Automatic Design Pattern Detection

Dirk Heuzeroth Thomas Holl Gustav Högström Welf Löwe

University of Karlsruhe,Germany
IPD, Program Structures Group
{heuzer|holl}@ipd.info.uni-karlsruhe.de

University of Växjö, Sweden
MSI, Software Technology Group
Welf.Lowe@msi.vxu.se

Abstract

We detect design patterns in legacy code combining static and dynamic analyses. The analyses do not depend on coding or naming conventions. We classify potential pattern instances according to the evidence our analyses provide. We discuss our approach for the Observer, Composite, Mediator, Chain of Responsibility and Visitor Patterns. Our Java analysis tool analyzes Java programs. We evaluate our approach by applying the tool on itself and on the Java SwingSetExample using the Swing library.

1. Introduction

A major task in software comprehension is the understanding of its *design* and *architecture*. As software can get large, this task should be supported by tools performing automatic analyses.

Design, however, is hard to detect automatically, as it is not tangible. Fortunately, standard solutions to solve certain design problems have been established. These design patterns are described together with the design problem(s) they intend to solve [1]. Many of them have been formalized, for example to introduce them automatically into software [2]. If it were possible to detect these patterns in software systems, one would be able to deduce the intended design.

Components and their ports, connectors between ports, and the containment relation of components in larger components define a system's *architecture* [3]. Components are coherent classes or coherent smaller components (our notion of components is recursive). Connectors and ports are implemented using basic communication constructs like calls, RPCs, RMIs, input output routines etc. provided by the implementation language or the component system. 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 [4].

In order to understand architecture, we would prefer to view a system consisting of components, abstract ports and connectors. However, legacy (source) code only contains classes, with port and connector implementations scattered throughout their code.

Fortunately, connectors are often implemented with design patterns, e.g. the Observer Pattern. Other patterns indicate a high coherence of classes, e.g., the Composite Pattern. If it were possible to detect these special patterns in software systems, it would be easier to deduce the system's architecture.

Design patterns usually have static and dynamic aspects. A pattern specifies structural connections among the classes via call, delegation or inheritance relations (static information). Additionally, it requires some specific sequences of actions and interactions of the objects of these classes (dynamic information).

Hence, we propose to analyze the structure and the behavior of the system with respect to the patterns indicating the instance of a certain design pattern. This is necessary, since there are situations, where neither static nor dynamic analyses alone are sufficient (or not with acceptable expenses). For example it is statically not 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 among objects are dynamic by nature.

We organize the paper in the following way: Section 2 describes our approach in general. Section 3 describes the design pattern detection with the Observer Patterns as a running example in detail. Section 4 discusses similar approaches for the other patterns. Section 5 evaluates our results in practical experiments. Section 6 sketches an approach of a general generator of pattern analyses based on our experiences with the patterns discussed so far.

Section 7 compares our result to related works. Finally, Section 8 concludes the results and points to directions of future work.

2. General Approach

A *design pattern instance* is defined by a tuple of program elements such as classes, methods or attributes conforming to the *restrictions* or *rules* of a certain design pattern. An example of such a tuple for an instance of an Observer Pattern is:
(`Subject.addListener`, `Subject.removeListener`, `Subject.notify`, `Listener.update`)

We distinguish the *role* of a class in a pattern from the program element itself. Observer and Observable, e.g., are roles of classes in the Observer Pattern. `SomeObserverClass` and `AnObservableClass` can be two concrete classes in the two roles. Moreover, we even distinguish an *object instance* from a design pattern instance, if the latter contains class program elements. As an example consider, that we could have different object instances of `SomeObserverClass` each with several object instances of `AnObservableClass`, or, e.g. `AnotherObservable` attached to it.

Our analyses distinguish between static and dynamic pattern *restrictions* or *rules*. The former restrict the code structure the latter the runtime behavior. Analyzing with the static restrictions results in a set of candidate occurrences 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 execute the program under investigation and monitor the executions of the candidate instances found by the static analysis with respect to the dynamic restrictions. Figure 1 illustrates our approach.

