Fuzzy Logic based Interactive Recovery of Software Design

(Research Abstract)

Jörg Niere
Software Engineering Group
Department of Mathematics and Computer Science
University of Paderborn
Warburger Straße 100, D-33098 Paderborn
Germany
nierej@uni-paderborn.de

ABSTRACT

This abstract presents an approach to semi-automatically detect pattern instances and their implementations in a software system. Design patterns are currently best practice in software development and provide solutions for nearly all granularity of software design and makes them suitable for representing design knowledge. The proposed approach overcomes a number of scalability problems as they exist in other approaches by using fuzzy logic, user interaction and a learning component.

1. MOTIVATION

"*Never touch a running system*" is one of the most famous idioms in computer science and best practice for many software systems. This results from the experience that changing those software systems often has side-effects usually in parts that have not been adjusted to the change. The side-effects usually result from an awful documentation of the software system dependencies, because time-to-market has often a higher priority compared to an expensive good documentation of the system. Large and older systems, so-called *legacy systems*, often contain even no or only fragments of documentation. A re-documentation of those systems is usually very expensive, because it is mostly done manually.

Today, the *Unified Modelling Language* (UML) has become a standard for describing software systems. The UML consists of several different diagram types used for different purposes in a software development process. It is therefore naturally to use the UML diagrams also for re-documenting existing system, because they are common knowledge for nearly all developers and enable a seamless integration later in a redesign process.

In addition to the different diagram types in UML, *design patterns* [GHJV95] are best practice in software development. Design patterns introduced by Gamma et al., former known as Gang-of-Four (GoF) patterns, provide solutions for recurring problems. Today patterns of all granularity of software design exist in literature, e.g. implementation patterns, distribution patterns, architecture patterns and design patterns. If a pattern is used in an actual software system's design, it is called a *pattern instance*. Typically, there exist many different pattern instances for one pattern and even the actual implementation of a pattern instance can differ from one instance to another.

GoF-patterns provided by Gamma et al. are the result of an intensive (more or less manual) reengineering process of existing software systems at Big Blue. Consequently, GoF-patterns can be seen as a comprised collection of recurring successfully employed implementations made by independent developers in different software systems. This makes them highly suitable as a mean for legacy system understanding and as a representation of design knowledge. In addition, patterns connect several parts of a system which makes them ideal to document dependencies.

Precise Pattern Definition

A GoF-pattern provides a solution for a problem in terms of a definition of the static and dynamic behaviour including usually one example and one implementation possibility. Thereby most parts of a GoF-pattern's description are informal which offers many interpretation opportunities. Typically, the static structure of a GoF-pattern is given as an OMT [RBP+91] diagram, which is comparable with an UML class diagram, while the behaviour is mostly described in prose and thereby not formally defined. To support an automated recognition of GoF-pattern and other pattern instances in existing software systems a formal definition of a pattern is indispensable.

Applicability to Large Software Systems

In addition to a formal definition of a pattern the success of an automated recognition process of pattern instances highly depends on its scalability. Tools supporting an automated recognition of pattern instances must be able to analyse thousands or millions lines of code (LOC). The scalability is often strongly depending on the number of pattern definitions. Raising the scalability often means a reduction of the number of pattern definitions or relaxing the definitions in such a way that one pattern definition covers more than one pattern instance or implementation. Both, reducing the number of pattern definitions and relaxing the definitions, reduces the preciseness of the analysis where in the first case not all pattern instances are found and in the second case *false-positives* occur (erroneously recognized instances and implementations).

Adapting Patterns to a Specific Domain

In practice it is impossible to run a fully automated analysis with a catalogue consisting of all pattern instance and implementation definitions for all patterns. Fortunately, for the

This research is part of the FINITE project funded by the German Research Foundation (DFG)
http://www.upb.de/cs/ag-schaefer/Forschung/Projekte

analysis of a software system, it is usually sufficient to take only those patterns into account, which are relevant for the software system's domain. Focusing on a specific domain reduces the number patterns dramatically but does not solve the problem of a complete enumeration of all pattern instances and implementations of one pattern in one domain. Thus, a reengineering process has to be interactive where the engineer must be able to adapt a pattern instance or implementation definition to the actual system in a certain domain during the analysis. In addition, such an interaction allows the engineer to infer personal hypothesises and presumptions and to integrate results from other analyses, e.g. original documentation or interviews with the developers of the system.

