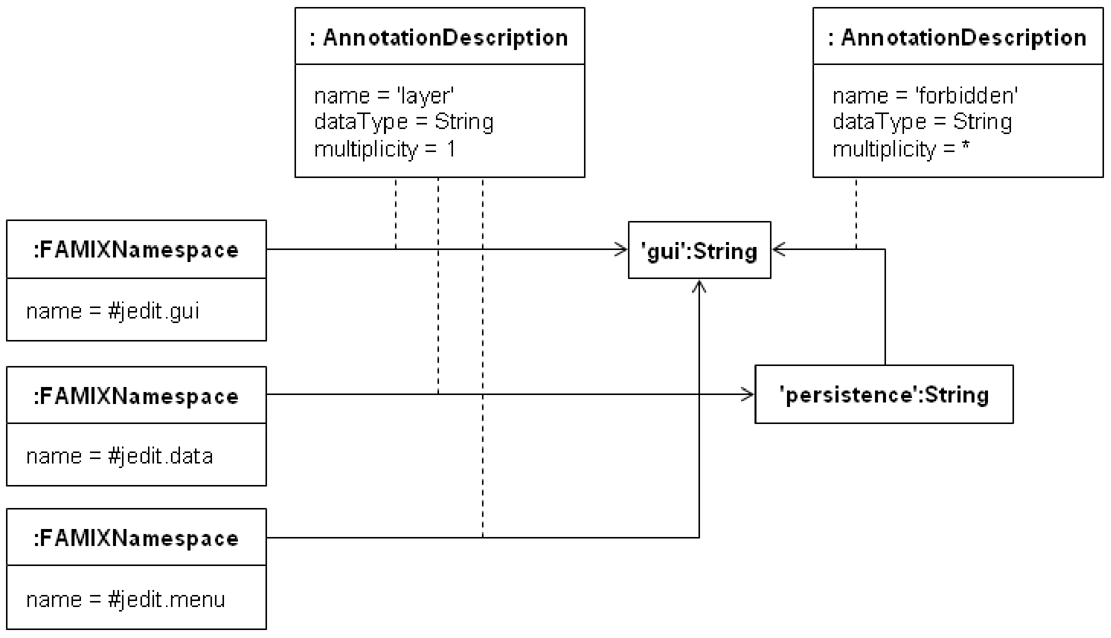


Figure 5.3: Annotating namespaces and their annotations

Creating a visualization. With this information, we can again create a Mondrian painting where we paint all namespaces grouped by layers (Figure 5.4). The invocations between the namespaces are now painted in a color according to the "allowed" and "forbidden" annotations of their layers. The gray lines mean that nothing has been specified. We do not at all claim that this picture shows the truth about JEdit, because the layer annotations were added without deep knowledge of the system. The important point is that we created a possibility of enhancing the analyzed model with architectural knowledge that could not be extracted from the code itself.

