Interpreting Story Diagrams for the Static Detection of Software Patterns

Markus Fockel, Dietrich Travkin, Markus von Detten
Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Paderborn, Germany
[mfockel|travkin|mvdetten]@mail.uni-paderborn.de

ABSTRACT

Software maintenance tasks require knowledge about the software's design. Several tools help to identify implementations of software patterns, e.g. Design Patterns, in source code and thus help to reveal the underlying design. In case of the reverse engineering tool suite Reclipse [15], detection algorithms are generated from manually created, formal pattern specifications. Due to numerous variants that have to be considered, the pattern specification is error-prone. Because of this, the complex, step-wise generation process has to be traceable backwards to identify specification mistakes. To increase the traceability, we directly interpret the detection algorithm models (story diagrams) instead of executing code generated from these models. This way, a reverse engineer no longer has to relate generated code to the story diagrams to find mistakes in pattern specifications.

1. INTRODUCTION

Due to requests for new features and the discovery of defects, software has to be continuously adapted and maintained. For this purpose, developers have to understand the design of a given software. Software design patterns [5] are approved, widely used solutions for design problems. Knowledge about their usage in the software helps to understand how the original developers intended the software to be extended or adapted and, thus, helps to avoid design deviations or errors.

Incomplete documentation often complicates the task of identifying pattern implementations in source code. Several tools have been developed to automate this tedious task (Dong et al. give an overview [2]). Based on a formal specification of a pattern, usually represented by a set of conditions, these tools automatically detect pattern implementations in source code.

Nevertheless, due to numerous implementation variants[1] to be considered during pattern specification, the task of specifying a pattern is error-prone which sometimes results in missing pattern implementations (false negatives) or finding more than are actually present (false positives). To correct a specification, a reverse engineer has to identify the erroneous or missing conditions in the specification that lead to the unexpected detection results which, in turn, requires traceability of the detection process.

[1]For example, different loop implementations or the distinction between interfaces and classes in Java.

In case of the reverse engineering tool suite Reclipse[2] [15], the detection process is quite complex. Reclipse automatically derives detection algorithms from pattern specifications, creates models of these algorithms in form of class and story diagrams [3], generates code out of these models, and executes this code to detect pattern implementations in given source code [10]. To trace the detection process, a reverse engineer has to observe the generated detection code's behavior, deduce the elements which are representing this behavior in the generated story diagrams, and identify the corresponding conditions in the pattern specifications. Hence, the reverse engineer has to bridge two semantic gaps: the one between code and story diagrams and the one between story diagrams and pattern specifications.

A pattern specification only describing a class declaration and a contained method declaration already results in about 1000 lines of generated code. As example take the following excerpt of code that would be generated based on such a pattern specification. The lines 1 to 3 contain declarations of two variables `clazz` and `method` to represent the declarations and an auxiliary variable for the iteration through all elements contained in a class. In lines 4 and 5 a part of the code to be analyzed is assumed to be the specified class declaration. The remaining lines describe the search for a method declaration contained in the class represented by the previously found class declaration. As a class can contain several method declarations, this is done in a loop.

```
  ...
1 ATypeDeclaration clazz = null;
2 Iterator fujaba__IterClazzToMethod = null;
3 AMethodDeclaration method = null;
  ...
4 JavaSDM.ensure(_TmpObject instanceof
                              ATypeDeclaration);
5 clazz = (ATypeDeclaration) _TmpObject;
  ...
6 fujaba__IterClazzToMethod = clazz
                    .iteratorOfBodyDeclarations();
7 while (fujaba__IterClazzToMethod.hasNext()) {
8   _TmpObject = fujaba__IterClazzToMethod.next();
9   JavaSDM.ensure( _TmpObject instanceof
                             AMethodDeclaration);
10  method = (AMethodDeclaration) _TmpObject;
    ...
  }
  ...
```

[2]http://www.fujaba.de/reclipse

The reverse engineer has to mentally bridge the semantic gap between this code and the corresponding story diagram and eventually the pattern specification. She has to find the conditions in the pattern specification that are represented by the variables in the generated code and then identify the error in the specification by debugging the code execution.

We're aiming to avoid the semantic gaps by directly interpreting the pattern specifications, thereby adding tracing functionality to Reclipse's pattern detection, similar to debuggers. As a first step, we remove the semantic gap between generated code and story diagrams by directly interpreting the story diagrams instead of generating code. As there is an existing interpreter for story diagrams [7] with a corresponding debugger being currently developed [8], this is a promising solution. Furthermore, by exploiting runtime information, interpreting story diagrams can be more efficient than executing code [7].