2. RELATED WORK

There exist several approaches in the field of program analysis and program understanding in the literature, but this section focuses on approaches which also try to detect pattern instances and which are related to the used techniques in this approach.

Comparable work on reengineering source code stems from Harandi and Ning [HN90] who present program analysis based on an Event Base and a Plan Base. A parser constructs rudimentary events from the source code to be analysed. Plans define the relationship between (incoming) event(s) and they fire a new event corresponding to the plans intention. Each plan corresponds to exactly one pattern variant instance or implementation, which lets the approach fail for large systems, because of the large number of different pattern instances or implementations for even one pattern. The same holds for the approach of Paul and Prakash [PP94]. Both approaches use a deductive execution semantics where in each deduction step pattern matching techniques are used. Thus, both approaches can be seen as a basis for other approaches.

Wills [Wil96] presents an approach to identify common computational structure such as searching or sorting algorithms. Wills uses also pattern matching techniques where patterns are encoded as rules stemming from a special graph grammar. The graph grammar combines control flow and data flow and is thereby comparable to Harandi and Ning's plans. Unfortunately, an evaluation shows that her approach is only able to analyse a few thousand lines of code, because the subgraph isomorphism problem is NP-complete and her pattern matching algorithm is a rudimentary implementation.

Jahnke [Jah99] presents a successful result analysing large relational database systems. He integrates the re-engineer in his analysis process and is able to handle the large search space. In addition, possibilistic logic integrated in his Generic Fuzzy Reasoning Nets (GFRNs) handles uncertainty. GFRNs use so-called clichés as irrevocable facts, which are detected in the database's code using a classical pattern matching approach. This is sufficient for the analysis of relational database systems but fails for the recovery of pattern instances or implementations. Applying the process as well as the GFRNs to the recovery of some easy implementation pattern instances fail, because there exist only a few number of cliché instances of one cliché in comparison to the large number of pattern instances and implementations, cf. [JNW00].

Concerning the detection of design patterns, Kraemer and Prechelt [KP96] present an approach of analysing C/C++ source code to extract design patterns. Hence, they analyse the header files (structural parts) only, they get many false-positive, because many design patterns are structural identical but behavioural different.

Analysing behaviour as well as structure using patterns is presented by Keller et al. in [KSRP99]. A common abstract syntax graph model for UML is used to represent source code as well as patterns. Matching the syntax graph of a pattern on the syntax graph of the source code is done using scripts. The scripts have to be implemented manually. The definitions in such a script language become difficult to maintain and to reuse. It is also not possible to modify a script, which means an adaptation of a pattern to a certain domain, during the analysis run-time. Thus, the approach fails for the analysis of unknown software systems.

Tonella and Antoniol's [TA99] approach of detecting patterns is orthogonal to the other ones. They do not use pre-defined patterns but try to find a pattern instance in source code but analyse the code for recurring constructs. Statistic evaluation summarizes the analysis result and present which construct occurs how many times in the program. This approach seams to be more useful for the detection of (new) patterns than for pattern instances but could be used to identify unknown pattern instances or implementations in unknown systems. Categorizing the found results and comparing them to existing patterns has to be done manually.