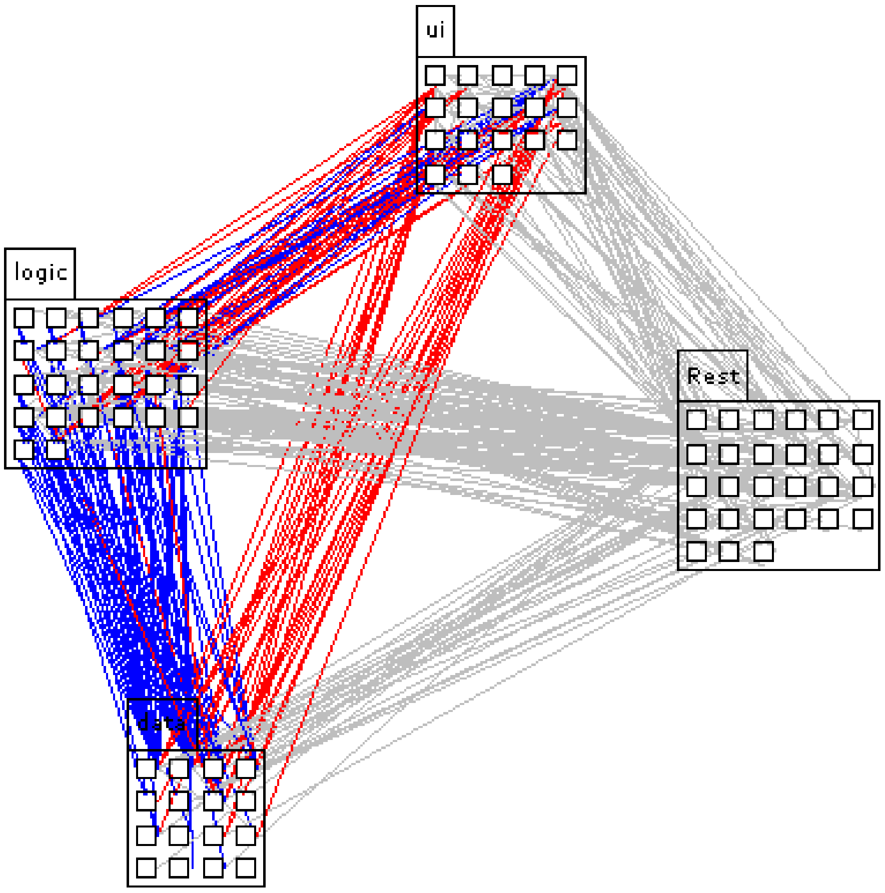


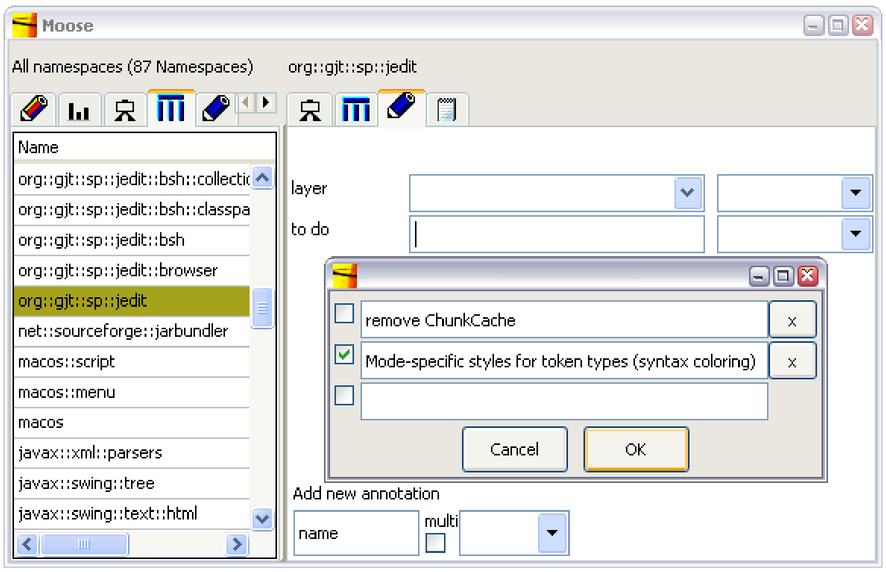
Figure 5.4: JEdit in four layers and their access violations (red)

5.2 Checklist

In this use case, tasks are added to code artifacts and a list of all tasks in the system is generated. This is similar to the @TODO annotation in Java. A checklist is useful for example if during reverse engineering, the user detects issues that he wants to address later on. He can add tasks directly to the affected part of the system.

Introducing a task type. When we want to attach tasks to some elements, maybe classes and namespaces (packages), we introduce a new data type: ToDo. A new class ToDo is created that has two instance variables: a String named "task" and a Boolean named "done". Like this, each ToDo has a task description and can be marked as done. This new type is introduced the same way as Question in Section 3.2 (p.16).

Editing the tasks. After having created a ToDo data type, we add a toDo AnnotationDescription to the elements we want to annotate. The AnnotationDescription will have the name toDo, the type ToDo and be multivalued. Like this, every element can have as many tasks as needed. Then, we can annotate our elements as seen in Figure 5.5. We can add the same AnnotationDescription to several elements, *e.g.,* FAMIXClass and FAMIXNamespace.

