Enriching Reverse Engineering with Annotations*

Andrea Brühlmann, Tudor Gîrba, Orla Greevy, Oscar Nierstrasz

Software Composition Group, University of Bern, Switzerland
http://scg.unibe.ch/

Abstract. Much of the knowledge about software systems is implicit, and therefore difficult to recover by purely automated techniques. Architectural layers and the externally visible features of software systems are two examples of information that can be difficult to detect from source code alone, and that would benefit from additional human knowledge. Typical approaches to reasoning about data involve encoding an explicit meta-model and expressing analyses at that level. Due to its informal nature, however, human knowledge can be difficult to characterize up-front and integrate into such a meta-model. We propose a generic, annotation-based approach to capture such knowledge during the reverse engineering process. Annotation types can be iteratively defined, refined and transformed, without requiring a fixed meta-model to be defined in advance. We show how our approach supports reverse engineering by implementing it in a tool called Metanool and by applying it to (i) analyzing architectural layering, (ii) tracking reengineering tasks, (iii) detecting design flaws, and (iv) analyzing features.

1 Introduction

Most reverse engineering techniques focus on automatically extracting information from the source code without taking external human knowledge into consideration. More often than not however, important external information is available (*e.g.*, developer knowledge or domain specific knowledge) which would greatly enhance analyses if it could be taken into account.

Only few reverse engineering approaches integrate such external human knowledge into the analysis. For example, reflexion models have been proposed for architecture recovery by capturing developer knowledge and then manually mapping this knowledge to the source code [1,2]. Another example is provided by Intensional Views which make use of rules that encode external constraints and are checked against the actual source code [3].

In this paper we propose a generic framework based on annotations to enhance a reverse engineered model with external knowledge so that automatic analyses can take this knowledge into account. A key feature of our approach

* Models 2008, Krzysztof Czarnecki, et al. (Eds.), LNCS, vol. 5301, Springer-Verlag, 2008, pp. 660-674.

is that the types of supported annotations need not be fixed in advance. A developer can introduce new types of annotations at any time, and refine or transform the existing types in a flexible way. For example, using the reflexion model approach to analyze how a system is split into architectural layers, the reverse engineer can define the notion of a layer, instantiate this concept for the different layers that can be found in the code and then attach the specific layers to code entities.

Annotations are intended to augment reverse engineering analyses by providing additional information otherwise absent from the source code. This information is then available to refine the results of queries, visualizations, and other automated tasks.

We have built a prototype, called Metanool, that demonstrates our approach, and we have integrated it into the Moose reengineering environment [4]. Our approach is inspired by Adaptive Object-Models and it is applied in the context of reverse engineering [5]. We argue that the approach is quite general, and could be easily integrated into any reverse engineering tool or development environment based on explicit meta-models of software. We believe that such a mechanism to facilitate the incorporation of external knowledge should be an integral part of every reverse engineering tool suite.

Metanool's annotations are similar to Java annotations [6] in the sense that they are meta-described, hence can be examined by the run-time system, but the two differ in several important ways: (1) Metanool annotations can be changed at runtime, (2) they can be manipulated from the user interface, and (3) they can be attached to any object.

We have applied Metanool to a number of different experimental use cases to assess the usefulness of our approach to reverse engineering. In one experiment we used Metanool to encode manually detected architectural layers of jEdit, and then used this information to check for violations of architectural constraints. In another experiment, annotations were used to directly encode the architectural constraints themselves. In a very different use case, we used annotations to express and manage pending reverse engineering tasks. Two further use cases entailed augmenting automated design flaw detection with annotations to manually flag false negatives and false positives, and augmenting feature analysis with developer knowledge of available features. Although these experiments are only anecdotal in nature, they serve well to illustrate the usefulness of the approach.

In the next section we illustrate our approach as it is manifested by the Metanool tool. In Section 3 we outline the implementation and the meta-model of Metanool. Section 4 presents four example use cases for our approach, thus illustrating how Metanool annotations can effectively enrich the reverse engineering process. In Section 5 we compare our approach with related work, and in Section 6 we conclude with remarks on future work.