In this paper we present the actions we have taken to integrate the story diagram interpreter developed at the Hasso Plattner Institute in Potsdam [7] into Reclipse, the challenges we faced and an evaluation of the results.

2. THE PATTERN DETECTION PROCESS

Reclipse's current pattern detection process is depicted in Figure 1. First of all, the design patterns have to be defined manually as formal pattern specifications. Algorithms (in form of story diagrams) that describe the search for the specified patterns are automatically derived from the formal pattern specifications. These detection algorithm models are then used to generate code that is later called by the *inference algorithm*. The code in which to search for implementations of the specified patterns has to be transformed into an *abstract syntax graph* (ASG), which is done by Reclipse automatically. The inference algorithm receives the ASG and the generated detection algorithm code as input. It decides where in the ASG to search for a pattern and executes the respective detection algorithm code to do so. Finally, the output is an ASG in which the detected pattern implementations are marked by annotations. As Reclipse is based on the CASE tool Fujaba[3] [9], all models have been created or generated with that framework (signified in Figure 1 by the ellipses with the Fujaba inscription).

3. INTERPRETER INTEGRATION

In order to remove the semantic gap between story diagrams and generated detection code, we integrated the story diagram interpreter into the pattern detection process of Reclipse as illustrated in Figure 2. During that integration we faced several challenges.

First, the interpreter is based on a story diagram meta-model that is slightly different from the one used in Reclipse (provided by Fujaba). Hence, we had to translate the story diagrams from one dialect to another. Instead of generating Fujaba-conformant story diagram models from the pattern specifications, we now generate story diagram models that conform to the interpreter's story diagram meta-model (signified in Figure 2 by the different shape and number of elements in the detection algorithm models).

[3]http://www.fujaba.de

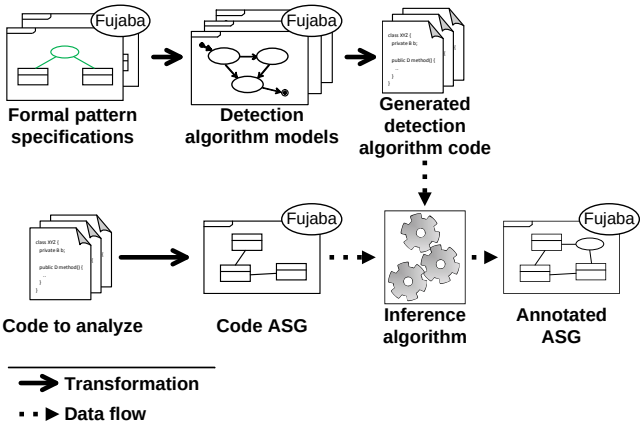

Figure 1: Original pattern detection process.

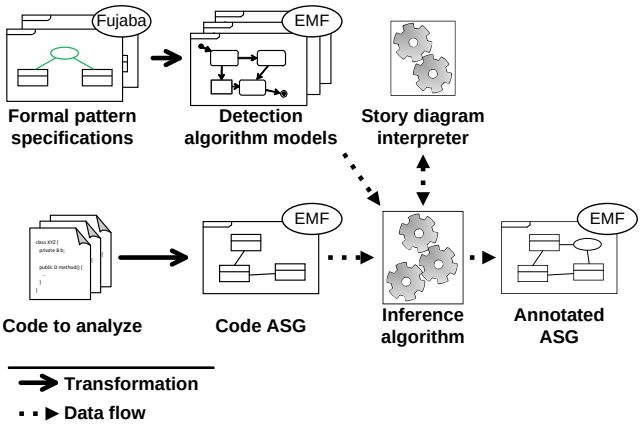

Figure 2: Adapted pattern detection process.

Second, Reclipse and the story diagram interpreter are implemented based on different frameworks. While Reclipse is based on the CASE tool Fujaba with its model format and API, the story diagram interpreter is based on the Eclipse Modeling Framework (EMF)[4] [14] and takes Ecore models as input. Thus, we also had to adapt or convert the class and story diagram models as well as the input ASG model from one technology to another (signified in Figure 2 by the ellipses with the EMF inscription).

Furthermore, instead of executing generated code we now have to run the interpreter on a detection algorithm model which enforces adaption of the inference algorithm.

3.1 Bridging the story diagram dialects

Reclipse and the story diagram interpreter use different story diagram meta-models which have different expressive power. Some things that can be modeled with Reclipse's meta-model cannot be modeled with the interpreter's meta-model. Other things are modeled differently. The Reclipse meta-model, for instance, contains an element called *statement activity* which can hold arbitrary Java code that is later integrated into the generated code. The interpreter obviously

[4]http://www.eclipse.org/modeling/emf/

does not generate any code, so its meta-model does not contain such an element.