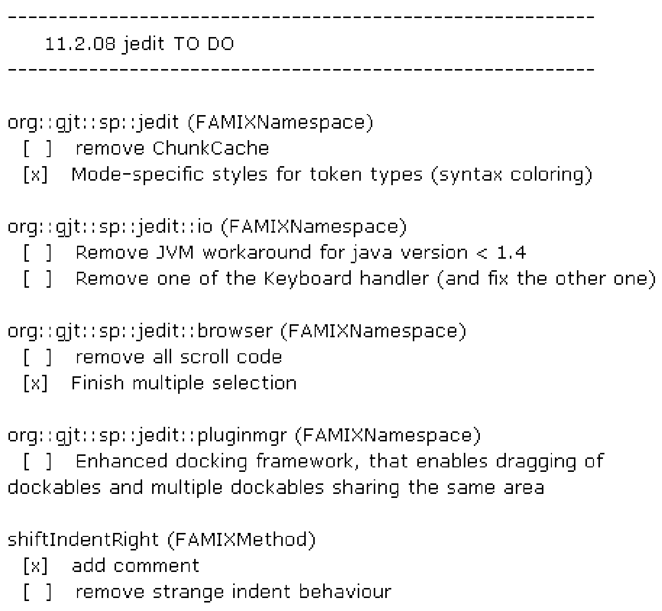


org::gjt::sp::jedit::bsh::collectic
org::gjt::sp::jedit::bsh::classpa
org::gjt::sp::jedit::bsh
org::gjt::sp::jedit::browser
org::gjt::sp::jedit
net::sourceforge::jarbundler
macos::script
macos::menu
macos
javax::xml::parsers
javax::swing::tree
javax::swing::text::html

[ ] remove ChunkCache
[✓] Mode-specific styles for token types (syntax coloring)

Figure 5.5: Editing tasks

Printing a checklist. With this information, we can easily print a checklist of all tasks in the analyzed model, as we can see in Figure 5.6.



```
-------------------------------------------------------
   11.2.08 jedit TO DO
-------------------------------------------------------

org::gjt::sp::jedit (FAMIXNamespace)
  [ ]   remove ChunkCache
  [x]   Mode-specific styles for token types (syntax coloring)

org::gjt::sp::jedit::io (FAMIXNamespace)
  [ ]   Remove JVM workaround for java version < 1.4
  [ ]   Remove one of the Keyboard handler (and fix the other one)

org::gjt::sp::jedit::browser (FAMIXNamespace)
  [ ]   remove all scroll code
  [x]   Finish multiple selection

org::gjt::sp::jedit::pluginmgr (FAMIXNamespace)
  [ ]   Enhanced docking framework, that enables dragging of
dockables and multiple dockables sharing the same area

shiftIndentRight (FAMIXMethod)
  [x]   add comment
  [ ]   remove strange indent behaviour
```

Figure 5.6: Generated Tasklist

5.3 Detection strategies

Another reverse engineering use case of annotations is based on detection strategies. Detection strategies define rules that automatically detect candidate classes that represent design flaws in a system, such as god classes, data classes, brain classes and other code smells [Lanza and Marinescu, 2006]. The shortcoming of automatically applying detection strategies there is that the results are not always correct. As a reverse engineer performing the analysis, we might want to encode our knowledge of the system into the detection strategy analysis mechanism to indicate which results represent false positives or false negatives before generating a report on the design of a system. This is a typical case where automatically and manually detected information need to be merged together. We claim that annotations are the right place to store this kind of information.

We explain our approach taking with the example of the detection of god classes. First, we run the automatic detection and receive a set of candidate god classes. We add the annotation description auto god class (type: Boolean) to classes. For our group of detected god classes, we set the value to true. Then, we add a second annotation description manual god class. There, we set the values manually for god classes that were not automatically detected but are considered by the reverse engineer as being god classes. A report engine can now take both annotations into account.

For example, an overview as seen in Figure 5.7 can be generated. It is a visualization that shows classes as squares and inheritance definitions as lines. In this system overview, classes are represented by rectangles in different sizes: large rectangles indicate god classes, whereas the small rectangles indicate non-god classes. Both the automatic and the manual detection are taken into account. We use color to represent how the decision was taken if a class is a god class or not. Automatically detected god classes without a manual confirmation are painted gray. Red indicates that the automatic detection has been confirmed manually. Orange classes have not been detected automatically but have been stamped manually as being god classes. For the non-god classes that are all drawn as small squares, we distinguish white and blue squares: blue means that they have been detected by the automatic detection strategy but have been refused by the reverse engineer.