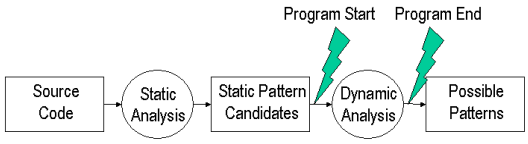

Figure 1. Static and Dynamic Analyses of Patterns.

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

For the understanding of a system's architecture, intra-component communications patterns are less important. We thus have the option to perform component pattern analysis first and filter the candidates for communication patterns inside components, leading to a tremendous reduction of communication pattern candidates. However, as our analyses are fast enough, we do not do so. Instead, we use the information on high-level connectors in the code as an additional indication for the correct partitioning of the legacy system into its components. The hypothesis behind this is that components have an elaborated interaction interface to other components while intra-component interactions are often implemented ad hoc. Hence, we expect many sources and targets of communication patterns to reside in different components when choosing the right partitioning into components while many of the essential communications become intra-component for a partitioning not intended.

Unfortunately, the analyses are not unique. Hence, the process of pattern and architecture detection is an iterative, interactive process where analyses and visualizations complement on another. However, visualizations are not focused in this paper, we have discussed them in more detail in another paper [5].

The static analyses are done with Recoder [6], a tool to read, analyze and manipulate Java programs. In principle, Recoder is a compiler front end. It reads the source code and constructs the abstract syntax tree (AST). Then it performs the static semantic analyses where it resolves names and types. In addition to the semantic analyses required by the programming language, it can compute additional relations over the syntactical and semantic elements, e.g. define-use-relations. It has a programmer interface to add new, user-defined analyses for the computation of arbitrary relations. In contrast to standard compilers, Recoder provides all these entities and relations as an API. In order to analyze the structure of a program, we simply write a program accessing this interface. The elements of candidates of pattern instances retrieved by the static analysis are nodes in the AST accessible via Recoder. Examples of such elements are classes and methods that potentially play different roles in a pattern.

These program parts are then instrumented using the code manipulation interface of Recoder. We simply add appropriate event generators tracing the dynamic execution of the candidates and providing runtime information. Dynamic analyses are the corresponding event handlers, checking that the protocol of the candidate execution conforms to the expected dynamic pattern restrictions.

Although, currently Recoder investigates Java sources only, the approach and the architecture can be applied to sources written in any typed language.

3. Pattern Detection

In this section, we present our approach to detect design patterns by combining static and dynamic analyses [7]. The static analysis computes potential program parts playing a certain role in a design 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. Subsection 3.1 discusses the static, Subsection 3.2 the dynamic analysis.

In the subsequent subsections, we use the Observer Pattern (event notification) [1] as a running example for a special architectural pattern.

According to its static rules, we consider every setting where an object allows registering and optionally deregistering other objects, and potentially notifies all registered objects by calling their update method as an instance of the Observer Pattern. Static analyses compute a set of classes that fulfill the necessary properties for subject and corresponding listener classes. Dynamic analyses then monitor objects of these classes during execution and check whether the interaction among them satisfies the dynamic Observer rules.

The following naming conventions refer to roles of certain methods of the Observer Pattern. Note, that this naming convention is only used for explanations – the static analysis does not refer to these names.

`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.

3.1. 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 relation *P* on the AST nodes and provides the result as a set of *Candidates*, i.e., a set of tuples of AST nodes with the appropriate static structure. This set is a conservative approximation to the actual patterns in the code.

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 conforming to the static rules is large. Brute force methods, as 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 `m` a potential listener, provided `p` is neither identical to nor a super or a subclass of `c` ($p \not\subset c \wedge c \not\subset p \wedge c \neq p$). Such a relation would contradict the decoupling of subject and listeners as required in the Observer Pattern rules. We then 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 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`), which is satisfied 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|removeListener`, `S.notify`, `L.update`). To compute the final set of candidates, we iterate over the tuples of *Y*.

We combine corresponding `addListener` and `removeListener` methods into one pattern candidate. If the `S.addListener|removeListener` entry of *Y* satisfies the `addListener` predicate defined below, we combine it with all other tuples in *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`), which 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 side 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.

