Pattern-Based Design Recovery of Java Software

Jochen Seemann and Jürgen Wolff von Gudenberg
Lehrstuhl für Informatik II
Universität Würzburg
Am Hubland, 97074 Würzburg, Germany
{seemann,wolff}@informatik.uni-wuerzburg.de

Abstract

In this paper we show how to recover design information from Java source code. We take a pattern-based approach and proceed in a step by step manner deriving several layers of increasing abstraction. A compiler collects information about inheritance hierarchies and method call relations. It also looks for particular source text patterns coming from naming conventions or programming guidelines.

The result of the compile phase is a graph acting as the starting graph of a graph grammar that describes our design recovery process. We define criteria for the automatic detection of associations and aggregations between classes, as well as for some of the popular design patterns such as composite or strategy.

1 Introduction

Java is commonly used for the development of new software systems. Many large systems consisting of numerous comprehensive packages are completely written in Java. Since Java is a new language, the software is well designed in general and quite well documented by instrumentation with specific comments and illustrated by the Javadoc tool. This tool, however, only produces a list of classes with their attributes and methods although well presented as a hypertext document. For large software systems a graphical overview may be preferred with emphasis on structure and relationships between classes, see [2] for C++ software. Even if CASE tools with roundtrip engineering facilities are used, changes of the source code often are traced back to the design diagrams in a simplistic manner. An attribute establishing a new association is registered, but the association is not.

In this paper we present some ideas on how to recover design information from pure Java source code. We take a pattern-based approach similar to [3] and proceed in a step by step manner deriving several layers of increasing abstraction.

A compiler that collects the usual information about inheritance hierarchies and method call relations may also look for particular source text patterns coming from naming conventions or programming guidelines. In our approach, the detection of predefined names that identify containers, for example `Vector` or `HashTable`, is used to determine the multiplicity of an association.

The result of the compile phase is a graph whose construction is described in section 2. This graph acts as the starting graph of a graph grammar and is subsequently transformed by the grammar productions. The productions in the graph grammar describe our design recovery process. Note that for every software project new patterns may have to be defined, so there is not one single Java design recovery graph grammar.

In this paper we illustrate our approach by detecting associations and aggregations between classes, as well as some of the popular design patterns like composite or strategy. For this purpose the starting graph contains information concerning interfaces, classes, and methods.

The first transformation generalizes method relations to those of containing classes (see section 3). In section 4 we derive higher order relations between classes like association or delegation. Section 5 then introduces criteria to detect some of the well known design patterns and shows their applicability for the Java AWT package. Our notation follows the Unified Modeling Language UML [10]. There is only one pass through the source code, each subsequent phase relies on the results of the previous phase but may also retrieve information from deeper levels down to the original data base. In order to prepare the transformations we always start matching patterns in the most abstract graph, and hence decrease our search space for the detection of more detailed patterns.

2 Construction of the Data Base

For our purpose we need information about the class and interface hierarchies, about attributes and methods of classes and relatively detailed statements about the call of methods. In particular we parse the source code, look for characteristic patterns and gather the following relations. In these descriptions we use the common prefix notation in the text and infix in the formal definitions.

- class *extends* class
- interface *extends* interface

- class *implements* interface

  We build up the two inheritance structures for classes and interfaces and their interrelation.

- class *references* class
- class *references* interface

  The term *references* (class, class) means that the class has an attribute of the type of the second class. This relationship may be an association or aggregation [10]. To distinguish between these two relations, we register, if the containing class initializes a new instance of the attribute. This is, however, not treated as an additional relation, but as a specific kind of the *references* relation. We are not interested in attributes of primitive types. In fact, one of our generalization steps will be to disregard pure data classes that behave like primitive types. Note that in Java the second class may also be an interface. Hence there is also a relation *references* (class, interface). Again, we do not collect all attributes of a class, but we always interpret an attribute of class type as a candidate for an association to the corresponding class.

- class *owns* method

  In contrast to attributes the containment of methods is helpful information, as we will see later.

- method *calls* method

  The *calls* (method, method) relation may be split to more detailed relations that are helpful in several contexts. In particular, we differentiate between calls invoked for:

this	the same object
super	its ancestor
attrib	an attribute
local	a local variable of a method
param	a method parameter
result	a method result
static	call of a static class method

  These kinds may be used to recover associations more exactly.