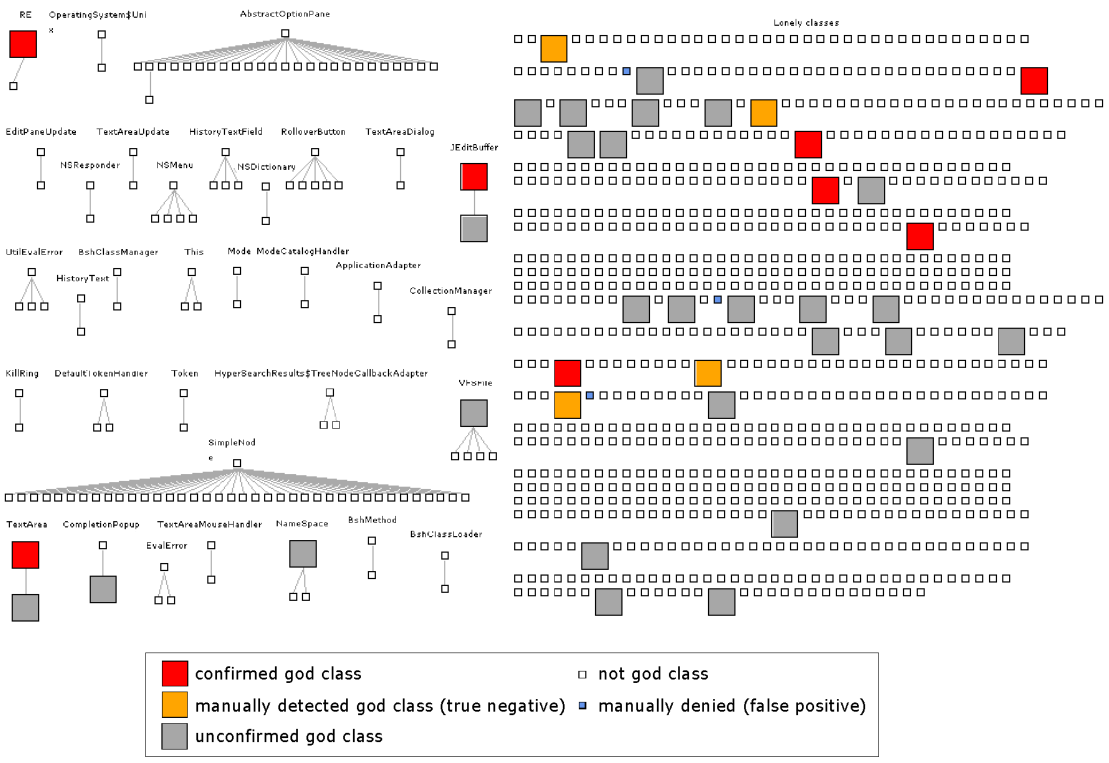


Figure 5.7: A god class report combining automatic and manual detection.

We can also just print a list with all god classes and sort them by “true positive”, “false positive”, “false negative” etc. We can also simplify the result and just make a list of all classes that from the combination of manual and automatic detection are considered as being god classes. The new list of god classes can be used in the same ways as the original automatically detected list, only with the advantage that it is more precise because human knowledge has been integrated.

5.4 Feature analysis

Features are abstractions that encapsulate knowledge of a problem domain and describe units of system behavior [Greevy, 2007]. Several researchers have identified the potential of exploiting features in reverse engineering [Eisenbarth *et al.*, 2003] [Antoniol and Guéhéneuc, 2005] [Greevy, 2007]. Feature identification approaches (*e.g., Software Reconnaissance* [Wilde and Scully, 1995]) describe various techniques for locating which parts of the code implement a given feature. Automatic approaches to feature identification are typically based on dynamic analysis where the features are executed on an instrumented system and the traces of all message sends are captured.

Feature analysis techniques are limited because features are abstractions, thus a feature representation in the model is an approximation by nature. This means that the results of feature analysis are influenced on the one hand by how the features are executed during the tracing phase and on the other hand by the number and choice of features traced.

In this section we show how the use of our *Metanool* annotations support feature analysis by addressing the above problem.

Feature analysis results. We performed a feature analysis on the Moose system with 8 specified features using a Dynamix model of Moose. Then we applied the *feature affinity* metric to the classes based on the results to quantify which classes belong to only one feature and which classes are used for many features.

Figure 5.8 shows a visualization of 8 *feature views* of our model [Greevy and Ducasse, 2005a]. For all 8 features, we have a rectangle that contains colored small rectangles which are classes. The classes are grouped and colored according to the *feature affinity* metric (for more details see [Greevy, 2007]). For example the class VisualWorksParseTreeMetricsCalculator is cyan because it was only used in the *computeMetrics* feature (single feature affinity). The yellow classes (low group affinity) (2) were touched by a low number of features. More general functionality is provided by the orange high-group-affinity (3) and the red infrastructural (4) classes.