This results in the following static analysis algorithm that computes the static candidates *C*:

```
C := ∅
for each class c do
  Y := ∅ // intermediate result
  for each method m in c do
     for each parameter type p in m
     where p⊄c ∧ c⊄p ∧ c≠p do
       for each call from c.n to p.u do
          if isNotifyListener(c.n, p.u)
            Y := Y ∪ {(c.m, c.n, p.u)}

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

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: expert knowledge like coding and naming conventions, dynamic analysis, or static data flow analysis. We choose dynamic analyses.

3.2. Dynamic Analysis

The static analysis provided tuples of AST nodes in the candidate set *C*. The dynamic analysis takes this set *C* 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 rules. In case of a rule violation, the candidate is rejected.

Each element of a candidate tuple is contained in a class definition or is a class definition itself. At runtime we might have many object instances of these classes and each should conform to the dynamic pattern. Additionally, patterns require *n:m*, *1:n*, or *1:1* relations among objects of the implementing classes. For each individual candidate tuple, the number of instance objects of their classes could be restricted. The Observer Pattern, e.g., requires a *1:n* relation of the subject instances and their listener instances.

Moreover, we might have more than one instance of the Observer Pattern defined by the different subject and listener classes.

Altogether, we trace a set of instances for each candidate tuple of a pattern. Each may contain several objects per position in the tuple. Considering our Observer Pattern, we thus assign to every candidate tuple (`S.addListener`, `S.removeListener`, `S.notify`, `L.update`), cf. Section 3.1, a set of instance tuples (`s.addListener`, `s.removeListener`, `s.notify`, {l_1`.update` … l_n`.update`}) where `s` is an instance (object) of class `S` and l_1 … 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 always the same and already captured in the candidate tuple. To avoid redundancies, we only associate a set of tuples (`s`, { l_1 … l_n }).

We monitor each node in a tuple of the candidates. Whenever we dynamically execute such a monitored node, we retrieve the entire candidate tuples in which the node is contained. 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 depend on the role of the method:

`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 turning 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.

4. Further Patterns

So far, we only discussed the detection of just one pattern, the Observer Pattern. In the same way, as we detected this pattern, we also implemented static and dynamic analyses to detect the Composite, Mediator, Chain of Responsibility and Visitor Patterns [1].

4.1. Composite Pattern

4.1.1 The Static Analysis algorithm is given below:

```
C := ∅
for each class c do
 Y := ∅ // intermediate result
 for each method m in c do
  for each parameter type p in m
  where c ⊂ p do
     for each call from c.o to p.o do
        if isCompositeOperation(c.o, p.o)
           Y := Y ∪ {(c.m, c.o, p.o)}

for each (c1.a,c1.o,p1.o) ∈ Y do
 for each (c2.a,c.o,p2.o) ∈ Y do
  if (isAddComponent(c1.a))
     if (c1.o = c2.o ∧ p1.o = p2.o)
        if (c.a1 = c.a2)
           C := C ∪ (c.a1,null,c.n1,p1.u1)
        else
           C := C ∪ (c.a1,c.a2,c.n1,p1.u1)
```

The predicate *isAddComponent* exactly corresponds to the *isAddListener* predicate and the *isComposite-Operation* predicate exactly corresponds to the *isNotify-Listeners* predicate. The main difference between the two patterns is the intended decoupling of classes: The Observer Pattern requires the Observers and Observable to be unrelated; the Composite Pattern requires Component and Composite to satify a subtype relation.

It is obvious, that this algorithm is similar to the static analysis for the Observer Pattern and was therefore easy to encode.

4.1.2 The Dynamic Analysis was even easier to port, since both patterns do not differ in their protocol. Hence, we applied the dynamic analysis algorithm of the Observer Pattern detection unmodified.

4.2. Mediator Pattern

A Mediator pattern consists of at least two classes both with a reference to each other. In general, a central Mediator has arbitrary many users. The Mediator refers to every User. Users in turn are connected back to their Mediator. Usually, the User is attached to the Mediator at construction. Later attachment by a method call is possible, as well.

4.2.1 The Static Analysis checks for every class (potential Mediator) if it has variables pointing to a User knowing that Users have a variable of the Mediator's class or its superclass. The User must have the Mediator class as an argument in its constructor or in at least one other method. It is not possible that the Mediator and its Users are of same type. As it does not make sense to implement a Mediator pattern with only one User, we reject candidates with only one possible user variable in the Mediator:

```
C := ∅
for each class c do
 Y := ∅ // intermediate result
 for each variable v in c do
  for each class u := class(v) do
     for each variable v' in u do
        if c=class(v') ∨ c ⊂ class(v')
           Y := Y ∪ {(c,v,u,v')}

for each (c,v,u,v')∈ Y do
 for each method m in u do
  if "v':= …" in body(m)
     for each parameter p in m do
        if c=class(p) ∨ c ⊂ class(p)
           C := C ∪ {(c,v,u,v',m)}
```

To avoid redundant storing, $\{(c, v_1, u_1, v'_1, m_1), (c, v_2, u_2, v'_2, m_2), \ldots\}$ is reduced to $\{(c, \{v_1, v_2, \ldots\}, \{u_1, u_2, \ldots\}, \{v'_1, v'_2, \ldots\}, \{m_1, m_2, \ldots\}), \ldots\}$.

4.2.2 The Dynamic Analysis distinguishes different Mediator objects for a Mediator candidate class `M` and different objects for all its User classes $U_1 \ldots U_n$. Again, the variables pointing to the User and the method receiving the Mediator reference are stored in the candidate and need not to be captured in the dynamic tuple of objects (`mObject`, $\{u_1 \ldots u_n\}$).

Whenever the method `m` handing over the Mediator *object* to the User *object* is invoked, it is checked whether it send a self-reference to the mediator *object*. Whenever, the body of that method is executed, it is checked whether this object reference is actually stored in the Mediator typed variable `v'` in the User.

4.3. Chain of Responsibility Pattern

A Chain of Responsibility consists of a list of classes, each delegating to the next in the list and inheriting from a same base class. All classes in the chain implement and (potentially) invoke a Handler method defined in the superclass. A call to the Handler method is (potentially) delegated along the chain.

4.3.1 The Static Analysis looks for any (potential Handler) class delegating to itself or one of its superclasses. Moreover, it checks if that superclass implements the potential Handler method invoked in the Handler, as well:

```
C := ∅
for each class c do
 Y := ∅ // intermediate result
 for each variable v in c do
  for each class u := class(v) do
     if c=u ∨ c ⊂ u
        Y := Y ∪ {(c,v,u)}

for each (c,v,u)∈ Y do
 for each method m in u do
  if "… v.m …" in body(m) in c
     C := C ∪ {(c,v,u,m)}
```

To avoid redundant storing, $\{(c_1,v_1,u_1,m),(c_2,v_2,u_2,m),\ldots\}$ is reduced to $\{(\{c_1,c_2,\ldots\},\{v_1,v_2,\ldots\},\{u_1,u_2,\ldots\},m),\ \ldots\}$.

4.3.2 The Dynamic Analysis checks how the chain is built up. An implementation of this pattern usually declares the variables that constitute the chain of a supertype of the actual chain classes. The dynamic analysis detects the actual classes that are linked. Afterwards one can decide by screening the classes, which links constitute a Chain of Responsibility.

Note, that the Handler methods need not to be called completely. While a sequence of calls to the Handler methods would support the candidate being a Chain of Responsibility instance, it is hard to define a protocol violation.

4.4. Visitor Pattern

Visitors of an object structure of some Element classes are invoked via some Visit method defined in Visitor classes. They call back some operations of the Elements to visit their state. Visitors inherit from a common superclass. A concrete Visitor is attached to an Element by an Accept method triggering the Visit method.

4.4.1 The Static Analysis starts to look for Elements first. In such a class, we expect an Accept method with a visitor class as a parameter. That parameter will be called (visited) with the Element class as an argument, which calls back the Element. In general the parameter of the Accept method is declared of the Visitor's superclass.

