Combining Static and Dynamic Analyses to Detect Interaction Patterns

Dirk Heuzeroth
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
heuzer@ipd.info.uni-karlsruhe.de

Thomas Holl
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
holl@ipd.info.uni-karlsruhe.de

Welf Löwe
Universität Karlsruhe
Program Structures Group
Adenauerring 20a
76133 Karlsruhe, Germany
loewe@ipd.info.uni-karlsruhe.de

ABSTRACT

We detect interaction patterns in legacy code combining static and dynamic analyses. The analyses do not depend on coding or naming conventions. We classified potential pattern instances according to the evidence our analyses provide. We discuss our approach with the Observer Pattern as an example. Our Java implementation analyzes Java programs. We evaluated our approach by self applying the tool looking for Observers in its code. We do not miss a pattern instance. The class of pattern instances our analyses provided a high evidence for, contains 80% of all actual pattern instances but no false positive.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Software Architectures, Reusable Software

1. INTRODUCTION

Systems integration is personally, physically, and temporally separated from component design. Hence, mismatches between components are the rule, not the exception, making adaptation an integral part of component-based systems design. This almost always affects the interaction of components and thus the system architecture [8].

In existing legacy systems, we often have to identify components, first. To perform the necessary changes, programmers further need to understand the system architecture and behavior. Since the system architecture is almost always scarcely documented or even not available, discovering or recovering design information from existing systems is crucial for understanding and refactoring these systems. Therefore, tools to automatically extract design and architectural information are required.

We propose to retrieve static as well as dynamic information. Both are then combined to obtain the desired information on the pattern to be detected. There are situations, where neither static nor dynamic analyses alone are sufficient (or not with acceptable expenses). E. g., it is not statically computable, which method or attribute is actually called or accessed at run time and how often. Even data flow analyses cannot predict all branches and loops, especially when the program to be analyzed requires user interactions. As objects are created at run time, relations over objects are dynamic by nature.

The idea is to distinguish a static and a dynamic pattern. The former restricts the code structure the latter the runtime behavior. Analyzing with the static pattern results in a set of candidate instances in the code. In practice this set is large and programmers hardly want to screen all of them to detect the actual instances. Therefore, we test executions of the instance candidates found by the static analysis wrt. the dynamic pattern.

The results of dynamic analyses depend on an execution of the candidate instances. Methods not executed at run time cannot be evaluated wrt. the dynamic pattern thus providing no information. However, testing techniques and environments guarantee that each reachable program part is executed while testing (of cause not every program sequence). Using these techniques, we may consider dynamic information available for each candidate instance. Moreover, we argue that parts that are less frequently executed are also less critical for understanding and for restructuring.

Our approach requires

- the source code to be available, and
- the programs to be executable to observe their dynamic interaction aspects.

We explicitly excluded all dependencies to coding and naming conventions. Hence, our approach also detects interaction patterns occurring by chance.

In the sequel, we consider the Observer Pattern [7] (event notification) as a special architectural pattern. It is frequently used in frameworks and applications to realize loose coupling of objects or components. Suppose the following

scenario: we tailor the framework or application for an environment requiring efficient communication among statically known partners. In such a setting, the Observer Pattern would be inappropriate. Thus, we need to detect it and replace it by a more efficient solution.

We need static and dynamic analyses to detect the Observer Pattern. The static analysis computes a set of classes that fulfill the necessary properties for subject and corresponding listener classes. The dynamic analysis then monitors objects of these classes during execution and checks whether the interaction among them satisfies the dynamic Observer protocol.

We present our approach comprising the static and dynamic analyses in Section 2. This section also discusses implementation details. In Section 3 we evaluate the results of applying our analyses to the code of our tool. Section 4 discusses related work. Finally, we conclude and show directions of future work in Section 5.

2. APPROACH

In this section we present our approach to detect interaction patterns by combining static and dynamic analyses. Section 2.1 introduces our component model to show the goal of the analyses. Section 2.2 discusses the static analysis, Section 2.3 the dynamic analysis. Section 2.4 sketches the design of our tool performing static and dynamic analyses.

2.1 Component Model

For the purpose of this paper, we define components to be software artifacts with typed input and output ports. This definition focuses on computational components, but is sufficiently general to cover all other variants. Input ports are connected to output ports via communication channels called connectors. The notion of ports and connectors are known from architecture systems [14, 2].