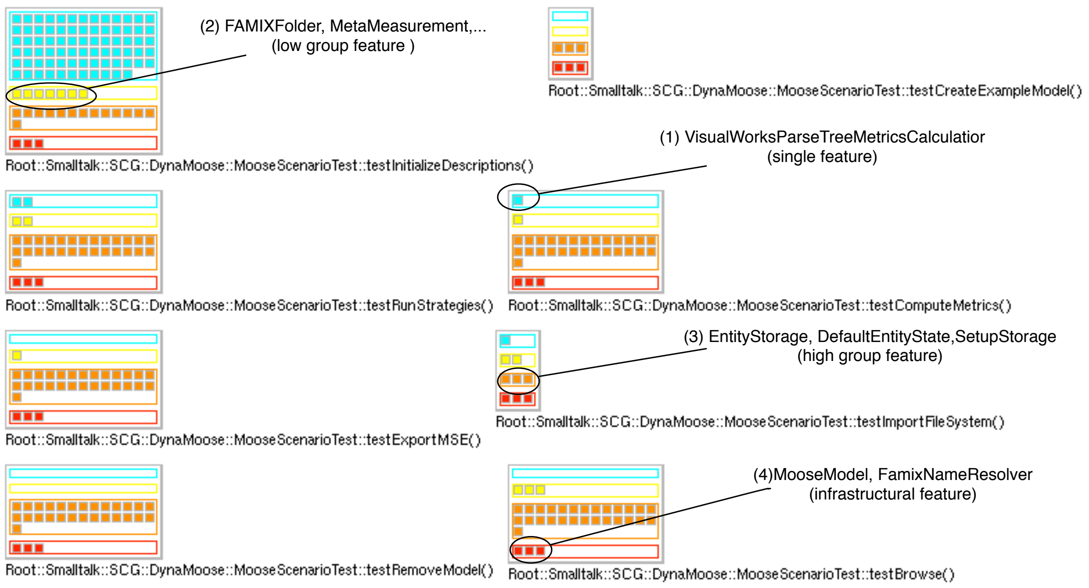

(2) FAMIXFolder, MetaMeasurement,...
(low group feature)
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testInitializeDescriptions()
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testCreateExampleModel()
(1) VisualWorksParseTreeMetricsCalculatior
(single feature)
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testRunStrategies()
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testComputeMetrics()
(3) EntityStorage, DefaultEntityState,SetupStorage
(high group feature)
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testExportMSE()
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testImportFileSystem()
(4)MooseModel, FamixNameResolver
(infrastructural feature)
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testRemoveModel()
Root::Smalltalk::SCG::DynaMoose::MooseScenarioTest::testBrowse()

Figure 5.8: 8 Feature Views of Moose showing the feature affinity values of classes

Validation of the results. Our feature views reveal three classes that have been automatically detected as 'infrastructural'. The developers of Moose deny this result. They state that the class FamixNameResolver does not implement infrastructural functionality but is specific to importing a moose model from Smalltalk source code, which is another feature that has not been considered in the analysis. Another one of the three detected infrastructural classes is called UNKNOWN, which represents a class that could not be identified and does not exist in the system.

This developer knowledge reveals two important facts to the reverse engineer: Firstly, the behavior of importing Smalltalk models needs to be treated as a distinct feature, and secondly, the behavior of the executed features was not well delimited. They all used Smalltalk source code import which made the class FamixNameResolver appear as an infrastructural class.

However, we do not want to throw away the results of dynamic feature analysis just because we have detected this false positive in 'featureAfffinity' assignment. Our feature views, though they are approximations, reveal other interesting information about the features of the Moose system. Instead we choose to refine the feature representations using annotations.

We create an annotation named feature-affinity with an enumeration type #('none' 'single feature' 'low group' 'high group' 'infrastructural') (see Figure 5.9. We then annotate the classes according the their automatically computed 'featureAfffinity' values, but we change the value manually for the classes that have been classified wrong. The class FamixNameResolver receives the value 'single feature' instead of 'infrastructural'.

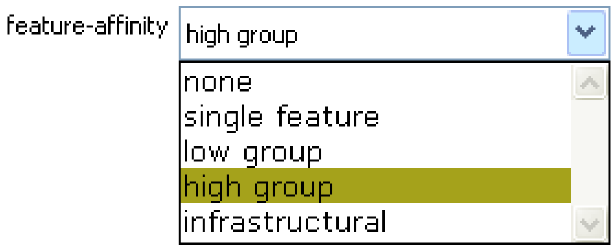


Figure 5.9: The feature-affinity annotation for classes

Refined visualization. In Figure 5.10 we show a system complexity view with the classes colored according to their feature-affinity annotation. The class MooseModel (1) is the only class that appears as 'infrastructural' in this picture. As this class is fundamental to every feature when using Moose, this result is more true to reality. The originally wrong categorized class FamixNameResolver (2) is now correctly painted as a 'single feature' class (blue).

With this example we have shown how an automatic feature analysis is enriched with developer knowledge using *Metanool* annotations and how this improves the reliability of the results.