2 Metanool by example

To provide a flavor of the reverse engineering use cases we solve, in this section we illustrate our approach by showing how to use our Metanool infrastructure to annotate a reverse engineered model of a software system. In this example, Metanool is used together with the Moose reengineering environment [4]. A model describing the static structure of jEdit [7] has been loaded into Moose, and Metanool is then used to enrich this model with external knowledge expressed by means of annotations. Here we imagine a developer identifying the architectural layers and determining which packages belong to which layer. Finally this enriched model is used to generate a report.

2.1 Adding an annotation

Figure 1 shows the Moose Browser presenting a model of jEdit, which has been loaded by parsing the jEdit source code. The figure shows that we have selected the org.gjt.sp.jedit.gui package (or namespace) (1).

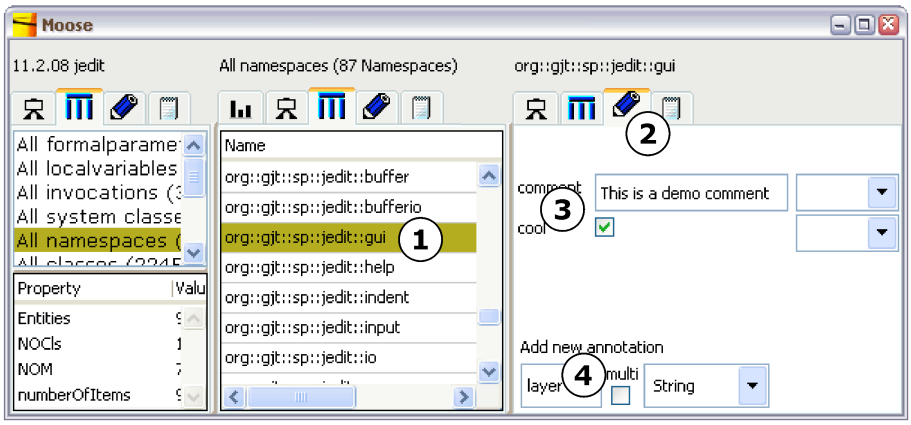

Fig. 1. A Moose model of jEdit.

On the right hand side, we have the Metanool annotations tab open for this package (2). It contains the list of annotations (3). There are already two annotations called comment and cool.

To create a new annotation, we enter the name, for example layer and assign a type to it, *e.g.*, String at the bottom of the annotations tab (4). As a result, our newly created annotation appears in the annotations list, where we can edit its value (see Figure 2). The annotation is added to all elements that have the same type as the selected element (by default, the value of the annotation is nil). As a consequence, in our example, we can now edit the layer names for all namespaces.

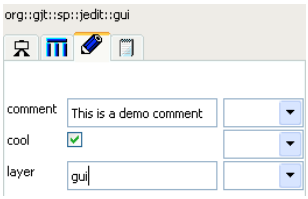

Fig. 2. A new annotation.

2.2 Refining an annotation type

A key feature of our approach is that annotation types can be changed easily at any time, even when values have already been assigned. After annotating some of the jEdit packages, it occurs to us that usability would be improved by adding a drop down menu with the list of known layers.

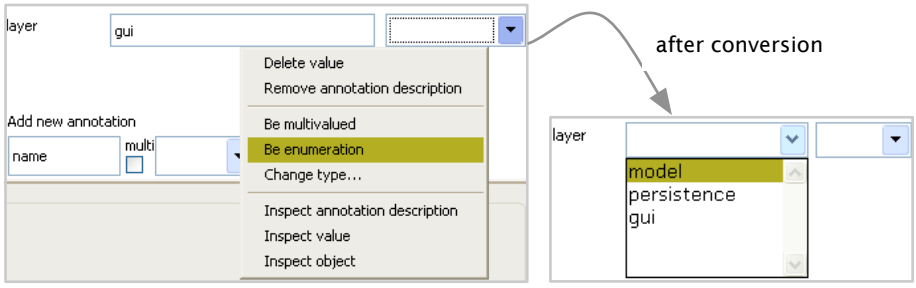

Fig. 3. Converting an annotation type to an enumeration.

In this case, we choose "Be enumeration" from the menu. This searches for all layer names that are already specified and transforms the type of the annotation into an enumeration of these names.

2.3 Creating a report

Finally, we can use the information from the annotated model to generate reports. For example, in Figure 4 we show a possible visualization that would be included in such a report. The visualization, generated using Mondrian [8], a scriptable visualization engine, shows the namespace (small squares) organized in layers. The Mondrian code to the left illustrates (in bold) how we take the 'layer' annotation into account to group the namespaces.