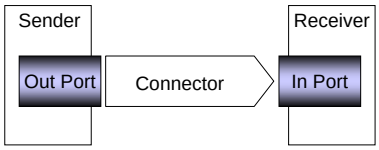

Figure 1: Basic Component Model

Some connectors may be as complex as most components, and thus require the same amount of consideration in design, but they all base on simple synchronous or asynchronous point-to-point data paths. Figure 1 sketches this basic component model.

In general, ports and connectors are implemented by patterns using basic communication constructs like calls, RPCs, RMIs, input output routines etc. provided by the implementation language or the component system. The Observer Pattern is such a port and connector implementation as it connects an event generator with some listener objects. The notification generally involves calling an event handling method of the listeners, where the subject waits for every call to return. Although, the pattern can be considered as asynchronous communication, since the events may occur arbitrarily, the notification itself constitutes a synchronous action.

In contrast to such an implementation, the ports and connectors themselves abstract from details. A port defines points in a component that provide data to its environment and require data from its environment, respectively. A connector defines out-port and in-port to be connected and specifies whether data is transported synchronously or asynchronously.

In order to extract components from a system and adapt them to a new environment, we prefer a view on the system containing abstract ports and connectors. However, legacy (source) code only contains port and connector *implementations* scattered throughout the code. The goal of our analyses is to compute the abstract port and connector view on these systems. Implementations of ports and connectors follow communication design patterns. In order to retrieve an abstract view, we search for the patterns. The static analysis computes potential program parts playing a certain role in a communication pattern. The dynamic analysis further examines those candidates. We can thus consider static and dynamic analyses as filters that narrow the set of candidates in two steps. Figure 2 illustrates our approach.

In the subsequent subsections, we use the Observer Pattern as a running example. The following naming conventions refer to roles of certain methods of the Observer Pattern. Figure 3 sketches an implementation. Note that this naming convention is only used for explanations in this paper; the static analysis does not refer to those name.

addListener: a method responsible for adding listener objects to a subject object.

removeListener: a method responsible for removing listener objects from a subject object.

notify: a method responsible for notifying the listeners of a state change in the subject.

update: a method implemented by the listener objects, called by the notify method.

We assume that addListener, removeListener as well as notify are contained in a single class and are not distributed among different hierarchies. This is not an unnatural restriction, but reflects object-oriented design principles.

2.2 Static Analysis

The program source code is the basis for the static analysis; it is represented by an attributed abstract syntax tree (AST) as computed by common compilers. A *static pattern* is a relation over AST node objects. It is defined by a predicate P using the information in the attributed AST as axioms. Names of variables, methods and classes nodes may be compared with each other but not with constants, thus making the pattern definitions independent on naming conventions.

The static analysis reads the sources of the program in question and constructs an attributed AST. Then, it computes the pattern P relation on the AST nodes and provides the result as a set of Candidates, i. e., pattern instances with the appropriate static structure. This set is a conservative approximation to the actual patterns in the code. The dynamic analysis, cf. Section 2.3, refines this approximation

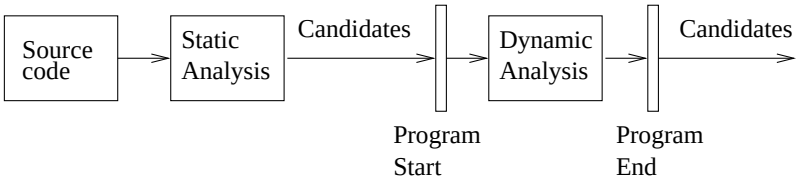

Figure 2: Process of detecting Communication Patterns.

```
public class Subject {
  private Container c = new Container();
  private State s = new State();

  public void addListener(Listener l) {
    c.add(l);
  }
  public void removeListener(Listener l) {
    c.remove(l);
  }
  public void notify() {
    if (s.notChanged()) return;
    for each l in c: l.update(s);
  }
}

interface Listener {
  public void update(Object o);
}

public class MyListener
  implements Listener {

  public void update(Object o) {
    doSomething(o);
  }
}
```

Figure 3: Pseudo code snippets sketching an implementation of the Observer Pattern

later on.

An Observer Pattern candidate is a tuple of method declarations of the form:

$$(S.addListener, S.removeListener, S.notify, L.update)$$

where S is the class declaration of the subject of observation and L the class or interface declaration of the corresponding listeners.