  Calls of methods for the same object are very common and therefore do not contribute much information. The same holds for calls to the ancestor, that indicate that an inherited method is redefined. Such calls may be discarded, therefore.

  In Java the call of a static method often substitutes a globally available function. This holds in particular for standard classes of the language like `System` or `Math`. So these calls are skipped.

  Some programmers prefer to localize calls of methods via attributes i.e. they declare a local object in the method and initialize it with the attribute of the class. Often this is accompanied by a downcast of the attribute. We, therefore, recommend an even finer investigation of the *calls (local)* relation and collect more information from the initialization.

  We further register how many different objects or methods are called inside the body of the method. This information may be used to estimate the cohesion and coupling and helps for the detection of some design patterns (see 4.2).

- class *downcasts* class

  The most surprising relation may be the downcast of a class to one of its heirs. This is a specific Java property indicating a narrowing of the type of an attribute. Downcasts are quite common in Java software.

  Casts between the primitive types do not deliver any information important for the design recovery and hence are removed. Downcasts from the source type `Object` indicate the actual element type of containers, all others may help to further investigate the *calls* relation.

- method *creates* class

  This means that the method creates an instance of the class by invoking its constructor.

Multiplicity is indicated by array type attributes or by attributes like Vector or Hashtable. For these containers we determine the most general element type.

Altogether we consider a graph with 3 different node types and 8 different edge types.

$S = (V, E, \phi, \eta)$ with
$V =$ CLASS $\cup$ INTERFACE $\cup$ METHOD,
$E \subseteq$ CLASS $\times$ CLASS $\cup$ INTERFACE $\times$ INTERFACE $\cup$ CLASS $\times$ INTERFACE $\cup$ CLASS $\times$ METHOD $\cup$ METHOD $\times$ METHOD $\cup$ METHOD $\times$ CLASS,
$\phi : V \rightarrow \text{TEXT} \times \text{TYPE}^n$
$\eta : E \rightarrow \text{TEXT} \times \text{TEXT} \times \text{MULTIPLICITY}$

where $\phi(c)$ denotes the name for a class or interface and $\phi(m)$ describes the name, return type and parameter types for a method with $n - 1$ parameters.

The set of edges E is subset of the union of the 6 cartesian products described in the bullet list above. All edges are directed. There may exist more than one edge between the same nodes. Loops may also occur in the graph.

η is the edge labeling consisting of three parts where the latter two are optional. The first component of the edge label specifies the edge type, *owns*, *calls*, e.g. For edges of type *calls* the second label denotes the specific kind, *local*, *attrib*, e.g. The third edge label denotes the multiplicity of the relation. Currently only *one* and *many* are distinguished.

3 Filtering the Graph

3.1 Complete Subgraphs

Now this very comprehensive graph has to be filtered to obtain a clearly structured overview of the underlying software. One way is to separate the graph according to the different edge types. This leads to the inheritance hierarchies of classes and interfaces. Formally this is a subgraph construction.

The other edge types describe more local information and thus their full subgraphs do not present valuable information.

3.2 Unification of Classes and Interfaces

The difference between classes and interfaces helps to specify the structure of design patterns more clearly. Hence, we do not recommend their unification, although the graph becomes simpler.

3.3 Method Call Graphs

The method call graph may be simplified by neglecting the labels and then a transformation to a relation *calls*(class, class).

m_1 *calls* $m_2 \wedge c_1$ *owns* $m_1 \wedge c_2$ *owns* $m_2 \Rightarrow c_1$ *calls* c_2

$(m_1, m_2 : \textit{methods},\ c_1, c_2 : \textit{classes})$

We now may additionally drop the recursive calls to the class itself.

A further simplification could be obtained, if we look into the *extends* and *implements* relations and only keep the call to the most general class or interface.

Based on the original data base, a subset of the *calls* (method, method) relation can be obtained by defining the *uses* relation.

A method m_1 *uses* another method m_2, if
$e = m_1 \textit{ calls } m_2 \in E \wedge \eta(e)_2 = \textit{this}$
i.e. calls it by the same object.

3.4 Creation

The *creates* relation can be generalized to a class-class relation.