3 Metanool meta-model and implementation

In this section we present the meta-model that Metanool uses to represent annotations, and we outline how Metanool uses this meta-model to manage and manipulate annotations.

```
namespacesByLayers := namespaces groupedBy: [:each |
   each annotations named: 'layer'.
].

view := ViewRenderer new.

view umlPackageShapeWithLabel.
view nodes:namespacesByLayers forEach: [:eachAssociation |
   view nodes: eachAssociation.
   view horizontalLineLayout.
].
view verticalLineLayout.

view open
```

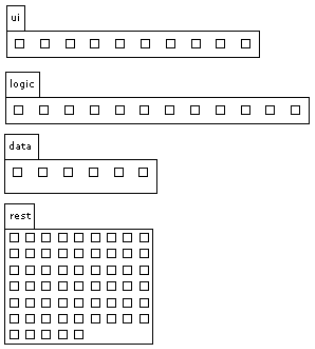

Fig. 4. Visualizing the layers with a Mondrian script.

3.1 Metanool meta-model

Figure 5 illustrates the annotation meta-model by means of a simple example. The example shows an small excerpt of the objects (e.g., :FAMIX.Namespace), classes (e.g., FAMIX.Namespace) and meta-descriptions (e.g., EMOF.Class) that were involved in the scenario from Section 2. In Moose, each object is attached to an EMOF meta-description [9]. We extended this model by allowing each object to have annotations, which in turn are extensions of EMOF.Property. The available annotations are encoded in AnnotationDescription objects that are attached to the meta-descriptions.

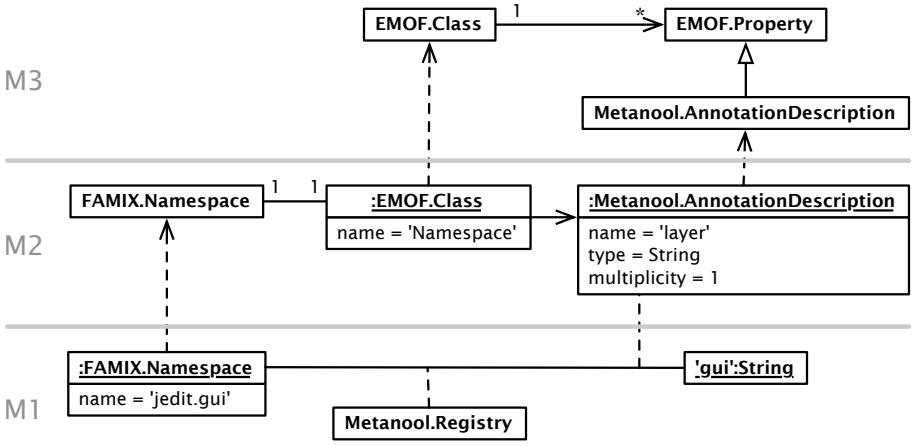

Fig. 5. Metanool core structure.

In this example, we see that the jedit.gui namespace is an instance of the FAMIX.Namespace class. This class is meta-described by an instance of EMOF.Class named 'Namespace'. It has one AnnotationDescription, which specifies that annotations of FAMIX.Namespace objects may have single-valued 'layer' annotations of type String.

Annotations are represented as associated properties. In this case the 'jedit.gui' namespace has the layer annotation 'gui'. Annotation descriptions are registered in a global registry named Metanool.Registry.

3.2 Representing and manipulating annotations

As we have seen, each annotation has a name, a type and a multiplicity. In our implementation, the allowable types include not only all classes of the host programming language (in this case Smalltalk), but we also allow enumerations of values of a given class. In order for the annotation editor to allow users to conveniently set the values for annotations, the corresponding Smalltalk class should be extended with a dedicated GUI method. In lieu of this, values can also be set programmatically for arbitrary types.

As exemplified in Section 2.2, Metanool allows the user to modify the annotation descriptions in different ways. First of all, the type can be changed. This is, of course, not problematic when no values have been stored for this annotation. But it is useful to have this flexibility later during analysis if the structure of a human knowledge concept emerges over time. When the type needs to be changed, Metanool takes all existing values and tries to transform them to fit the newly defined type. For example, when changing a simple type to an enumeration, Metanool will create an enumeration from all existing values.