In practice, the candidate set is large. Brute force methods, e. g. Prolog like resolution, are therefore not appropriate for use in practical tools. The search should be more directed.

To produce the candidate set for our example, the static analysis iterates over all program classes and their methods. For each method m of a class c, we first assume it plays the addListener or removeListener role. Therefore, we consider each parameter type p of method m a potential listener, provided p is neither identical to nor a super or a subclass of class c ($p \not\prec c$). Such a relation would contradict the decoupling of subject and listeners as defined in the Observer Pattern. We therefore determine all method calls issued from inside methods of class c to some method u defined in the potential listener class p. The methods of class c containing the calls to p.u are considered as potential notify methods and the method p.u as update method. To test whether method c.n might be a notify method we use the predicate

isNotifyListener(c.n, p.u): returns true iff c.n calls p.u and p is not a parameter of c.n.

The result of the iteration is a set Y of tuples:

$$(S.addListener \mid removeListener, S.notify, L.update).$$

To compute the final set of candidates, we iterate over the tuples of set Y. We combine corresponding addListener and removeListener methods into one pattern candidate. If the add | removeListerner entry of Y satisfies the addListener predicate defined below, we combine it with all other tuples of Y that have the same notify and update entries to associate it with the corresponding removeListener candidates. We also consider the case that a removeListener method need not be implemented and thus always construct tuples with the removeListener entry set to null. The addListener role is defined by the predicate

isAddListener(a): tests, whether the method a potentially stores the passed argument for future use, i. e., checks whether the argument

- is used on the right hand of an assignment statement, i. e., storing the argument locally in the object,
- or is passed as an argument to another method, i. e., potential call of a store method.

Figure 4 shows the static analysis algorithm we obtain.

```
Candidates := ∅

for each class c : {
  Y := ∅
  for each method m in c :
    for each parameter type p in m where (p ⊀ c) :
      for each call from c.n to p.u, n and u methods :
        if (isNotifyListener(c.n, p.u))
          Y := Y ∪ {(c.m, c.n, p.u)}

  for each (c.a1, c.n1, p1.u1) ∈ Y :
    for each (c.a2, c.n2, p2.u2) ∈ Y where
        (c.n1 = c.n2 ∧ p1.u1 = p2.u2) :
      if (isAddListener(c.a1)) {
        if (c.a1 = c.a2) {
          Candidates := Candidates ∪ (c.a1, null, c.n1, p1.u1)
        } else {
          Candidates := Candidates ∪ (c.a1, c.a2, c.n1, p1.u1)
        }
      }
}
```

Figure 4: Pseudo code sketching the static search for Observer Patterns

Although the candidate set is computed quite efficiently by the directed search algorithm, we still face the problem of being too conservative with our approximation: the candidate set is large compared to the set of actual pattern instances and not appropriate for providing it to the system designer as it is. There are three possible solutions:

Use Expert Knowledge Many approaches require expert knowledge to further restrict the candidate set. It often refers to naming conventions of methods and classes. Such approaches rely on coding discipline, which is hardly a realistic assumption in legacy codes.

In our example, we could try to eliminate methods without prefix add from the addListener candidates. However, this would also exclude register methods.

Dynamic Analyses execute the program and check if the sequence of values of variables or contents of containers is appropriate, i. e., matches the dynamic pattern. This is the approach we pursue in Section 2.3.

For the Observer Pattern, we check if the addListener method in a candidate tuple actually registers the object the notify method in the same tuple is called on.

Data Flow Analyses try to statically approximate the sequence of values that some variables have at runtime. Actually, we could formalize all rules for the dynamic matches as data flow problems. Unfortunately, data flow equations cannot be computed by a straight forward search. Instead, they require a fix point iteration and are therefore much more expensive than the simple search. Moreover, they can only make very conservative and thus mostly worthless assumptions on data provided by the user at run time. They are also imprecise in approximating object ids and aliases.

For our example, we need an alias analysis checking whether the parameter of the addListener method in a candidate tuple is an alias for the access path to the object the corresponding notify method calls update on.