Thus, to use the story diagram interpreter the Reclipse story diagram models had to be transformed into story diagram models conforming to the meta-model of the interpreter. This transformation had to take the meta-model differences into account. That means, some things are transformed into more complex "workaround" models and others cannot be transformed and thus may no longer be used in pattern specifications (unless the interpreter is extended).

For each element, we described a transformation rule from the Fujaba story diagram meta-model to the interpreter's story diagram meta-model, if possible. In the following, we describe the transformation of *story patterns* as an example. The full list of transformations can be found in a Master's thesis [4].

Story diagrams describe graph transformations. They closely follow UML activity diagrams and contain a number of story patterns connected by transitions that define the control flow. A story pattern contains a structure of objects that, if it is matched in a host graph (e.g. found in the ASG), is modified as defined by the story pattern (e.g. is annotated).

Figure 3 shows the meta-classes used to model story patterns in Fujaba (top) and the corresponding meta-classes of the story diagram meta-model used by the interpreter (bottom). In Fujaba, a story pattern (*UMLStoryPattern*) is contained in an activity (*UMLStoryActivity*). This activity can be marked as a *for-each* activity, which describes a loop in the control flow. A UMLStoryPattern contains a number of items (*UMLDiagramItem*) which are objects (*UMLObject*), links (*UMLLink*) and method calls (*UMLCollabStat*). Additionally, a story pattern can contain textual (Java) constraints (*UMLConstraint*) and *maybe* constraints that weaken the matching rule (i.e. allow to map more than one node in a story pattern to the same node in an ASG).

The story diagram meta-model of the interpreter in comparison combines the two classes *UMLStoryActivity* and *UMLStoryPattern* into one (*StoryActionNode*). It can contain constraints that are described by a hierarchy of *Expressions*. In the Fujaba model the Java constraints are integrated into the generated code whereas the interpreter evaluates the expression hierarchy. *Maybe* constraints are not supported by the interpreter. Objects and links are separately linked to a StoryActionNode. Methods calls are handled by another type of activity not shown in Figure 3.

This example shows that most elements can be translated quite easily, but some elements (e.g. *maybe* constraints) cannot be translated at all. We defined transformations for all elements that could be translated. These also contained elements that are part of the interpreter's meta-model, but not yet evaluated by the interpretation engine itself. So, once the engine is extended to evaluate these elements, they can be used for pattern specification again.

3.2 Bridging technical differences

Reclipse is based on Fujaba and the story diagram interpreter is developed with EMF. Fujaba and EMF are not

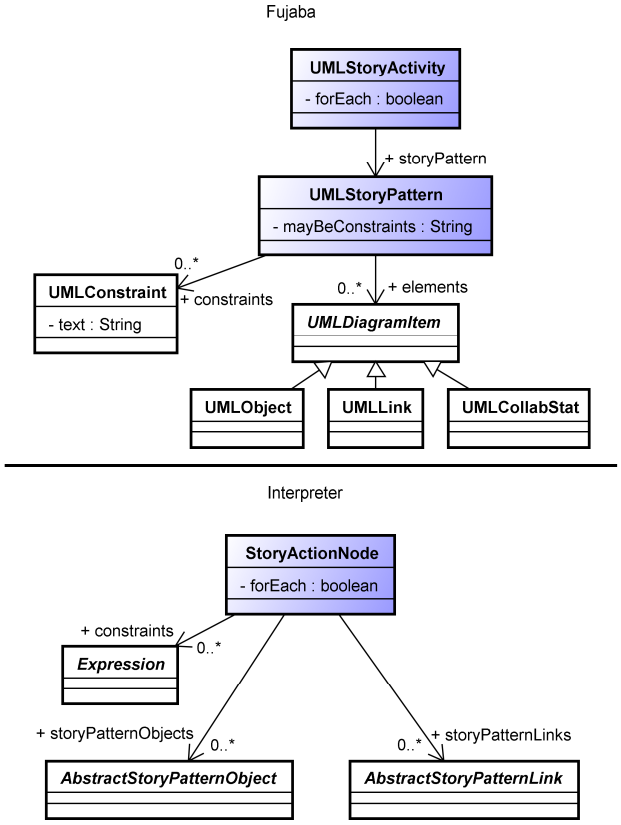

Figure 3: Story patterns in the two story diagram meta-models.

compatible. The saved models have different formats, the models are on different meta levels (UML vs. eMOF [11]) and the generated code follows different implementation conventions. To use the story diagram interpreter, we converted the Fujaba-based models generated by Reclipse (esp. story diagrams) and ASG models into EMF-based models.