$\exists m_1 : m_1 \textit{ creates } c_2 \wedge c_1 \textit{ owns } m_1 \Rightarrow c_1 \textit{ creates } c_2$

$(m_1 : \textit{method},\ c_1, c_2 : \textit{classes})$

This creation occurs in the constructor or in another method. The initialization of an attribute is another source of the *creates*(class, class) relation.

$c_1 \textit{ references(init) } c_2 \Rightarrow c_1 \textit{ creates } c_2$

4 Association, Aggregation and Delegation

4.1 Association and Aggregation

The intersection of the *calls*(class, class) relation with the *references* relation yields the association relation.

$$c1 \textit{ assoc } c2 \iff c1 \textit{ calls } c2 \wedge c1 \textit{ references } c2$$

We do not consider associations coming from temporary calls via method result and local values. We also disregard those based on method parameters. For a less stringent viewpoint, those associations may also be taken into account.

An association is an aggregation, if the referenced object is created by the class, hence the intersection of the *assoc* and the *creates* relation is computed.

$$c1 \textit{ aggreg } c2 \iff c1 \textit{ assoc } c2 \wedge c1 \textit{ creates } c2$$

The multiplicity of an association can be obtained from the type of the attribute. If more than one attribute of the same type are referenced, this can mean that there is one *one−to−many* association or several different links between the two classes. We decide for different *one−to−one* associations, if the intersection of the set of methods called for each attribute is empty, otherwise we assume a *one − to − many* relationship.

Note that all our associations are unidirectional. A bidirectional link certainly has to be created, if the methods responsible for each direction call each other. For a less detailed view of the model, however, it helps to unify all different associations between two classes.

4.2 Delegation

Delegation means the shift of the responsibility of a method to another object that frequently occurs in the "programming by contract" style.

m_1 *delegates* m_2, if

$e = m_1 \textit{ calls } m_2 \in E \wedge \eta(e)_2 \neq \textit{this}$
$\wedge\ \neg\exists\, m_3 \neq m_2 \in E : m_1 \textit{ calls } m_3$
or $\forall m \in E : m_1 \textit{ calls } m \Rightarrow \phi(m_1)_1 = \phi(m)_1$.

On the class level, a delegation is a stronger relation than an association:

$c_1 \textit{ assoc } c_2 \wedge \exists m_i \in c_i : m_1 \textit{ delegates } m_2 \Rightarrow c_1 \textit{ delegates } c_2$.

A delegation is assumed, if only one method is called or if all called methods have the name of the calling method (given as the first component of the method label). However, this may be too restrictive in another context.

4.3 Result

In this section we have shown how the most important relations in object-oriented design methods, the association and aggregation, can be derived from the source code by the combination of simpler, easy to find patterns. For the detection of such basic patterns like *calls* (method,method) we construct a dedicated parser and analyze the parse trees. This approach is similar to [5], e.g. Other systems are based on regular expressions, like grep or perl scripts. A very interesting lexically-based approach is described in [8]. Similar patterns may be extracted by software information systems (e.g. CIA [2], Sniff [1]).

If we apply all of the transformations described in this section, we obtain a graph that represents the static structure of a program showing classes and interfaces together with their inheritance and association relations.

Our pattern-based design recovery has thus contributed to the static program analysis. The static structure graph is given by a set of relations, it can be used as input for further tools. For example a dedicated layout algorithm [9] has been developed for these kinds of graphs.

5 Higher Order Patterns

Based on the static structure graph higher order patterns like the well known design patterns may be recovered. In this paper we show how composite, strategy or bridge patterns [4] can be revealed. A discussion, how to discover patterns from [4] may be found in [6]. Note, however, that we only work out the structural aspects of the patterns, so slightly different implementations may not be found whereas some other clusters may randomly match our pattern.

The discovered patterns are used to insert new nodes and edges into the graph. There are two possible ways for graph transformations. The first follows the UML notation where class diagrams are not changed, but additional pattern symbols with connections to the constituents of the pattern are drawn. In the second way patterns are considered as new units of the software. So a pattern is a kind of super-node in our structure graph that hides the relations inside the pattern.