2.3 Dynamic Analysis

The static analysis provided tuples of AST nodes as candidates. The dynamic analysis takes this Candidates set as its input. It monitors the execution of the nodes of every tuple. It further tracks the effects of the executed nodes to check whether the candidate satisfies the *dynamic pattern*. The dynamic pattern is a protocol (formal language) over a set of *events*. Events are state transitions of the system to analyze, e. g., assignments or method calls. In case of a protocol violation, the candidate is marked and an error message is attached to it.

Each node of a candidate tuple is contained in a class definition or is a class definition itself. At runtime we might have many instance objects of these classes. Each set of those object instances should conform to the dynamic pattern. In our scenario, e. g., we might have more than one instantiation of the Observer Pattern defined by the subject and listener classes of a candidate tuple.

Moreover, patterns indicate $n : m$, $1 : n$, or $1 : 1$ relations among objects of the classes implementing a pattern. For each single candidate tuple, it could be required that the number of instance objects of their classes is restricted. The Observer Pattern, e. g., requires a $1 : n$ relation of the subject instances and their listener instances.

Altogether, we trace a set of instances for each candidate tuple of a pattern. Each such set may contain several objects per position in the tuple. Considering our Observer Pattern scenario, we thus assign to every candidate tuple

$$(\mathsf{S.addListener}, \mathsf{S.removeListener}, \mathsf{S.notify}, \mathsf{L.update}),$$

cf. Section 2.2, a set of instance tuples

$$\{(\ s.\mathsf{addListener}, s.\mathsf{removeListener}, s.\mathsf{notify}, \{l_1.\mathsf{update} \ldots l_n.\mathsf{update}\}\ )\}$$

where s is an instance of S and $l_1 \ldots l_n$ are instances of L. It is not necessary to store the subject s three times. Furthermore, the addListener, removeListener, notify, and update methods are already captured by the candidate tuple. So, to avoid redundancies, we only associate a set

$$\{(s, \{l_1 \ldots l_n\})\}$$

with each candidate tuple.

We monitor each node in a tuple of the candidates. Whenever we dynamically execute such a monitored node, we retrieve all the candidate tuples the node is contained in. Depending on the node's unique role in each single tuple, we execute dynamic test actions on the object sets associated to the corresponding candidate tuples.

In the Observer Pattern, we use the subject object as a key to retrieve the affected object set of each candidate tuple. To determine the proper object set, we distinguish two cases: If the method complies with the addListener, removeListener or notify roles, then the key subject object is the object the method is called on. If the method complies with the update role, then the key subject object is the object the corresponding notify method is called on.

The dynamic test actions for the Observer Pattern are:

addListener: We add the passed argument to the subject's list of listener objects. No protocol mismatch can be detected here.

removeListener: We remove the passed argument from the subject's list of listener objects. A protocol mismatch occurs, if the listener to be removed has not been added before. This can also be caused by a programming error. We therefore allow to turn off this criterion.

notify: We do not change the set of subject or listener objects. A correct protocol updates all or no listener objects (atomic update). To check this protocol, we have to distinguish between the method entry and the method exit. At the method entry, we mark all attached listener objects as not-updated. At the method exit we check whether all or no listener objects have been marked as updated. In this case, the protocol is satisfied. The other case indicates a protocol violation.

To accept the case of not updating any listener objects as a protocol match makes sense, because notify may

be called, although the subject's state did not change. Then there is no need to notify the attached listeners.

update: We do not change the set of subject or listener objects. If the update method has been called by the notify method of the same candidate tuple, we mark the listener object as updated. To recognize this, we need to detect the source of the method call, a functionality to be provided by the dynamic framework.

A call of update by the corresponding notify method is a protocol mismatch if the listener object has not been attached previously.

The dynamic analysis partitions the candidate tuples into the following categories:

Full match: Tuples contained in this category completely confirm to the dynamic pattern (protocol).

Listener objects are added via the addListener method, optionally removed with the removeListener method and their update method is called by the notify method at least one time. We distinguish $1 : n$ and $1 : 1$ matches; the latter conform to the protocol but only one listener is detected.

May match: At least one of the tuples nodes is executed, but only a correct prefix of the protocol is detected (could be completed to a correct protocol).

E. g., listener objects are added via the addListener method, but no update method is called.

Mismatch: Tuple violated the protocol requirements. The violation is logged via an error message.

No decision: None of the (monitored) nodes of a tuple is executed. Note that this category remains empty if we use a test environment, which guarantees the execution of each single program part.