The story diagram meta-model of the interpreter was created using EMF. Using the transformation described in Section 3.1, we could translate Fujaba story diagram models into EMF story diagram models. Based on this transformation, the Reclipse component that generates detection algorithm models from pattern specifications was replaced with a new component that creates detection algorithm models that are conformant to the interpreter's EMF-based meta-model. The new component was implemented manually. This way, it could easily be derived from the former (also manually implemented) component, so that it fits into the process and conforms to its interfaces.

Because of the interpreter's requirement to get EMF-based models as input, we also switched from the Fujaba-based ASG meta-model to an EMF-based ASG meta-model. The former, hardly maintainable component for parsing source code into an ASG was replaced by a manually implemented slim component that uses an existing Eclipse plug-in for parsing Java code (JDT[5]).

[5]http://www.eclipse.org/jdt/

3.3 Inference adaptations

In the original pattern detection process the inference algorithm selects an element of the ASG and starts the search for a pattern by executing the corresponding generated code. As we removed the code generation step, the inference algorithm had to be adapted so that it triggers the interpretation of a story diagram rather than the execution of code.

In addition, the inference algorithm had to be adapted to the use of EMF-based models instead of Fujaba-based models. Thus, we adapted existing interfaces and introduced new ones in the inference algorithm's implementation. This way, the reverse engineer now can choose whether she wants to use the former process based on code generation or the one based on the interpreter. The needed code adaptations and additions had to be done manually, because of the complexity of the existing (manually evolved) code base.

4. EVALUATION

We applied the adapted pattern detection process to evaluate our success. The traceability of the process improved, because the engineer no longer needs to bridge the gap between generated code and story diagrams. The interpreter provides a log of all interpretation steps. The coming story diagram debugger will further improve the traceability by visualizing the current state of execution and offering opportunities to observe and influence the execution. Although there are some limitations in the use of the story diagram interpreter, the first detection results are promising.

4.1 Limitations

Most story diagram elements could be translated from the one meta-model to the other. Except for *maybe* constraints, all non-translatable elements are provided in the interpreter's meta-model, but not yet supported by it's execution algorithm. For example, *paths* are not supported so far. A path between two ASG nodes describes that there is a directed, possibly indirect connection between the nodes. In the meta-model paths are represented by a special type of link between objects, but the execution algorithm does not separately handle them. In the story diagram translation we included these elements, so that they can be used as soon as the interpreter supports them.

4.2 Detection results

We evaluated the adapted detection process by detecting patterns in JUnit[6] 4.8.2 and comparing the detection results with those obtained with our previously applied detection process. For this purpose, we re-used an existing catalog of design pattern [5] specifications and auxiliary subpattern specifications.

The catalog had to be modified due to the limitations of the interpreter and its story diagram meta-model (cf. Section 4.1). Pattern specifications that could not be modeled for use with the interpreter were removed for both detection processes. Pattern specifications that had to be weakened (some conditions had to be removed because of the lack of expressiveness) for the interpreter use, were only modified for the run of the adapted detection process.

[6]http://www.junit.org

Pattern	SDI	CodeGen
AbstractStructureImplementation	114	78
AbstractType	17	16
ContainerWriteAccessMethod	379	12
DirectGeneralization	77	75
Field	194	191
Implementation	29	29
IndirectGeneralization	66	31
InterfaceType	13	10
MultiReference	276	7
OverriddenMethod	157	122
SingleReference	135	115
TemplateMethod	86	11
Visitor	1	1

Table 1: Pattern detection results.

Table 1 contains the pattern detection results. The "SDI" column lists the number of pattern implementation candidates detected by the adapted process using the story diagram interpreter. The column "CodeGen" lists the number of candidates detected with the original process using generated code.

As some patterns had to be removed from the catalog because of the interpreter limitations, the only "real" design pattern implementations found by either detection process were *Template Method* and *Visitor*. The latter was found equally often. Candidates for the *Template Method* pattern were found more often by the adapted process than by the original. The same holds for most other patterns (e.g. *ContainerWriteAccessMethod* and *MultiReference*). This is a result of the weakened pattern specifications. For example, the story diagram meta-model used by the interpreter does not support paths. So, they had to be removed, meaning that instead of searching for connected nodes, arbitrary, possibly unconnected nodes satisfying all other conditions are searched in the ASG. This results in more matchings.