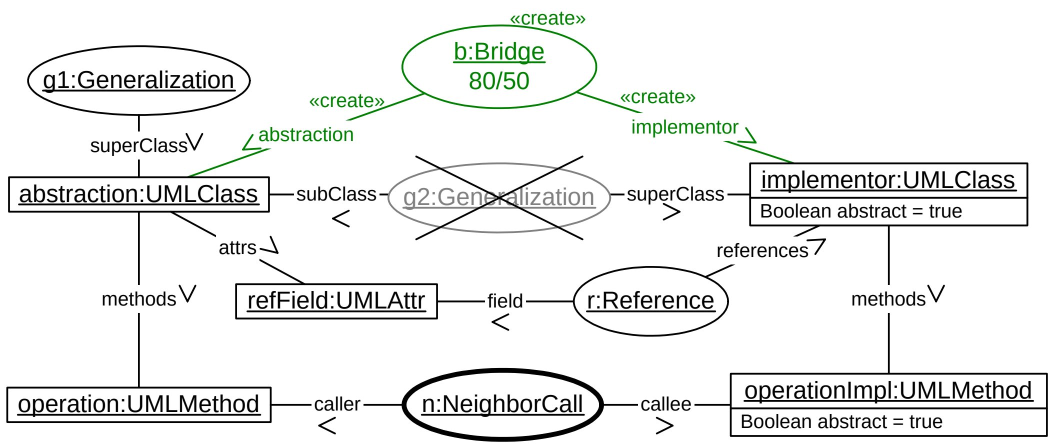

Figure 1. Bridge-GoF-pattern definition

3. MY APPROACH

This research presents a reengineering system including a process and techniques to extract pattern instances from a software system's implementation semi-automatically. Although the focus lies on GoF-patterns the approach can also be used to detect other kinds of pattern instances such as architectural, implementation or distribution patterns. The approach uses layered graph grammars and graph parsing techniques [RS95] combined with fuzzy logic [ZK92] to provide an interactive pattern matching algorithm.

Graph grammars [Roz97] build the formal basis of the approach. Patterns are encoded as graph transformation rules and the to be analysed source code is parsed in its abstract syntax graph representation. The graph transformation rules are notated as UML collaboration diagrams, which are common knowledge and reduce the learn effort for other engineers. For example, Figure 1 shows the definition of a Bridge pattern. Oval shaped objects represent other pattern instances and rectangle shaped objects represent abstract syntax graph objects. The objects in the rule must form a connected graph and all links are an instance of an association in an corresponding class diagram. The links have a certain read direction notated as an arrow at the link's name but are traversable in both directions. Crossed-out elements represent negative application conditions.

Common parts in different patterns can be defined in separate rules and can be (re)used in the rules for the original patterns. Such common parts are called *sub-patterns* or *sub-rules*. The Bridge pattern uses the Generalization, Reference and NeighborCall pattern. In addition to the composition of patterns the approach supports also (structural) inheritance in object-oriented terms. Both raises the reuse and reduces the number of rule definitions. To handle the large number of implementation variants of a pattern instance, rules defining patterns and sub-patterns are enhanced with fuzzy values to describe a degree of uncertainty, cf. [ZK92]. Uncertainty allows one rule to match for several implementations with a certain degree. This reduces the number of required rule definitions dramatically.

The detection algorithm uses a certain graph parsing technique, cf. [RS95], which annotates the abstract syntax graph of the source code for any found pattern. In case of the Bridge pattern, a new b:Bridge object is created. The algorithm uses a *forward/backward chaining* (combined bottom-up, top-down) strategy.

The bottom-up strategy is similar to deductive approaches and brute-force algorithms, e.g. used by Wills, but my algorithm tries to apply patterns with a high partial order number. Usually, applying those patterns fails, because other sub-pattern are needed. In this case the algorithm switches into top-down strategy, which uses information annotated by the patterns to establish or revert