2.4 Tool Design

We implemented the static algorithm using our Recoder [1] library. The library constitutes a Java framework for static analyses and program transformations. It consists of a compiler front-end, a pretty printer as a back-end, and a library of analysis, program generators and transformations. Currently, the front-end tool supports Java sources only, but the architecture in general can be applied to other languages as well. The front-end performs syntactic and semantic analysis including name and type analysis and provides an API to access the abstract syntax tree and the results of semantic analysis, e. g. type and inheritance information. It further provides functions to determine method calls, uses of definitions and much more. In our example implementation of the static analysis, we use this API to scan method declarations for conformance to the static Observer Pattern roles (cf. Section 2.2).

To obtain run time information there are four alternatives, as shown by [12]: Code instrumentation, annotation of runtime environments, post mortem analysis, and on-line debugging or profiling. We took into account two of the above alternatives for the dynamic analysis: the on-line debugger approach with the Java Debug Interface JDI and automatically instrumenting the code with Recoder transformations.

JDI is part of the Java Platform Debugger Architecture (JPDA) [9], the debugging support for the Java 2 Platform. JPDA provides the infrastructure needed to build end-user debugger applications. It consists of multiple layered APIs from which we only use the Java Debug Interface (JDI), a high-level Java programming language interface with the possibilities for launching run time environments for a program, starting a debugee and controlling the execution of this debugee by another program. Additionally, the control program can access the state of the debugee's execution.

On top of Recoder and JDI, we use the VizzAnalyzer [11, 12], our framework for the static and dynamic analysis of Java programs, providing a GUI to control analyses and visualizing their results.

The JDI approach allows to leave the source code unchanged. However, we experienced severe performance problems, since the debugee launches its own virtual machine. This leads to inter-process calls for obtaining information. Additionally, running programs in debug mode already slows down performance significantly. Since we aim at involving user interactions in future applications, the bad performance turns this approach unsuitable.

Instrumentation on the other hand eliminates the lack of performance. Moreover, all instrumentations are performed automatically with Recoder. Although, currently Recoder investigates Java sources only, the approach and the architecture can be applied to sources written in any typed language. A drawback is the recompilation of the instrumented program.

3. EVALUATION

To survey our tool, we apply it to the code of the tool itself (including the Recoder package). Statistics about the tool are given in Table 1.

	classes	methods	observers[1]
Recoder	590	6300	2
Analyzer	30	200	3
TOTAL	620	6500	5

Table 1: Statistics about the surveyed system

The main task of the static algorithm is to reduce the amount of candidates. In case of the Observer Pattern detection, it therefore applies the isAddListener and isNotifyListeners predicates as main criteria. This reduces the set of $7.5 * 10^{13}$ possible candidates[2] to 4800 tuples containing all 5 Observer Pattern instances. The corresponding analysis phase needs about 70 seconds on a Pentium III, 550Mhz, 256MB RAM, running Windows NT 4 with JDK 1.3.

Table 2 shows the results of the dynamic analysis. The "Detected" row lists the numbers of tuples of the corresponding category detected by our tool, whereas the "Real" row lists

[1] No subjects using delegation nor sub-classes of subjects.

[2] There are 6500 methods in the 4 possible roles plus 6500 methods in 3 roles and empty removeListener role: $\binom{6500}{4} + \binom{6500}{3}$.

	Full 1:n match	Full 1:1 match	May match	No decision	Mismatch
Detected	4	6	308	2015	2467
Real	4	0	0	1	0

Table 2: Results

the number of tuples of the corresponding category that represent real Observer occurrences.

The **Full** $1:n$ **match** category shows that all Observer Pattern instances used in that program run were classified correctly. The **Full** $1:1$ **match** column reveals that delegation confuses our analyses. The reason is, that delegation shows the same static and dynamic properties as the Observer Pattern. The only difference is that delegation always constitutes a $1:1$ relation. This is one of the reasons the static algorithm produces a lot of false positives. The following code illustrates this effect:

```
class Delegates {
  X delegate;
  // will be detected as addListener
  void set( X x ) { delegate = x; }
  // will be detected as notify
  void internalAction() { delegate.provideFunctionality(); }
}
```

A $1:1$ relation is suspicious, but need not be a mismatch, since this may be a valid configuration of the Observer Pattern. In case the set method of Delegates objects is called multiple times followed by a call of internalAction, our algorithm detects internalAction's violations of the notify role. In our case, all 6 tuples in the **Full** $1:1$ **match** class were actually delegations.