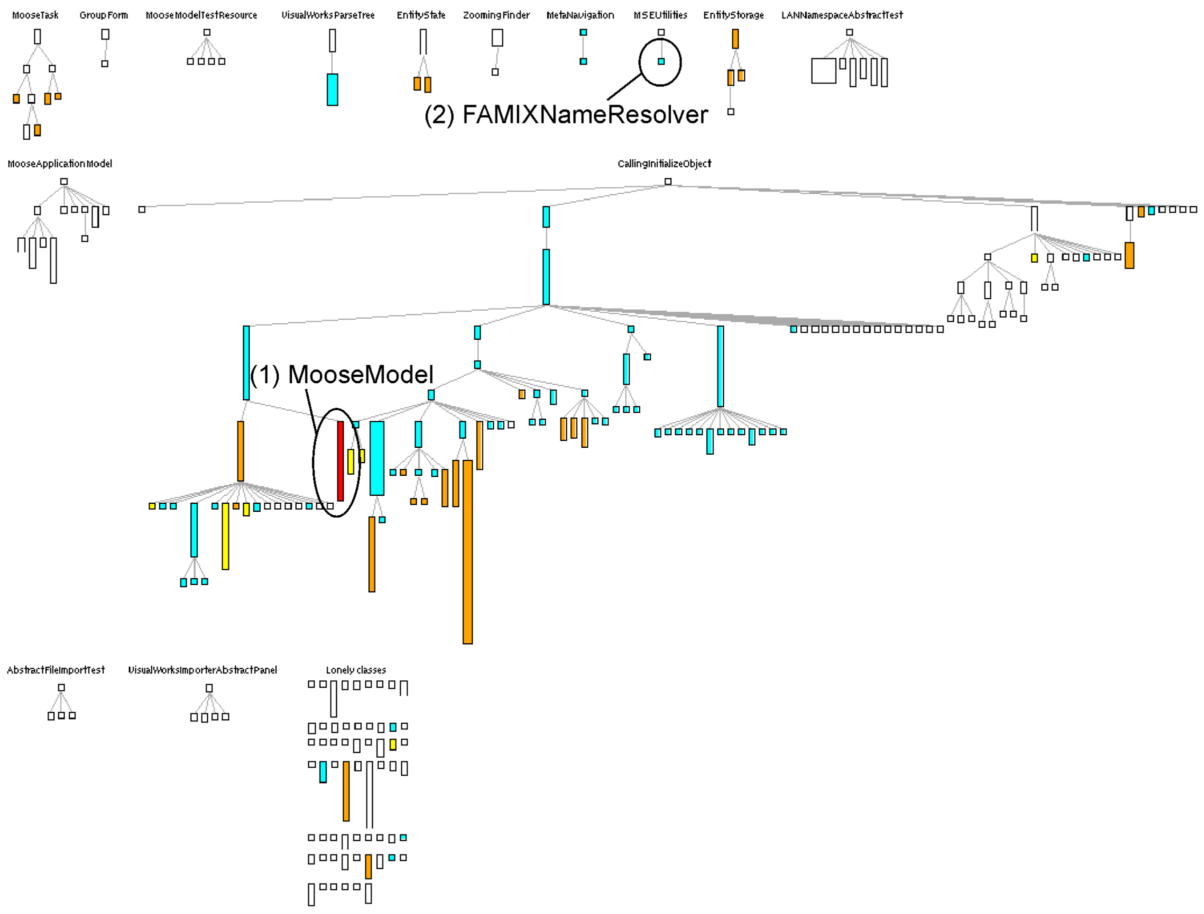


Figure 5.10: The refined feature affinity values of classes

6 Conclusions

In this chapter we conclude by summarizing the contributions of this dissertation and discussing possible future work that could be done in this area.

6.1 Contributions

The contributions of this thesis are:

A generic approach for integrating external knowledge into the reverse engineering process. We motivated the need of extending software analysis tools with a mechanism to include external knowledge. Our approach is to use annotations which are attached to objects and are meta-described in annotation descriptions. We emphasized that such a mechanism should provide possibilities to change the annotations and their descriptions during the whole analysis process. Furthermore we pointed out that the reverse engineer should be able to introduce new types.

An implementation of this approach. We developed an annotation framework called *Metanool* as an implementation of our approach. *Metanool* is implemented in Smalltalk and is integrated in the Moose reverse engineering environment. It provides an flexible and extensible annotation model. It also offers the possibility to watch and manipulate the annotations and their descriptions in a gui environment.