5.1 Composite Pattern

A composite pattern is the recursive structure in an inheritance tree, where one of the leaves is an aggregation of several roots. This pattern can be discovered in our graph.

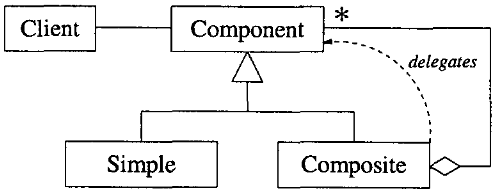

Figure 1: Structure of the composite pattern (from [3])

Start with a class C, collect all its descendants and see, if one of them holds a multiple aggregation to the root and delegates via that link, then there is high evidence for a composite pattern, where class C plays the role of "component", the picked out descendant class and its heirs act as "composite" and all other descendants are "leaves". The layered approach is obvious in this procedure. We start with the basic relation, derive structural patterns like aggregation or delegation, and then recover design patterns.

Note, that our really simple graph pattern generalizes the original composite pattern, since we take the transitive closure of the inheritance relation and do not require that the component already offers the full interface of the composite or that all components of the composite are called in a loop [4]. Nevertheless the number of wrongly found composite patterns is low.

We either transform the graph by introducing a new node type PATTERN and draw accordingly labeled edges to the relevant classes, or, we use an alternative transformation that creates supernodes and hence abandons a certain amount of the information gathered.

$SUB(C) = \{D | D \ extends^* \ C\}$

Note: *extends** denotes the transitive closure of the *extends* relation.

Check, if

$\exists D \in SUB(C) : D \ aggreg(multiple) \ C \ \wedge \ D \ delegates \ C$
$\Rightarrow composite(C, \mathcal{D}, \mathcal{A})$

where $\mathcal{D} = SUB(D)$ and $\mathcal{A} = SUB(C) - SUB(D)$.

We can transform this ternary relation into the usual binary structure of our graph by construction of super-nodes for sets of classes. $\mathcal{D}$ inherits C, $\mathcal{A}$ inherits C with corresponding labels.

5.2 Strategy Pattern

Characteristic for the strategy pattern [4] is the delegation of an algorithm to an interface or a class where the choice of different subclasses is possible.

In Java we only consider interfaces. We do not impose any further syntactic or semantic restrictions to identify the strategy pattern which leads to the situation, that we cannot distinguish strategy from similar patterns like state.

Start with class C and an interface D, where C *delegates* D. All classes B from B *implements* D, which provide implementations of the delegated methods offer a different strategy. Very close to the strategy pattern, where emphasis is

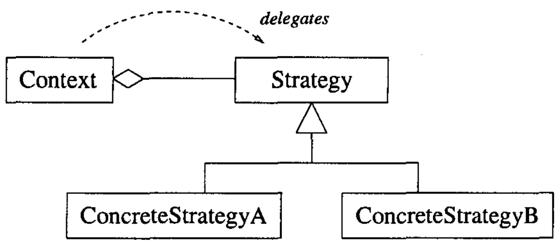

Figure 2: Structure of the strategy pattern (from [3])

put on the algorithm, is the state pattern [4], where the behavior associated with a particular state of the client is encapsulated.

As a simplification of our graph, all classes implementing the interface D may be omitted and a new edge C *deleg_strategy* D is introduced.

5.3 Bridge Pattern

Characteristic for the bridge pattern are two parallel hierarchies, one for a more abstract model and the other for its implementation. Each element of the model hierarchy holds a link to a corresponding element of the implementation part. This link is usually established by an association or aggregation between the two roots. It is inherited down the hierarchies. This association is used to delegate the implementation. In other words, it is a hierarchy of strategy patterns. Sometimes it suffices, if the model is implemented using the inherited interface, but sometimes a more specific interface has to be used. This is visible in Java source code by a narrowing downcast.