The **May match** class contains no Observer instance. In over 70% of the tuples, either only the addListener or only the notify method was called, but these methods cannot provoke a protocol mismatch.

The Observer Pattern instance in category **No decision** has not been executed and, therefore, not classified. If we ensured by employing testing technology that every candidate method gets executed, we could classify all tuples and thus achieve an empty **No decision** set.

All detected mismatches were correct, i. e., these tuples did not represent an implementation of the Observer Pattern.

4. RELATED WORK

Other approaches to detect patterns mostly restrict themselves to static analyses using rather strong static signatures. These approaches fail to detect behavioral patterns as their static patterns are not distinctive enough, but their static analyses are nevertheless worth noting. We discuss some of these below.

The Pat system [13] detects structural design patterns by extracting design information from C++ header files and storing them as Prolog facts. Patterns are expressed as rules and searching is done by executing Prolog queries. The Goose system [6] gives a graphic visualization of C++ program structures using a similar approach for their detection. Additionally, it detects patterns indicating design problems.

[10] present static analyses to discover design patterns (Template Method, Factory Method and Bridge) from C++ systems. They identify the necessity for human insight into the problem domain of the software at hand, at least for detecting the Bridge pattern due to the large number of false positives.

[3] additionally uses dynamic information, analyzing the flow of messages. His approach is restricted to detecting design patterns in Smalltalk, since he only regards flows in VisualWorks for Smalltalk. He therefore annotates the Smalltalk runtime environment. Another drawback is, that his approach gathers type information only at periodic events.

[5] also employ code instrumentation to extract dynamic information to analyze and transform architectures. The presented approach only identifies communication primitives, but no complex protocols.

5. CONCLUSIONS AND FUTURE WORK

The present paper shows how to detect communication patterns in legacy systems. Therefore, we filtered static analysis information using dynamic analysis results. This approach improves the quality of the results tremendously as protocol conformance of a pattern can be checked. Moreover, we partitioned the candidate pattern instances into the categories Full match, May match, and Mismatch giving more differentiated information to the user.

By now, dynamic analyses cannot give any clue if the candidate is not executed. We will avoid this using results from testing theory. Another direction of future work is the framework extension to support more patterns and antipatterns [4], as well. Since implementing static and dynamic algorithms by hand is a costly concern, we need to develop a tool generating analysis programs from pattern specifications. Finally, we strive to improve visualization of detected patterns to increase user support for understanding large scale software systems.

6. REFERENCES

[1] Uwe Aßmann, Dirk Heuzeroth, Andreas Ludwig, and R. Neumann. Recoder. http://recoder.sourceforge.net, 2001.

[2] Len Bass, Paul Clement, and Rick Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[3] K. Brown. Design reverse-engineering and automated design pattern detection in smalltalk, 1997.

[4] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley, New York, NY, 1998.

[5] S. J. Carriere, S. G. Woods, and R. Kazman. Software Architectural Transformation. In *Proceedings of WCRE 99*, October 1999.

[6] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Technology of Object-Oriented Languages and Systems - TOOLS 30*, pages 18–32. IEEE Computer Society, 1999.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[8] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann. Aspect-Oriented Configuration and Adaptation of Component Communication. In Jan Bosch, editor, *Generative and Component-Based Software-Engineering, Third International Conference, GCSE 2001*, number 2186 in LNCS. Springer, September 2001.

[9] Java platform debugger architecture, http://java.sun.com/products/jpda/.

[10] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-based reverse-engineering of design components. In *International Conference on Software Engineering*, pages 226–235, 1999.

[11] Welf Löwe. VizzEditor, VizzScheduler, and VizzAnalyzer. http://i44pc29.info.uni-karlsruhe.de/VizzWeb, 2001.

[12] Welf Löwe, Andreas Ludwig, and Andreas Schwind. Understanding software – static and dynamic aspects. In *17th International Conference on Advanced Science and Technology – ICAST 2001*, pages 83–88, 2001.

[13] L. Prechelt and Ch. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *J.UCS: Journal of Universal Computer Science*, 4(12):866ff, 1998.

[14] M. Shaw and D. Graham. *Software Architecture in Practice – Perspectives on an Emerging Discipline*. Prentice Hall, 1996.