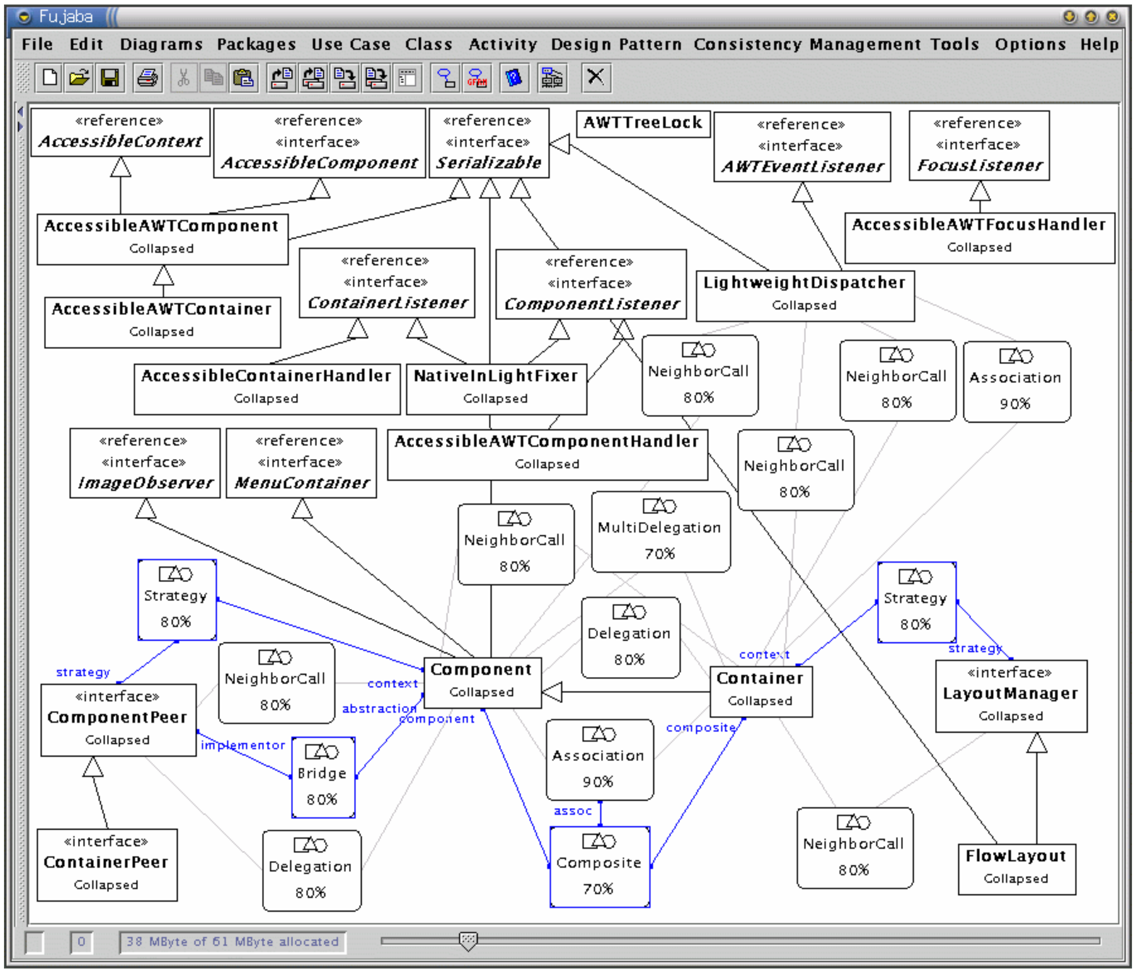
Fujaba
File Edit Diagrams Packages Use Case Class Activity Design Pattern Consistency Management Tools Options Help
«reference» AccessibleContext
«reference» AccessibleComponent
«reference» «interface» Serializable
AWTTreeLock
«reference» «interface» AWTEventListener
«reference» «interface» FocusListener
AccessibleAWTComponent Collapsed
AccessibleAWTFocusHandler Collapsed
AccessibleAWTContainer Collapsed
«reference» «interface» ContainerListener
«reference» «interface» ComponentListener
LightweightDispatcher Collapsed
AccessibleContainerHandler Collapsed
NativeInLightFixer Collapsed
NeighborCall 80%
Association 90%
«reference» «interface» ImageObserver
«reference» «interface» MenuContainer
AccessibleAWTComponentHandler Collapsed
MultiDelegation 70%
Strategy 80%
Delegation 80%
strategy
context
Component Collapsed
Container Collapsed
«interface» LayoutManager Collapsed
«interface» ComponentPeer Collapsed
abstraction
component
implementor
Bridge 80%
composite
assoc
Composite 70%
«interface» ContainerPeer Collapsed
FlowLayout Collapsed
0
38 MByte of 61 MByte allocated

Figure 2. Part of the AWT library analysis result

sub-patterns. Such an interleaved strategy accelerates providing first interesting analysis results. This is the major advantage in comparison to pure deductive approaches, which are usually able to provide interesting results only after a complete analysis.

Typically, the found analysis results are uncertain corresponding to the fuzzy values defined in the rules. The fuzzy value of a found Bridge pattern is the minimum of the rules fuzzy value (80) and all depending pattern instances. Uncertain results are accepted or rejected automatically when the uncertainty is higher or lower a certain threshold (50 in case of the Bridge pattern in Figure 1) or the engineer can accept and reject results manually. A learning component logs the interactions and recalculates the fuzzy values of the rules, and the acceptance and rejection thresholds based on statistic analyses. This automates further analysis. In addition to the acceptance or rejection of uncertain results, the engineer is also able to adapt the rules to a certain domain during the analysis process. This becomes very efficient in combination with the early delivery of interesting results by the detection algorithm, because actions taken by the engineer always influence further analysis. For example, in case of an emergency, i.e. all instances are false-positives or no instance is found, the engineer can stop the analysis, investigate the current results and adapt the rules adequately in a very early analysis state.