```
C := ∅
for each class c do //element?
 Y := ∅ // intermediate result
 for each method a in c do //accept?
  for each parameter p in a do
     for each c'=class(p) do  //visitor?
        for each method v in c' do //visit?
           if ("… x.v(y) …" in body(a) ∧
                c'=class(x) ∧ c=class(y) )
              Y := Y ∪ {(c,a,c',v)}

for each (c,a,c',v)∈ Y do
 for each method o in c do //call back?
  if "… z.o() …" in body(v) ∧ c=class(z)
     C := C ∪ {(c,a,c',v)}
```

To avoid redundant storing, $\{(c,a,c'_1,v),(c,a,c'_2,v),\ldots\}$ is stored as $\{(c,a,\{c'_1,c'_2,\ldots\},v),\ \ldots\}$.

4.4.2 The Dynamic Analysis basically tests for object identities. It checks if the (potential) Visitor *object* given with a (potential) Accept method is actually the same *object* the (potential) Visit method is invoked on. Moreover, it checks if the parameter given with this Visit method is actually the (potential) Element *object*. Finally, it checks whether this element object is actually called back from the Visitor object.

5. Evaluation

A problem that automatically occurs in this type of analyses is how to verify the results experimentally. Ideally, one would compare the analysis results with pattern documentation of the analyzed software. Unfortunately, no real world software is that well documented. We cannot assume that patterns are documented (otherwise we would have easier ways to understand software systems). The analyzer can detect patterns and by-hand checking we can assure them as correct, but it is hard to estimate the number of undiscovered patterns, i.e. the true negatives.

Therefore one has to start the validation in systems that are known.

The GOF-book [1] gives example implementations for the described patterns. These implementations are implemented in C++, but it was easy to translate them to Java. Moreover, to make the evaluation more credible, we insertet some fault patterns similar to the correct patterns. The analysis tool did not detect any false positive patterns and it did detect all real patterns.

To continue the verifying process, we applied our tool to the source code of our tool itself including the Recoder packages. In this code the Observers were well documented. Hence we were able to count false positives and true negatives.

Then we applied it to the SwingSet2 example of the JDK 1.3.1 including the javax.swing.* packages. In order to determine the number of Observers in this unkown software, we trusted the strict naming conventions of SUN's library programmers. We firstly issued a

```
grep"add\w*Listener\w*{"
```

and checked the hits manually for correctness. To catch Observer Pattern instances violating the naming conventions, as well, we also screened the output of our tool and added the remaining occurrences. An example of an Observer Pattern instance violating the naming conventions is the `DefaultTableColumnModel` in the `javax.swing.table` package with methods `addColumn` in the `addListener`-role, `removeColumn` in the `removeListener`-role and `recalcWidthCache` in the `notify`-role. This procedure yields a fairly good approximation of the number of Observer Patterns really contained in the code.

Finally, we analyzed the other patterns. Unfortunately, these patterns have not been documented that well and no naming conventions applied. Hence, we only excluded the detection of false positives by screening the analysis results. An estimation of true negatives was not possible.

5.1. Observers in our Analyzer Tool

Statistics about our analysis tool including the Recoder package are given in Table 1.

Table 1. Statistics on our tool including the Recoder.

	Classes	Methods	Observers[1]
Recoder	555	6734	2
VizzAnalyzer	43	214	3
Sum	598	6948	5

The main task of the static algorithm is to reduce the amount of candidates. In case of the Observer Pattern detection, the amount of possible candidates is equal to 9.7×10^{13} calculated by the formula:

$$\binom{6948}{4} + \binom{6948}{3}$$

[1] Subjects used neither delegation nor sub-classes of subjects.

The former term accounts for the 6948 methods in the 4 possible roles, the latter term models tuples with empty `removeListener` role. Our static analysis applies the *isAddListener* and *isNotifyListeners* predicates as main criteria to reduce this amount of candidates to 28030 tuples containing all 5 Observer Pattern occurrences. The corresponding analysis phase needs about 70 seconds on a Pentium III, 500Mhz, 256MB RAM, running Windows NT 4 with JDK 1.3.1.

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

Table 2. Results of dynamic analysis of our analysis tool.

	Full 1:n match	*Full 1:1 match*	*May match*	*No decision*	*Mis-match*
Detected	4	5	67	18638	9316
Real	4	0	0	1	0

The *Full 1:n match* category shows that our tool classified all Observer Pattern occurrences used in that program run 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 fragment illustrates this effect:

```
class Delegate {
   X delegate;

  //is detected as addListener
   void set(X x) { delegate =x;}

   //is detected as notify
   void internalAction(){delegate. ();}
}
```

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 `Delegate` objects is called many times followed by a call of `doSomething`, our algorithm detects `doSomething`'s violations of the `notify` role. In our case, all 5 tuples in the *Full 1:1 match* class were actually delegations.

The *May match* class contains no Observer occurrence. 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 tuple in category *No decision* has not been executed, and therefore not been 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.

5.2. Observer in the SwingSet2 Example

To evaluate the quality of our analyses in this case study, we first need to determine the number of Observer Pattern occurrences in the example code. Since SUN's developers obey naming conventions, we found quite a few pattern occurrences. To catch also Observer pattern occurrences violating the naming conventions, we screened the output of our tool and added the remaining occurrences. This procedure yields a fairly good approximation of the number of Observer patterns actually contained in the code and results in the statistics presented in Table 3.

Table 3. Statistics about the SwingSet2 example including javax.swing.* packages.

	Classes	Methods	Observers[2]
Swing	1357	11478	59
SwingSet2	124	403	0
Sum	1481	11881	59

The high number of 59 Observer Pattern occurrences in Table 3 results from our way of counting: Since we identify the particular method roles, a subject class containing several `notify` methods contributes to multiple Observer Pattern occurrences. We count every such combination as one separate occurrence.

In the same way as in the previous case study, the static analysis reduces the number of 8.3×10^{14} possible candidates to 200 738 tuples containing all 59 Observer Pattern occurrences. The corresponding analysis phase needs about 190 seconds on a Pentium III, 500Mhz, 256MB RAM, running Windows NT 4 with JDK 1.3.1.

Table 4. Results of dynamic analysis for the SwingSet2 example.

	Full 1:n match	*Full 1:1 match*	*May match*	*No decision*	*Mismatch*
Detected	13	805	36	85620	114264
Real	13	20	2	24	0

Table 4 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 the number of tuples of the corresponding category that represent real Observer occurrences.

The results show that all 13 Observer Pattern occurrences in the full 1:n match category have been classified correctly by our analyses. Besides the 20 real Observer Pattern occurrences the *Full 1:1 match* category again contains a lot of Delegation Pattern occurrences as in the Recoder example. The same holds for the *May match* category. Note that the *Mismatch* category again does not contain a real Observer Pattern occurrence, i.e., all detected mismatches were correct.

5.3. Further Patterns

To validate our analyses algorithms for the Mediator, Chain of Responsibility and Visitor Patterns, we applied them to our tool itself. We first discuss the results of the static analyses and then those of the dynamic analyses.

The results from the static analyses are each presented as a table of the detected candidate tuples.

The static analysis of the Mediator Pattern detected 298 possible tuples. The high number of candidates can be misleading, but one pattern instance may lead to a set of candidate tuples, because we consider all possible combinations of roles.

The static analysis of the Chain of Responsibility Pattern provided 349 tuples. The tuples represent elements of the chain linked together and not a whole chain because a static analysis cannot conclude how the instances of the classes are connected together. Only a bunch of different elements probably linked together somehow are detected statically. In some cases, there were just only one possible class that the chain is constituted from. However, at runtime a long chain of objects of that instance are observed. The high number of tuples is somewhat misleading: The highest number of different classes that potentially constitute a Chain of Responsibility was seventeen; all classes issued a call to the same method of a common super class. The analysis has not included a detection of these superclass calls. This is not a restriction, as they will be detected anyway during the analysis of the superclass.

When used in visualizations, the analyzed Recoder application visits all AST nodes and presents them in some way. In this setting, we detected the `PrettyPrinter` class as a Visitor. The program has implemented a subclass of the `PrettyPrinter` and the analysis detected 1701 different pairs of instances that can constitute the whole Visitor pattern. That also included some possible subclasses. The high number of detected elements can be seen as very misleading, most of the detected pairs were connected to the same visitor and they had the same Accept methods. This fact can also be