A set of examples of how our mechanism can be used to support existing approaches that take external knowledge into account. We first showed different ways of using annotations in architecture recovery. Then we demonstrated how task lists are realized with annotations. Annotations were then used to let reverse engineers refine automatic detection results. Finally we showed how annotations support feature analysis.

6.2 Future work

During the development of our approach, some ideas and points of interest have been found that could be the subject of future research in the area of annotations:

User defined type transformation. Changing the type of an annotation and automatically transforming of all existing values to conform to the new type is a central point of our approach. An extension would be if the reverse engineer could define a transformation strategy at the moment when he wants to transform the values, instead of being limited to the implemented transformation strategies. This would not only allow the user to change to a new type without implementing the strategy upfront but also to change the existing strategies when needed.

Java annotations. Java annotations could be enhanced with gui support to provide some of the benefits of *Metanool* also in Java source code. This would broaden the topic onto forward engineering as well as to a lightweight reverse engineering approach directly in the Java source code, making use of the development environment tools such as for example Eclipse [Murphy *et al.*, 2006].

Annotations for forward engineering in general. Annotations could be integrated in code browsers in various ways to also support forward engineering. For example, source artifacts could be tagged and categorized by annotations and then be browsed based on these categories. Browsers could also use colors to show some annotation based information directly in the navigation.

Scoping. In Metanool, annotation descriptions are system-wide unique. Further development could find out if more sophisticated scoping of annotations is needed. Also, the idea of value inheritance could be interesting. With value inheritance, a child object would return for a certain annotation description the annotation value of its parent unless it has an own value.

A Quick Start

This section describes how to download and run *Metanool*.

A.1 Installing Metanool

- Download VisualWorks 7.5. Non Commercial from `http://www.cincomsmalltalk.com/userblogs/cincom/blogView?content=smalltalk`
- Open visualnc.im with VisualWorks (drag it onto the VM, *e.g.,* Cincom/vw7.5nc/bin/win/vwnt.exe on a Windows machine)
- Connect to the repository (Store / Connect to Repository) with the following settings:
  - interface: PostgresSQLEXDIConnection
  - environment: db.iam.unibe.ch_scgStore
  - user name: storeguest
  - password: storeguest
  - table owner: BERN
- Load MooseSetup 1.6 (Store / Published items)
- Load System-Announcements 1.1.
- Load the latest version of Widgetry
- Load the latest version of Metanool

A.2 Getting started

Choose one of the following actions to create your first annotations:

- Execute `'Hello world' annotations openEditor`
- Open Moose (Tools / Open Moose Zooming Finder) and open a *Metanool* tab
- Execute some code from the introduction of this dissertation (Section 1.2 (p.3))

List of Figures

Bibliography

[Antoniol and Guéhéneuc, 2005] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM'05)*, pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.

[Arévalo *et al.*, 2003] Gabriela Arévalo, Stéphane Ducasse, and Oscar Nierstrasz. Understanding classes using X-Ray views. In *Proceedings of 2nd International Workshop on MASPEGHI 2003 (ASE 2003)*, pages 9–18. CRIM — University of Montreal (Canada), October 2003.

[Arévalo *et al.*, 2004] Gabriela Arévalo, Frank Buchli, and Oscar Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of WCRE '04 (11th Working Conference on Reverse Engineering)*, pages 122–131. IEEE Computer Society Press, November 2004.

[Breu and Krinke, 2004] Silvia Breu and Jens Krinke. Aspect mining using event traces. In *Proceedings of International Conference on Automated Software Engineering (ASE 2004)*, pages 310–315, 2004.

[Breu and Zimmermann, 2006] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[Christl *et al.*, 2005] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the reflexion method with automated clustering. In *Working Conference on Reverse Engineering (WCRE)*, pages 89–98, 2005.

[Cubranic and Murphy, 2003] Davor Cubranic and Gail Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[Deissenboeck and Ratiu, 2006] Florian Deissenboeck and Daniel Ratiu. A unified meta-model for concept-based reverse engineering. In *Proceedings of the 3rd*

International Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM'06), 2006.

[Dekel, 2003] Uri Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, February 2003.

[Demeyer *et al.*, 2002] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns.* Morgan Kaufmann, 2002.

[Ducasse and Lanza, 2005] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.

[Ducasse *et al.*, 2005] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[Ducasse *et al.*, 2008] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, and Lukas Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 2008. To appear.

[Eisenbarth *et al.*, 2003] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[Gîrba *et al.*, 2007] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Daniel Raţiu. Using concept analysis to detect co-change patterns. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2007)*, pages 83–89, 2007.