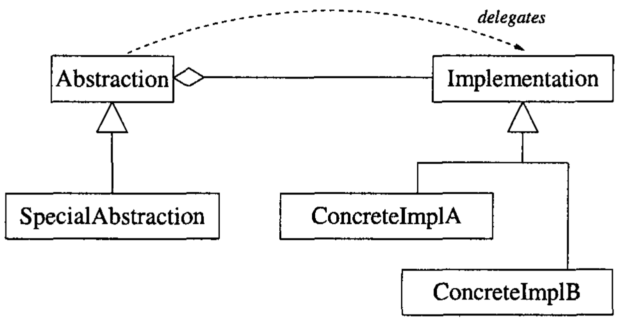

Figure 3: Structure of the bridge pattern (from [3])

Note that the structural view of the bridge pattern in [4] only concentrates on one element in the hierarchy and is identical to strategy or state. Therefore the bridge pattern cannot be distinguished from other design patterns using the structural view of [4]. Our approach will also classify bridges without downcasts as strategies.

We now define patterns for our restricted view of the bridge pattern.

Start with class C and interface D where C *delegates* D. Check if both are roots of inheritance hierarchies and if

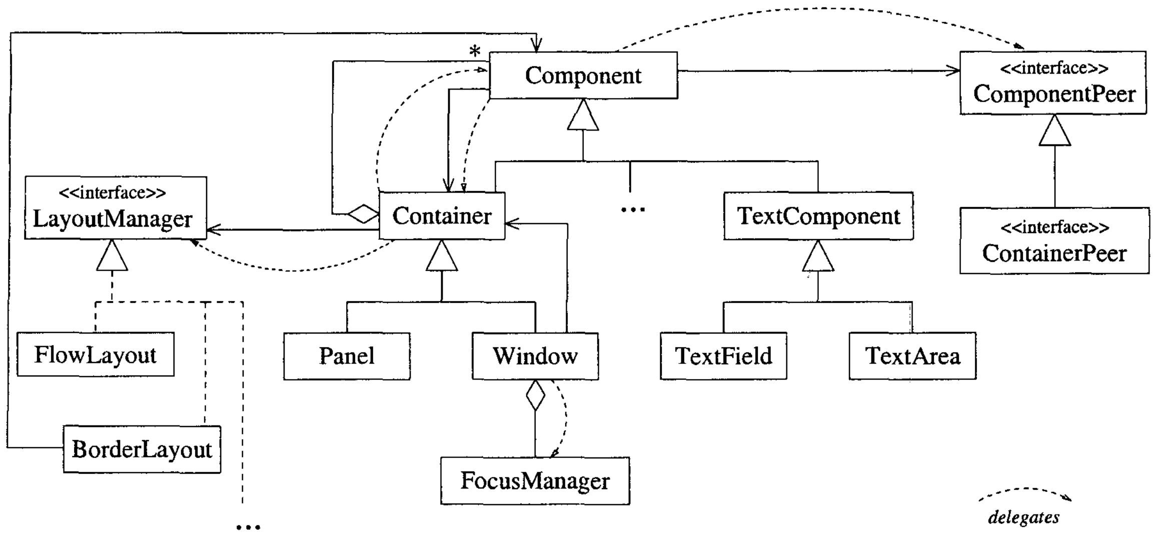

Figure 4: Case study: A part of the Java AWT library

downcasting is used in the second hierarchy. This downcasting indicates that the association or delegation of the two roots is refined for some of the subclasses.

Let $C \in$ CLASS, $D \in$ INTERFACE, where C *delegates* D
if $\exists A \in SUB(C)$, $B \in SUB(D)$
with A *calls(local)* $B \wedge D$ *downcasts* B
$\Rightarrow A$ *calls(attrib)* $B \Rightarrow A$ *delegates* B.

This also gives high evidence for a bridge pattern where C and D are the roots of the hierarchies. All subclasses of the two linked hierarchies contribute to the bridge. Note that the implications hold for the AWT package and most Java software.

Again we insert a new pattern node "bridge" with edges to the roots of the two hierarchies.

5.4 Summary

We have developed criteria to discover design patterns that are are completely based on the structure of the source code. The patterns help to understand the architecture of the Java source code. It is an extension to existing work on this field such as [7], because we can detect more instances of a pattern than approaches strictly relying on the pattern structures in [4]. The reason for this is that we collect more information from the source code, for example method calls.

The presence of interfaces in Java code makes it easier to define the structure of a pattern in contrast to C++.

The stepwise detection of more and more general patterns seems to be promising for the discovery of design information from source code. We want to demonstrate the approach in a small case study that was introduced in the last section. We are working on a compiler to construct the fundamental data base from Java source code and a tool for pattern matching and graph transformations.