An annotation can also be transformed into an enumeration: in this case, all existing values of this annotation are collected into a set which is then defined to be the type of the annotation. The reverse operation is also possible: the new type is determined by taking the most specific common superclass of all the enumeration elements.

We also allow the multiplicity of an annotation to be modified after it has been defined. An annotation can easily be made multi-valued at any time, and almost as easily be made single-valued again. Going from multi-valued to single-valued will result in loss of information if multiple values need to be dropped. Metanool prompts the user which values to keep.

3.3 Programmatically manipulating annotations

Annotations can also be created and edited programmatically from the host language. The following lines show the code to create a layer annotation and edit it in Smalltalk. It also exemplifies the relations between annotation descriptions and their values:

```
layerDescription := AnnotationDescription name: 'layer' type: String.
FAMIX.Namespace annotationDescriptions add: layerDescription.

namespaceA annotations named: 'layer' put: 'ui'.
namespaceB annotations named: 'layer' put: 'data'.
```

Values can be read from annotations as follows:

```
namespaceA annotations named: 'layer'.
```

Further details of Metanool can be found in Brühlmann's Masters thesis [10].

4 Reverse engineering with annotations

We now illustrate the usefulness of annotations for reverse engineering by presenting four typical use cases. First, in subsection 4.1 we show how annotations can be used to check architectural constraints by encoding either the layers or the constraints themselves. Next, in subsection 4.2 we show how a specialized annotation type with its own GUI can easily be added to represent pending tasks. In subsection 4.3 we show how false negatives and false positives can be tagged as annotations to refine the results of automated design flaw detection. Finally, in subsection 4.4 we show how implicit knowledge about internal features can be used to enrich and refine the results of feature analysis.

4.1 Checking architectural constraints

Software companies often define a standard architecture for a set of applications. As new applications are developed or existing applications evolve, they should be checked for compliance with architectural constraints. Unfortunately architectures are typically not explicit in the source code. Packages are generally 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 even be associated with multiple layers. In the absence of a dedicated language construct to express architectural layering, annotations provide an ideal way during analysis of a system to associate packages with layers. An analysis of the call graph can then make use of the layering annotations to check the architectural constraints [2].

We present two ways to encode layering knowledge with annotations. In the first case we number layers and annotate packages according to their layer number. In the second case we name the layers, and directly encode the desired accessibility constraints as annotations. In both cases we then visualize the packages and their access relations, coloring with red forbidden calls and with blue allowed calls.

Numbered layers. In this use case, we took the work of Patel *et al.* who previously analyzed jEdit and classified its packages according to 14 layers [11].

We defined a single-valued annotation description named layerNumber of type Integer and then encoded knowledge about the 14 layers by annotating each package with the layer numbers. Figure 6 shows a Mondrian visualization of the 14 layers depicting allowed calls (*i.e.* going downwards) with blue, call violation (*i.e.* going upwards) in red, and not qualified calls in gray. The code of the visualization is similar to the one from Figure 4.

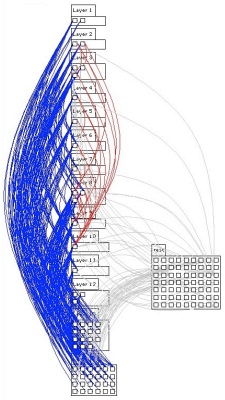

Fig. 6. jEdit in 14 layers with access violations in red.

Named layers with explicit constraints. Another approach is to add a String annotation to each package, containing the name of its layer, *e.g.*, gui, model, persistence etc. Relations between these layers specify whether calls of components from one layer to another are allowed or forbidden. These relations are also expressed using annotations. We create two annotation descriptions and attach them to String (see Figure 7). They are multi-valued because each layer has multiple forbidden or allowed layers to access. The reason why we added it to String instead of a dedicated Layer class, is just convenience, as strings can easily be instantiated using 'quotes'.

We can again display the call relationships (see Figure 7) with a dedicated visualization that takes annotations into account. Blue calls are explicitly allowed and red calls are forbidden. Grey calls are not specified with our annotations, so they are neither allowed nor forbidden.