[Godin *et al.*, 1998] Robert Godin, Hafedh Mili, Guy W. Mineau, Rokia Missaoui, Amina Arfi, and Thuy-Tien Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983.

[Gosling *et al.*, 2005] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification (Third Edition).* Addison Wesley, 2005.

[Greevy and Ducasse, 2005a] Orla Greevy and Stéphane Ducasse. Characterizing the functional roles of classes and methods by analyzing feature traces. In *Proceedings of WOOR 2005 (6th International Workshop on Object-Oriented Reengineering)*, July 2005.

[Greevy and Ducasse, 2005b] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.

[Greevy, 2007] Orla Greevy. *Enriching Reverse Engineering with Feature Analysis.* PhD thesis, University of Berne, May 2007.

[Huchard *et al.*, 2000] Marianne Huchard, Hervé Dicky, and Hervé Leblanc. Galois Lattice as a Framework to specify Algorithms Building Class Hierarchies. *Theoretical Informatics and Applications*, 34:521–548, 2000.

[Koschke and Quante, 2005] Rainer Koschke and Jochen Quante. On dynamic feature location. *International Conference on Automated Software Engineering, 2005*, pages 86–95, 2005.

[Koschke and Simon, 2003] Rainer Koschke and Daniel Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, page 36. IEEE Computer Society, 2003.

[Kuhn *et al.*, 2007] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, March 2007.

[Lanza and Marinescu, 2006] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice.* Springer-Verlag, 2006.

[Lienhard *et al.*, 2005] Adrian Lienhard, Stéphane Ducasse, and Gabriela Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, November 2005.

[Maletic and Marcus, 2000] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53, November 2000.

[Marcus and Maletic, 2001] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, November 2001.

[Marcus and Maletic, 2003] Andrian Marcus and Jonathan Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing.

In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 125–135, May 2003.

[Marcus and Poshyvanyk, 2005] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[Marcus *et al.*, 2004] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, November 2004.

[Marin *et al.*, 2007] Marius Marin, Arie van Deursen, and Leon Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Software Engineering and Methodology*, 17(1):1–37, 2007.

[Marinescu *et al.*, 2005] Cristina Marinescu, Radu Marinescu, Petru Mihancea, Daniel Ratiu, and Richard Wettel. iPlasma:an integrated platform for quality assessment of object-oriented design. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 77–80, 2005. Tool demo.

[Mens *et al.*, 2002] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of SEKE 2002*, pages 289–296. ACM Press, 2002.

[Mens *et al.*, 2006] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006.

[Meyer *et al.*, 2006] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[Murphy *et al.*, 1995] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[Murphy *et al.*, 2006] Murphy, Kersten, and Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, jul 2006.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the*

European Software Engineering Conference (ESEC/FSE'05), pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[Patel *et al.*, 2003] Sandipkumar Patel, Yogesh Dandawate, and John Kuriakose. Architecture recovery as first step in system appreciation, 2003. http://softeng.polito.it/events/WESRE2006/03Dandawate.pdf.

[Pinzger *et al.*, 2002] Martin Pinzger, Michael Fischer, Harald Gall, and Mehdi Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 170–178, 2002.

[Raţiu and Deissenboeck, 2006] Daniel Raţiu and Florian Deissenboeck. How programs represent reality (and how they don't). In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, Los Alamitos CA, 2006. IEEE Computer Society.

[Raţiu and Deissenboeck, 2007] Daniel Raţiu and Florian Deissenboeck. From reality to programs and (not quite) back again. In *Proceedings of the 15th International Conference on Program Comprehension, (ICPC 2007)*, pages 91–102, Los Alamitos CA, 2007. IEEE Computer Society.

[Raţiu and Juerjens, 2007] Daniel Raţiu and Jan Juerjens. The reality of libraries. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering, (CSMR 2007)*, pages 307–318. IEEE Computer Society, 2007.

[Snelting and Tip, 1998] Gregor Snelting and Frank Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[Stroustrup and Ellis, 1990] Bjarne Stroustrup and Magaret A. Ellis. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Tonella and Antoniol, 1999] Paolo Tonella and Giuliano Antoniol. Object oriented design pattern inference. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 230–238. IEEE Computer Society Press, October 1999.

[Wilde and Scully, 1995] Norman Wilde and Michael Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[Yoder and Johnson, 2002] Joseph W. Yoder and Ralph Johnson. The adaptive object model architectural style. In *Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02)*, August 2002.

[Yoder *et al.*, 2001] Joseph Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object models. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, pages 50–60, 2001.