[2] Subjects using neither delegation nor sub-classes of subjects.

interpreted as the static analysis detected only a few Visitor pattern instances but those patterns are very large.

Table 5. Result after the static analysis. The numbers show the statically detected candidate tuples per pattern.

Mediator	Chain of Responsibility	Visitor
298	349	1701

The dynamic analysis for the Mediator pattern sorted out most candidates. After an execution of the test program, the set of patterns was reduced; they were either mismatches or the actual methods were not executed in the program run. The 18 matched tuples did not contain any false positives. By manual screening we have grouped them into six pattern instances.

Table 6. Result of the dynamic analysis of Mediator.

Static Candidates	*Match*	*Mis-match*	*No Decision*
298	18	60	219

The dynamic analysis of the Chain of Responsibility pattern rejected almost all candidates. An explanation can be that all possible static chains were not executed or the method calls were connected to a different object that was not a part of the potential chain. Manual screening revealed that the 24 matched tuples did not contain any false positives and are connected to each other in one chain.

Table 7. Result of the dynamic analysis of Chain of Responsibility.

Static Candidates	*Match*	*Mis-match*	*No Decision*
349	24	158	167

The dynamic check of the Visitor concluded that 349 of the static candidates really are parts of actual patterns. No candidates visited were rejected. This indicates that the static Visitor pattern is already pretty distinctive. By manual screening we concluded that these 349 dynamic candidates together represent two instances of Visitor patterns. No false positives were detected.

Table 8. Result of the dynamic analysis of Visitor.

Static Candidates	*Match*	*Mis-match*	*No Decision*
1701	349	0	1352

Altogether, we obtained the following results from our analysis including the manual screening:

Table 9. Result after the dynamic check and manual screening per pattern.

Mediator	Chain of responsibility	Visitor
6	1	2

After having performed the above analyses of our analysis program, we carried out an expanded analysis with other test programs that also used the Recoder library. We hoped that they would execute more candidates from the Recoder library that had been put in the *No Decisionc* class in the first try. Unfortunately, these test programs did not generate more information, because there have still been some candidates were not executed dynamically.

However, we have a clear indication that we should to improve the dynamic analyses: If a pattern candidate occurrence is not executed during a concrete program run, our dynamic analyses cannot provide any evidence for its conformance to or its violation of the pattern rules. We will avoid this problem using results from testing theory. These ensure that every point of programs, if reachable at all, gets executed at least once.

However, once a candidate is visited at in the dynamic analyses, the dynamic pattern rules are distinctive enough to reject false positives.

6. Generation of Analyzers

Hand-coded pattern analyses are the solution for the standard patterns. Non-standard patterns that need to be detected could be added easily by modifying the pattern recognition as demonstrated with the patterns above – to see that, compare the different analysis algorithms: they are quite similar. However, in order to be complete and general and to avoid such coding, the analyses should be generated from a pattern specification.

Our generator accepts a specification language, which is based on predicate calculus for static constraints and defines pre- and post-constraints to check the dynamic behavior of candidate methods. The language design allows specifying arbitrary interaction patterns.

The generation of the static analysis uses the search algorithm already applied in our static analysis algorithms for the Observer and the Composite Patterns as template. This means, we iterate over the abstract syntax tree of the program to be analyzed and check the static constraints (predicates) specified for the pattern to be detected. When we are looking for a relation between classes, e.g., the specification contains a predicate concerning class entities. The generator produces a loop over all classes of the program to be analyzed and inserts check code to check the specified relation. This translates to further loops depending on the nesting of the entities contained in the specified relation and/or if-commands. To query the abstract syntax, we use our Recoder library.

The generation of the dynamic analysis is much harder, since we currently do not know if one generic data structure is sufficiently suited to simulate all possible protocols. This is why our specification for the dynamic