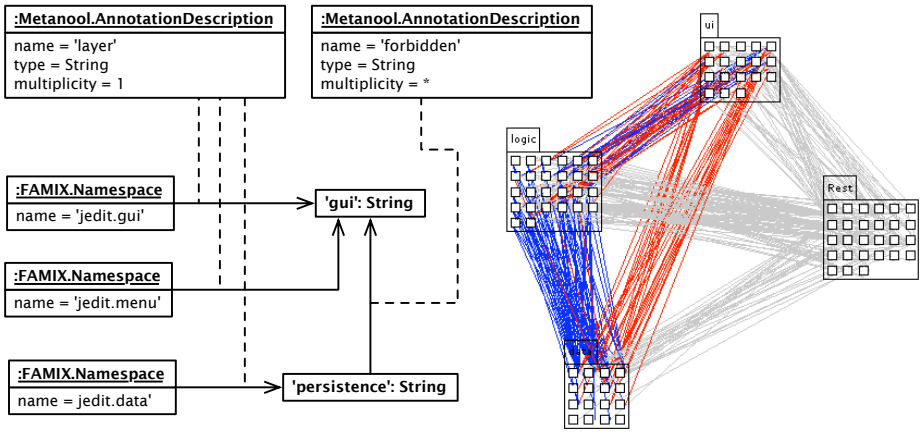

Fig. 7. Explicit constraints over named layers. The 'allowed' annotations are similar to the 'forbidden' one, and were omitted from the diagram due to space limitation.

4.2 Task lists

In this example, tasks are added to objects and a list of all tasks is generated. This is analogous to the @ToDo annotation in Java.

Introducing a task type. To attach to-do tasks to elements such as classes and namespaces, we must introduce a new type: ToDo. We define the new class ToDo with two instance variables: a String named 'task' and a Boolean named 'done'. Like this, each ToDo has a description and can be marked as done. This new type can already be used in Metanool as is, but it is better to offer a specialized GUI to make it easy to enter and update tasks. The metanoolEditPane method creates the GUI editor pane for tasks consisting of a checkbox (by adding the default edit pane for Boolean) and a text input field (by adding the default edit pane for String):

```
ToDo≫metanoolEditPane
    ↑ Form new
        addComponent: (Boolean metanoolEditPaneForProperty: 'done' of: self);
        addComponent: (String metanoolEditPaneForProperty:  'task' of: self).
```

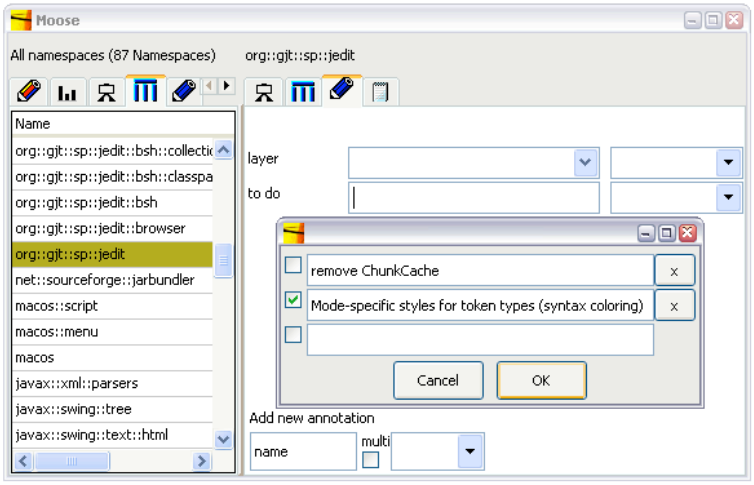

Fig. 8. A specialized GUI for multi-valued to-do annotations.

Editing the tasks. After creating the ToDo type, we add a multi-valued "to do" annotation description of type ToDo to the elements we want to annotate. Like this, every element can have as many tasks as needed (see Figure 8). We can also add the same annotation description to several elements, *e.g.*, FAMIX.Class and FAMIX.Namespace.

4.3 Detection strategies

Detection strategies are metrics-based rules to detect candidates for common design flaws in a system, such as god classes, data classes, brain classes and other code smells [12]. Since detection strategies only detect *candidates*, manual inspection is often needed to filter out false positives. Annotations can be used to flag these false positives, and also to flag false negatives that may have been detected by other means.

Let us consider the example of detecting god classes in jEdit. First we add annotation descriptions "auto god class" and "manual god class", both of type Boolean. We set the auto god class annotation to true for all classes identified by our detection strategy. Then we manually set the annotation for all god classes we have knowledge of.