Despite the fact that we have more false positives with our adapted detection process, the results are promising. Avoiding the code generation step significantly increases the traceability of the pattern detection. Debugging the pattern specifications and the detection process is easier and will be even more traceable with the interpreter's debugger [8]. The results obtained with the interpreter deviate from the original results (cf. Table 1) solely because of the weaknesses in the current interpreter implementation. Our story diagram translation already supports some story diagram elements that the interpreter does not yet consider. Thus, by improving the interpreter the pattern specifications will become more sophisticated and the detection results will equal the results achieved by the original process.

5. RELATED WORK

The overall goal of our work was to simplify the debugging of the detection process. The two main challenges with our approach were the transformation of the story diagram meta-models and the migration from Fujaba to EMF.

Geiger and Zündorf developed a tool to debug code generated by Fujaba and connect it at runtime to the correspond-

ing story diagrams [6]. As an alternative to using the story diagram interpreter, we could have used this approach, but that would have made the detection process more complex instead of simplifying it and we could not have benefited from the possible performance gain resulting from the use of information that is only available at runtime [7].

There are numerous approaches for model-to-model transformation which we could have used to translate the story diagrams from one meta-model to the other. Among them are *Triple Graph Grammars* (TGGs, [13]) and OMG's QVT (Query/View/Transformation, [12]). These approaches support model synchronization and bidirectional transformations. We decided against generating story diagrams conforming to one meta-model and then translating them to story diagrams conforming to another meta-model during each pattern detection. Instead, we decided to adapt the story diagram generation once and omit the creation of obsolete story diagram models. Since we already had a generator for Reclipse's story diagrams, we only had to replace the creation of story diagram elements such that they conform to the new meta-model. Furthermore, Java code represented by plain text in generated story diagrams significantly complicates the translation with TGGs and QVT.

Amelunxen et al. [1] developed an approach to tool integration using TGGs. This approach still requires manual code adaptations and as it uses TGGs it has the aforementioned disadvantages. Thus, we chose another solution.

6. CONCLUSIONS AND FUTURE WORK

To simplify the pattern detection process of the Reclipse tool suite and support the engineer in finding mistakes in his specifications, we integrated a story diagram interpreter and removed the code generation step.

The used interpreter still has some limitations. It does not yet support certain story diagram features that were supported by the formerly used story diagram meta-model. Adding these features is future work which is already started by the SDM unification task force that aims to unify the meta-models used by several teams from the Fujaba community.

Furthermore, the story diagram debugger [8] needs to be integrated. This debugger would simplify the search for pattern specification errors. The language used for pattern specifications is partly very similar to story patterns. So, if the debugger reveals an error in a story pattern, it will be easy to find the corresponding element in the pattern specification.

7. REFERENCES

[1] C. Amelunxen, F. Klar, A. Königs, T. Rötschke, and A. Schürr. Metamodel-based tool integration with MOFLON. In *ICSE '08: Proceedings of the* 30^{th} *International Conference on Software Engineering, Leipzig, Germany*, pages 807–810, 2008.

[2] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, Sept. 2009.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764, pages 296–309. Springer Verlag, November 1998.

[4] M. Fockel. Interpretation von Graphtransformationsregeln zur statischen Erkennung von Software-Mustern. Master's thesis, University of Paderborn, Oct. 2010. (In German).

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.

[6] L. Geiger and A. Zündorf. Design Level Debugging with Fujaba. In *International Workshop on Graph-Based Tools (GraBaTs), Barcelona, Spain*, 2002.

[7] H. Giese, S. Hildebrandt, and A. Seibel. Improved Flexibility and Scalability by Interpreting Story Diagrams. In T. Margaria, J. Padberg, and G. Taentzer, editors, *Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, volume 18. Electronic Communications of the EASST, 2009.

[8] A. Krasnogolowy. Entwurf und Implementierung eines Debuggers für Story-Diagramme. Master's thesis, Hasso-Plattner-Institut für Softwaresystemtechnik GmbH, Potsdam, Germany, 2010. (In German).

[9] U. A. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA Environment. In *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, 2000.

[10] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, FL, USA*, pages 338–348. ACM Press, May 2002.

[11] Object Management Group. Meta Object Facility (MOF), Jan. 2006. OMG document formal/2006-01-01.pdf.

[12] Object Management Group. Query/View/Transformation (QVT), Apr. 2008. OMG document formal/08-04-03.pdf.

[13] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, 20^{th} *Int. Workshop on Graph-Theoretic Concepts in Computer Science, Heidelberg, Germany*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163. Springer Verlag, 1994.

[14] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2^{nd} edition, Dec. 2008.

[15] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa*, 2010.