part of a pattern requires defining the data structure for the simulator and the actions to perform. It thus corresponds exactly to giving an implementation for the dynamic analysis, i.e., to the event handlers. However, events are generated automatically. The code is instrumented automatically at the statically recognized candidates, usually method entries and exits and hands over parameters. These predefined hooks for the event listeners checking the dynamic pattern simplify the task significantly.

At the moment, we are already able to generate the presented analyses to detect the Observer Pattern. To validate that our specification language and the generator are general enough, we currently try to detect the Mediator Pattern as well. The problem is to keep the language small and simple enough so that pattern specifications are at the right level of abstraction and more appropriate than directly coding the analysis algorithms.

7. Related Work

Quite a bit of work has already been done in the field of automatic pattern detection.

Keller et al. [8] describe a static analysis to discover design patterns (Template Method, Factory Method and Bridge) from C++ systems. The authors 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.

The Pat system [9] 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.

Brown [10] 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 he only gathers type information at periodic events.

Carriere et al. [11] also employ code instrumentation to extract dynamic information to analyze and transform architectures. The presented approach only identifies communication primitives, but no complex protocols.

The present paper extends our previous results [7] in two ways. Firstly, it implements more than the Observer Pattern analyzer and extends the experiments to unknown code. These extensions show that the results (and shortcomings in the dynamic analysis) can be generalized. Secondly, it sketches our approach to and first results of automatic generation of analyses.

8. Conclusion

We presented an approach to support the understanding of software systems by detecting design patterns automatically. We use static and dynamic analyses. More specifically, we filtered information gained by static pattern detection on the code using the observations of dynamic code execution. This approach improves the quality of the results tremendously as protocol conformance of a pattern can be checked. We argued that neither static nor dynamic analyses by themselves provide an adequate approach to find patterns in software systems. The number of false positives is small, in most experiments even zero.

However, if pattern candidates are not executed during the dynamic analyses we cannot provide any evidence for its conformance to or its violation of the protocol. Hence the number of true negatives could be large. Currently, we try to apply results from test theory to ensure that every (reachable) candidate gets eventually executed. To further reduce the true negatives, we plan to integrate data flow analyses and checking of naming conventions into our static analyses.

9. References

[1] Gamma, E; R. Helm, R. Johnson, and J. Vlissides (1995), “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional Computing Series, Addison-Wesley Publishing Company, New York, NY.

[2] Genssler, T.; B. Mohr, B. Schulz and W. Zimmer (1998), “On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems”. In Proc. 27th TOOLS.

[3] D. Garlan, M. Shaw (1993), “An Introduction to Software Architecture”, In Advances in Software Engineering and Knowledge Engineering, vol. 1, World Scientific Publishing Company, Singapore, pp. 1-40.

[4] Heuzeroth, D., W. Löwe, A. Ludwig, and U. Aßmann (2001), “Aspect-Oriented Configuration and Adaptation of Component Communication”, In 3rd Int. Conf. GCSE, Springer, LNCS 2186, p. 58 ff.

[5] Heuzeroth, D. and W. Löwe (2003), “Understanding Architecture Through Structure and Behavior Visualization” In: Kang Zhang (Ed.). “Software Visualization - From Theory to Practice”, Kluwer Academic Publishers.

[6] Ludwig, A., R. Neumann, U. Aßmann, and D. Heuzeroth (2001), “RECODER Homepage”, http://recoder.sf.net.

[7] Heuzeroth, D., T. Holl, and W. Löwe (2002), “Combining Static and Dynamic Analyses to Detect Interaction Patterns”, In Proc. 6th Int. Conf. IDPT.

[8] Keller, R. K., R. Schauer, S. Robitaille, and P. Page (1999), “Pattern-Based Reverse-Engineering of Design Components”, In Proc. ISCE, pp. 226-235.

[9] Prechelt, L. and C. Krämer (1998), “Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns”, J.UCS: 4, 12, 866ff.

[10] Brown, K. (1997), “Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk”, Master Thesis, Univserity of Illinois at Urbana-Champaign

[11] Carriere, S. J., S. G. Woods, and R. Kazman (1999), “Software Architectural Transformation”, In Proc. 6th WCRE.