In Figure 9 we see a visualization of the jEdit class hierarchy in which all god classes are represented by large rectangles. Classes are colored black only if the manual and automatic annotations agree. False positives are shown as small pink boxes (manually refused). False negatives are shown as large red boxes (manually detected but not automatically). Grey boxes are automatically detected god classes that have not been manually rejected.

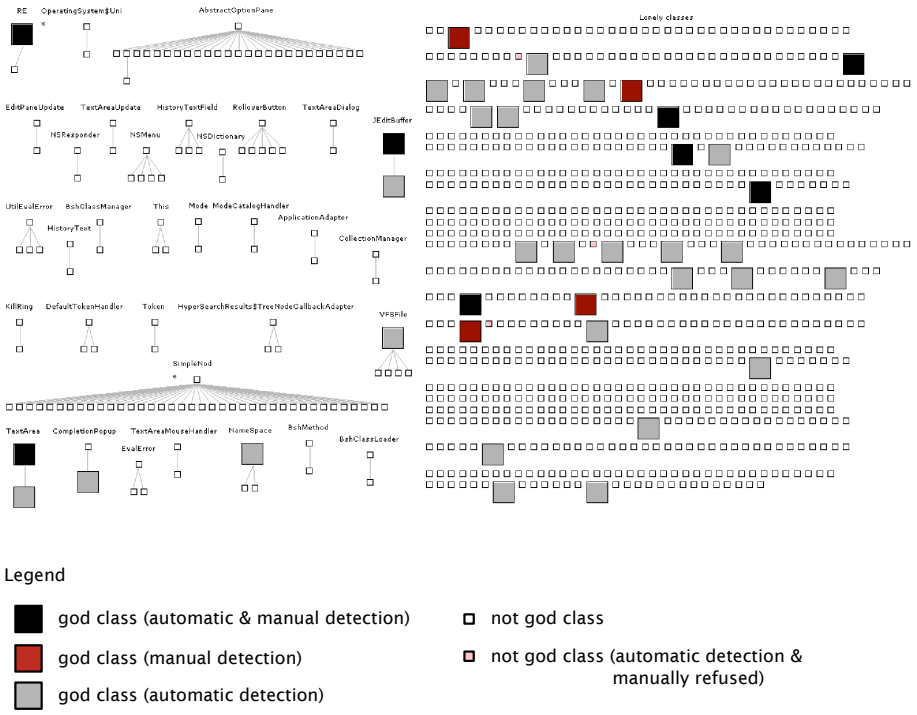

Fig. 9. Automatic and manual detection of god classes. The reason for the existence of 5 categories out of two boolean variables is that the default value is nil.

4.4 Feature analysis

The final example shows how to use our annotations to perform and validate the results of automatic feature analysis based on dynamic analysis of traces [13].

Features are abstractions that encapsulate knowledge of a problem domain and describe units of system behavior [14]. Several researchers have identified the potential of exploiting features in reverse engineering [15,16]. Feature identification approaches (*e.g.*, *Software Reconnaissance* [17]) 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.

We show how Metanool annotations can help to refine the results of feature analysis by taking additional developer knowledge into account. In particular, the presence of hidden features can perturb the analysis unless they are taken into account.

Feature analysis of Moose. We performed a feature analysis on the Moose system itself with 8 specified features using a Dynamix model of Moose [14]. Then we applied the feature affinity metric to the classes based on these results to quantify which classes belong to only one feature (single feature affinity), and which classes are used for many features (ranging from low group affinity, to high group affinity and infrastructural).

Our feature analysis automatically detects three classes as being *infrastructural*. The developers of Moose deny this result. They state that one of the classes detected as infrastructural, the FamixNameResolver class, implements functionality specific to importing a Moose model from Smalltalk source code. This functionality should in fact constitute a distinct feature. Another of the three detected infrastructural classes is named UnresolvedClass in our model. The events captured for a feature may reference classes that are not present in static model of the system and thus not every participating class of a feature can be resolved to a class in the model. As each feature exhibited unresolvable classes, automatic feature analysis incorrectly identified this UnresolvedClass as an infrastructural class.

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 these false positives in *feature affinity* 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 the enumeration type #('none' 'single feature' 'low group' 'high group' 'infrastructural'). We then anno-

tate the classes according the their automatically computed *featureAffinity* values, but we change the value manually for the classes that have been classified wrongly. The class FamixNameResolver takes the annotation value 'single feature' instead of 'infrastructural'.