6 Case Study: The AWT Package

We illustrate our approach by recovering design information from the Java AWT package (Here: AWT Version 1.0).

We start with the `Component` class the ancestor of all GUI classes. We find four references to

```
ComponentPeer peer;  Container parent;
Color foreground;  Font font;
```

Component owns 73 methods. The call relation is generalized to classes. We detect the following relations.

`Component`	*calls(attrib)*	`Container`
`Component`	*calls(local)*	`Container`
`Component`	*calls(attrib)*	`ComponentPeer`
`Component`	*calls(local)*	`ComponentPeer`
`Component`	*calls(static)*	`Toolkit`
`Component`	*calls(result)*	`Toolkit`
`Component`	*calls(param)*	`Graphics`
`Component`	*calls(this)*	`Component`
`Component`	*calls(local)*	`Window`
`Component`	*calls(param)*	`Event`
`Component`	*calls(static)*	`System`
`Component`	*calls(param)*	`PrintStream`

The local calls to `ComponentPeer` and `Container` can be changed to *calls(attrib)* by closer inspection of the initialization. The local call to `Window` reduces to a *calls(this)* via downcasting. So we notice a downcast from `Component` to `Window`.

Hence associations to `Container` and `ComponentPeer` are derived, that both turn out to be delegations, see for example, methods `disable` or `setForeground`.

```
...
public  void setForeground(Color c) {
    foreground = c;
    if (peer != null) {
        c = getForeground();
        if (c != null) {
        peer.setForeground(c);
        }
    }
}
...
```

Whereas the former does not lead to one of our patterns, because Container is no interface, the latter is an indication for the bridge pattern, because one root of a hierarchy of classes delegates to a root of a hierarchy of interfaces.

The class Container extends Component, it references LayoutManager and an array of Component which is created by attribute initialization.

```
...
public class Container extends Component {
  int ncomponents;
  Component component[] = new Component[4];
  LayoutManager layoutMgr;
...
```

The following call relations are found:

Container	*calls(static)*	System
Container	*calls(local)*	ContainerPeer
Container	*calls(this)*	Container
Container	*calls(attrib)*	Container
Container	*calls(param)*	Component
Container	*calls(attrib)*	Component
Container	*calls(super)*	Component
Container	*calls(local)*	Component
Container	*calls(local)*	Component
Container	*calls(local)*	LayoutManager
Container	*calls(attrib)*	LayoutManager
Container	*calls(param)*	Graphics
Container	*calls(local)*	Graphics
Container	*calls(param)*	Event

The deeper look at the initialization shows that most *calls(local)* transform to *calls(attrib)*. Let us illustrate this, by considering the following method.

```
...
public Component add(String name, Component comp) {
     Component c = add(comp);
     LayoutManager layoutMgr = this.layoutMgr;
     if (layoutMgr != null) {
         layoutMgr.addLayoutComponent(name, comp);
     }
        return c;
...
```

The addLayoutComponent is called for the local variable layoutMgr which is indeed the attribute layoutMgr initialized in line 3.

We further find a downcasting from ComponentPeer to ContainerPeer.

```
...
public Insets insets() {
    ContainerPeer peer = (ContainerPeer)this.peer;
    return (peer != null) ? peer.insets()
                          : new Insets(0, 0, 0, 0);
}
...
```

this.peer is the inherited attribute of type ComponentPeer.

Since Container creates the array Component[] we derive a multiple aggregation that turns out to be a delegation by a closer look at some of the methods, addNotify, for example.

```
...
public void addNotify() {
       for (int i = 0 ; i < ncomponents ; i++) {
           component[i].addNotify();
       }
       super.addNotify();
   }
...
```

The information obtained so far recovers the composite pattern. Container, and hence all its heirs play the role of "composite" of arbitrary components. All other children of Component act as "leaves".

Container further delegates to

- LayoutManager
- ContainerPeer
- Container itself (via the parent attribute)

The delegation to LayoutManager gives a hint to check for a strategy or bridge pattern. Since LayoutManager is implemented by several classes: FlowLayout, BorderLayout etc. we conclude the strategy pattern.