The reengineering system is specified using UML and the Fujaba environment, cf. [FNTZ98, KNNZ00]. Fujaba provides editors for UML class and activity diagrams and a code generation algorithm and is used to specify the reengineering system as well as its results. The effectiveness of my approach and the tools is shown analysing large systems. For example Java's Abstract Window Toolkit (AWT), the SWING library with about 200k LOC and Fujaba itself with more than one million LOC.

Figure 2 shows a cut-out of the AWT library class diagram consisting of the kernel classes and annotated by the patterns Composite, Strategy and Bridge including their preciseness. Note, the class diagram contains only rudimentary directed references extracted from the source code. Associations are a part of the pattern catalogue and thus represented as annotations.

All examples have been built using design patterns, though the evaluation is used to show the preciseness of my approach, extracting GoF-pattern instances and comparing them with the documentation of the systems. Using the reengineering system analysing a legacy system shows the application of my approach on foreign implementations. Thereby the advantages of integrating the engineer in the process are also investigated and discussed.

The current Fujaba analysis prototype is available at `http://www.upb.de/cs/fujaba` and for more details on the algorithm and the evaluation see [NSW+02].

REFERENCES

[FNTZ98] T. Fischer, J. Niere, L. Torunski and A. Zündorf. *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language*. In G. Engels and G. Rozenberg, editors, Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany, LNCS 1764. Springer Verlag, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[HN90] M.T. Hanrandi and J.Q. Ning. *Knowledge Based Program Analysis*. IEEE Transactions on Software Engineering, 7(1):74–81, IEEE Computer Society Press, 1990.

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999.

[JNW00] J.H. Jahnke, J. Niere and J.P. Wadsack. *Automated Quality Analysis of Component Software for Embedded Systems*. In Proc. of the 8th International Workshop on Program Comprehension (IWPC), Limerick, Irland, pages 18–26. IEEE Computer Society Press, June 2000.

[KNNZ00] H.J. Köhler, U. Nickel, J. Niere and A. Zündorf. *Integrating UML Diagrams for Production Control Systems*. In Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, pages 241–251. ACM Press, 2000.

[KP96] C. Krämer and L. Prechelt. *Design recovery by automated search for structural design patterns in object-oriented software*. In Proc. of the 3rd Working Conference on Reverse Engineering (WCRE), Monterey, CA, pages 208–215. IEEE Computer Society Press, November 1996.

[KSRP99] R.K. Keller, R. Schauer, S. Robitaille and P. Page. *Pattern-Based Reverse-Engineering of Design Components*. In Proc. of the 21st International Conference on Software Engineering, Los Angeles, USA, pages 226–235. IEEE Computer Society Press, May 1999.

[NSW+02] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals and J. Welsh. *Towards Pattern-Based Design Recovery*. In Proc. of the 24th International Conference on Software Engineering (ICSE), Orlando, Florida, USA, ACM Press, May 2002.

[PP94] S. Paul and A. Prakash. *A Framework for Source Code Search Using Program Patterns*. IEEE Transactions on Software Engineering, 20(6):463–475, IEEE Computer Society Press, June 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Permalani, F. Eddy and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Roz97] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[RS95] J. Rekers and A. Schürr. *A Graph Grammar Approach to Graphical Parsing*. In Proc. of the IEEE Symposium on Visual Languages, Darmstadt, Germany. IEEE Computer Society Press, 1995.

[TA99] P. Tonella and G. Antoniol. *Object Oriented Design Pattern Inference*. In Proc. of the 5th Symposium on Software Development Environments (SDE5), pages 230–238. IEEE Computer Society Press, September 1999.

[Wil96] L.M. Wills. *Using Attributed Flow Graph Parsing to Recognize Programs*. In Proc. of of the 5th International Workshop on Graph Grammars and Their Application to Computer Science, LNCS 1073, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.

[ZK92] L.A. Zadeh and J. Kacprzyk. *Fuzzy Logic for the Management of Uncertainty*. John Wiley and Sons, Inc., 1992.