Refined visualization. In Figure 10 we show part of the system complexity view with the classes coloured according to their feature–affinity annotation. The class MooseModel (1) is the only class that appears now as 'infrastructural ' in this picture. As this class is fundamental to every feature when using Moose, this result is closer to reality. The originally wrongly categorized class FamixNameResolver is now correctly coloured as a 'single feature' class (blue).

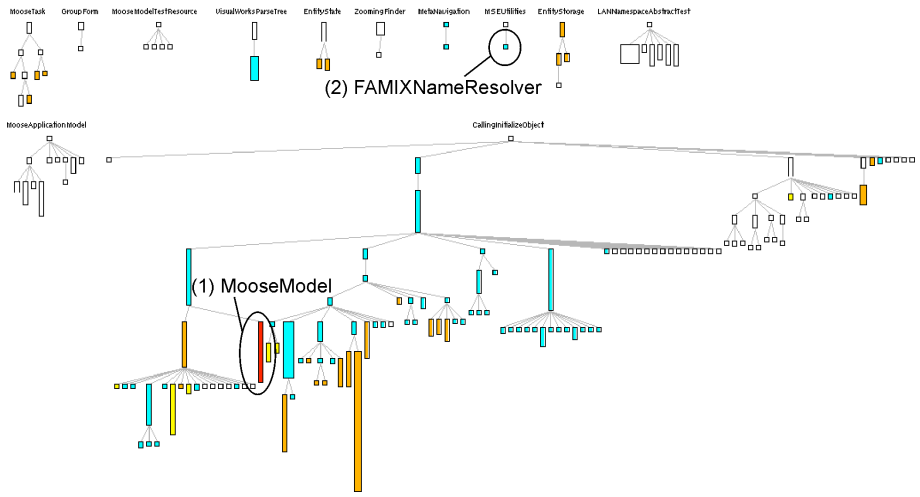

Fig. 10. The refined feature affinity values of classes

5 Related work

Every programming language offers the possibility of embedding textual comments in the source code to encode external knowledge directly in the system [18,19]. Comments serve as a form of documentation and should be accessible to code analysis tools since they provide an important source of additional information. Comments, however, are unstructured and cannot easily be taken into account for further analysis. One notable exception is the use of clustering techniques to identify concepts implicit in the textual comments of source code [20].

A more advanced approach of annotating source code is offered by Java Annotations[6]. They are meta-described by instances of AnnotationTypes thus enabling run-time examination. However, as Java Annotations are source code based, they cannot be used to 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 specialized editors for annotating and for editing the meta-descriptions.

Adaptive Object-Models (AOM) [5,21] encode domain entities in metadata instead of classes. Thus, whenever a change is needed, the meta-model 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 solves 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.

Annotations can also be fruitfully integrated into the runtime of a programming language. Reflectivity is a framework that extends Smalltalk with support for reflection at the sub-method level [22]. Representations of source code can be annotated at run-time for use by various tools. This has been used to perform feature analysis [23]. Instead of generating traces to be analyzed *post hoc*, objects representing the static structure of the system are annotated with feature information while features are exercised. However, as opposed to Metanool, the annotations proposed in Reflectivity are not typed thus they are less useful for modeling complex concepts.

6 Conclusions

We have presented an approach to enriching the results of reverse engineering analyses with human knowledge by means of iteratively and incrementally defined, typed annotations. Annotations are meta-described, and can therefore be analyzed and manipulated at run-time by the same reverse engineering tools that carry out analysis of unadorned source code models.

We have demonstrated our approach by presenting Metanool, a tool for defining, editing and manipulating annotations. We have also shown four concrete scenarios of our approach using Metanool in the context of the Moose reengineering environment. We have shown how annotations can be fruitfully exploited to enrich reverse engineering with information that is implicit in, or even absent from the source code, such as adherence to architectural constraints, the presence of design flaws, or the relationship between software components and system features. We have also shown how annotations can flexibly added over time, and transformed as the needed structure of the information emerges. Graphical editors can be easily added to support new types of annotations.