The third delegation to ContainerPeer indeed delivers the bridge pattern. Because a container and its peer are each part of the corresponding hierarchy and there is downcasting of the peer classes, all ingredients of the bridge pattern have been found. In fact, we have a larger bridge comprising nearly all components.

The delegation to Container itself does not lead to one of our three sample patterns.

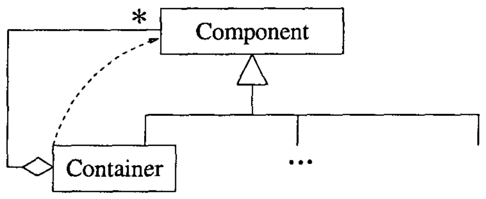

Figure 5: Case study: Composite pattern

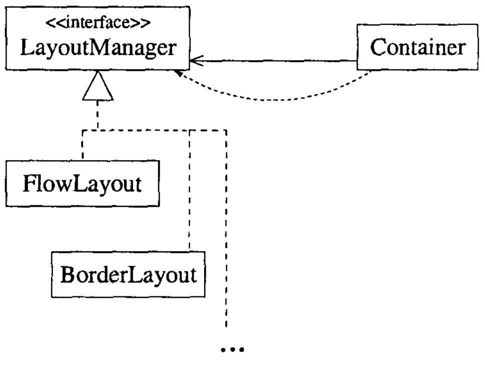

Figure 6: Case study: Strategy pattern

Let us take a closer look at two other classes.

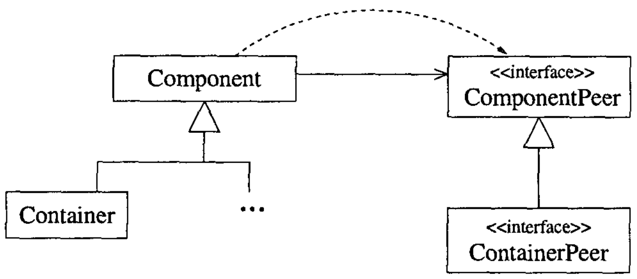

Figure 7: Case study: Bridge pattern

`Window` *extends* and *calls(attrib)* `Container`, but this association is neither a delegation nor an aggregation, therefore no new composite pattern is detected.

It further aggregates a `FocusManager` and delegates to it. This is, however, not a strategy pattern, since `FocusManager` is a single class. Generalization to several different `FocusManagers` would preferably be done by such a strategy.

`BorderLayout` *implements* LayoutManager, it references 5 Components. The relation `BorderLayout` *calls(attrib)* `Component` may lead to a pattern. From a closer look to the sets of the methods we derive a multiple association with a "composite"-like delegation, but since `BorderLayout` is no heir of `Component` obviously no composite pattern must be derived.

Acknowledgement

We like to thank the referees whose valuable comments helped to improve the presentation of our ideas.

References

[1] W. Bischoffberger: Sniff — A Pragmatic Approach to a C++ Programming Environment, *Proc of the 1992 USENIX C++ Conference*, 1992.

[2] J. Grass, Y. Chen: The C++ Information Abstractor, *Proc of the 1990 USENIX C++ Conference*, 1990.

[3] R. Kazman, M. Burth : Assessing Architectural Complexity, in P. Nesi, F. Lehner (eds.): *Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, IEEE CS press, 1998, pp. 104-112.

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

[5] W. Griswold, D. Atkinson: Managing the Design Tradeoffs for a Program Understanding and Transformation Tool, *J. Syst. Softw. 30*,1, 1995, pp 99-116

[6] L. Hankewitz: *Object-Oriented Design Recovery based on Source Code*, (in German), Masters Thesis, Würzburg University, 1997.

[7] C. Krämer, L. Prechelt: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software, in: *Proc. Working Conference on Reverse Engineering*, IEEE CS press, 1996, pp. 208-215.

[8] G. Murphy, D. Notkin: Lightwight Lexical Source Model Extraction, *ACM Trans on Software Engineering and Methodology 5*,3, 1996, pp.262-292.

[9] J. Seemann: Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams, in: *Proc. Graph Drawing '97*, LNCS 1353, Springer-Verlag, 1997, pp. 415–424.

[10] Rational Software Corporation: *The Unified Modeling Language 1.1*, `http://www.rational.com/uml/`, September 1997.