During the development of Metanool, several directions for further research became apparent:

– *User defined type transformation.* Changing the type of an annotation and automatically transforming all existing values to conform to the new type is

a central point of our approach. A useful extension would be to allow the reverse engineer to define a transformation strategy. This would be similar to the solution featured in DabbleDB[1], and it would not only allow the user to change existing annotations to a new type without implementing the strategy up front, but also to augment the existing strategies when needed.
- *Scoping.* In Metanool, annotation descriptions are globally defined. More sophisticated scoping of annotations could be useful, especially for analysis of large systems where certain annotations types might only make sense for particular subsystems. Also, the idea of value inheritance could be interesting. With value inheritance, a child object would return the annotation value of its parent unless it has its own value.
- *Java annotations.* Java annotations could be enhanced with GUI support to provide some of the benefits of Metanool in Java source code. Lightweight reverse engineering tools could then be better supported in environments such as Eclipse [24].
- *Annotations for forward engineering.* Annotations could be integrated into code browsers in various ways to also support forward engineering. For example, source code artifacts could be tagged and categorized by annotations and then be browsed based on these categories.

Acknowledgments. We gratefully acknowledge the financial support of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" (project no. 2234) and the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008). We would also want to thank Alexandre Bergel and Lukas Renggli for their comments on drafts on this paper.

References

1. Koschke, R., Simon, D.: Hierarchical reflexion models. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003), IEEE Computer Society (2003) 36
2. Murphy, G., Notkin, D., Sullivan, K.: 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, ACM Press (1995) 18–28
3. Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: Co-evolving code and design with intensional views — a case study. Journal of Computer Languages, Systems and Structures **32** (2006) 140–156
4. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), New York NY, ACM Press (2005) 1–10 Invited paper.
5. Yoder, J., Balaguer, F., Johnson, R.: Architecture and design of adaptive object models. In: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01). (2001) 50–60

[1] http://www.dabbledb.com

6. Sun microsystems: Java annotations (2004) http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.
7. jEdit web site: jEdit: a programmer's text editor (2008) http://www.jedit.org.
8. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: An agile visualization framework. In: ACM Symposium on Software Visualization (SoftVis'06), New York, NY, USA, ACM Press (2006) 135–144
9. Ducasse, S., Gîrba, T.: Using Smalltalk as a reflective executable meta-language. In: International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006). Volume 4199 of LNCS., Berlin, Germany, Springer-Verlag (2006) 604–618
10. Brühlmann, A.: Enriching reverse engineering with annotations. Master's thesis, University of Bern (2008)
11. Patel, S., Dandawate, Y., Kuriakose, J.: Architecture recovery as first step in system appreciation. In: 2nd Workshop on Empirical Studies in Reverse Engineering, Politecnico di Torino (2006) http://softeng.polito.it/events/WESRE2006/.
12. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice. Springer-Verlag (2006)
13. Greevy, O., Ducasse, S.: 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), Los Alamitos CA, IEEE Computer Society (2005) 314–323
14. Greevy, O.: Enriching Reverse Engineering with Feature Analysis. PhD thesis, University of Berne (2007)
15. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Computer **29** (2003) 210–224
16. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: a novel approach and a case study. In: Proceedings IEEE International Conference on Software Maintenance (ICSM'05), Los Alamitos CA, IEEE Computer Society Press (2005) 357–366
17. Wilde, N., Scully, M.: Software reconnaisance: Mapping program features to code. Software Maintenance: Research and Practice **7** (1995) 49–62
18. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification (Third Edition). Addison Wesley (2005)
19. Stroustrup, B., Ellis, M.A.: The Annotated C++ Reference Manual. Addison Wesley (1990)
20. Kuhn, A., Ducasse, S., Gîrba, T.: Enriching reverse engineering with semantic clustering. In: Proceedings of 12th Working Conference on Reverse Engineering (WCRE'05), Los Alamitos CA, IEEE Computer Society Press (2005) 113–122
21. Yoder, J.W., Johnson, R.: The adaptive object model architectural style. In: Proceeding of The Working IEEE/IFIP Conference on Software Architecture 2002 (WICSA3 '02). (2002)
22. Denker, M., Ducasse, S., Lienhard, A., Marschall, P.: Sub-method reflection. Journal of Object Technology **6** (2007) 231–251
23. Denker, M., Greevy, O., Nierstrasz, O.: Supporting feature analysis with runtime annotations. In: Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007), Technische Universiteit Delft (2007) 29–33
24. Murphy, Kersten, Findlater: How are Java software developers using the Eclipse IDE? IEEE Software (2006)