Abstract

The object-oriented design community has recently begun to collect so-called software design patterns: descriptions of proven solutions common software design problems, packaged in a description that includes a problem, a context, a solution, and its properties. Design pattern information can improve the maintainability of software, but is often absent in program documentation. We present a system called Pat for localizing instances of structural design patterns in existing C++ software. It relies extensively on a commercial CASE tool and a PROLOG interpreter, resulting in a simple and robust architecture that cannot solve the problem completely, but is industrial-strength; it avoids the brittleness that many reverse engineering tools exhibit when applied to realistic software. To evaluate Pat, we quantify its performance in terms of precision and recall. We examine four applications, including the popular class libraries zApp and LEDA. Within Pat’s restrictions all pattern instances are found, the precision is about 40 percent, and manual filtering of the false positives is relatively easy. Therefore, we consider Pat a good compromise: modest functionality, but high practical stability for recovering design information.

1 Design patterns for program understanding

The general problem of automatically recovering design intentions from program source code cannot be solved at all. Even partial solutions are extremely difficult. Not only is there an
enormous number of possible design intentions and not only can each of them be realized in an enormous number of different ways, but also is any single hint that is available in the source program very small, ambiguous, and unreliable. However, object-oriented programming and in particular software design patterns reduce the task, as they make much more design structure explicit in the source code.

Software design patterns, as introduced by Gamma et al. [7], Buschmann et al. [5] and others, are packaged descriptions of a common software design problem, its context, appropriate terminology, one or several solutions, and their advantages, constraints, and other properties. A design pattern packages expert knowledge and can be reused frequently and easily. Each pattern is a microarchitecture on a higher abstraction level than classes.

Design patterns are a rewarding target for reverse engineering: According to [1, 7], design patterns improve communication, both between designers and from designers to maintainers, by defining a common design terminology. Hence, it is useful to recognize instances of design patterns in designs where they were not used explicitly or where their use is not documented. Recognizing them could presumably improve the maintainability of software, because larger chunks could be understood as a whole. In fact we have found in a controlled experiment [12, 13] that maintainers equipped with design pattern information solved maintenance tasks quicker or with fewer errors than a control group of maintainers having only detailed “normal” source code comments. This also indicates that it is not trivial for people to detect design patterns in software. Thus, a tool for automatic design pattern recovery would be useful.

Automatically finding all instances of all design patterns is still an impossible task, but design patterns open, for the first time, the possibility of a reasonable, yet low-complexity partial solution to design recovery. This is the purpose of the present work: to explore the low end of design recovery. How simple can the architecture of a tool be that still produces useful output? Can we realize a reasonable part of the possible functionality for a tiny fraction of the usual cost?

The advantage of this approach is practicality. Such a tool will be much more robust than a more complicated one. It can be made industrial-strength — able to dependably process realistic software — with modest effort.

Our contributions are twofold:

1. We present an approach of extreme simplicity for finding instances of structural design patterns in existing software (C++ in our example) and a tool, the Pat system.
2. We empirically show that despite its simplicity the approach solves a non-negligible part of the design pattern recovery problem for real-world software.

Throughout most of this article we will use the term pattern to refer only to the solution used in a particular design pattern instance.

We describe in order the general approach taken, the representation we used for patterns and software, and a quantitative evaluation of the system. Afterwards we discuss alternative approaches to the problem and related work.
2 Approach

2.1 Fundamental design decisions

The following central design decisions underlie our work:

- **D1**: Rely on existing tools as much as possible, in particular for processing the source code.

- **D2**: Prefer restricting the tool’s functionality over making it fragile.

- **D3**: Within its fundamental restrictions, make the tool freely extensible.

The consequence of D1 is using a commercial object-oriented CASE tool as a front-end and Prolog as a search machine. The consequence of D2 is searching for structured design patterns only (i.e., patterns that hardly rely on specific behavior of methods, but are largely determined by the static structure of the classes instead) and limiting the search to only one implementation variant of each pattern. The consequence of D3 is encoding the pattern descriptions modularly.

2.2 The basic idea: How the Pat system works

We will use an example to explain how Pat works.

![OMT diagram of the design pattern “Adapter”](image)

Figure 1: OMT diagram of the design pattern “Adapter”

See the description of an Adapter (more precisely: Object Adapter) in the OMT class diagram [17] of Figure 1. The purpose of an Adapter is to provide an additional interface to an adapted class (called the adaptee), so that the adapter class can adhere to the calling conventions of a client but the interface of the adaptee need not be changed. The Adapter pattern requires that there are four classes Client, Target, Adapter, and Adaptee. Adapter must be a subclass of Target and must delegate Client calls to a method Request of the Target class to a method SpecificRequest (with different interface) of the Adaptee class. An Adapter instance needs an association (e.g. a pointer) to an Adaptee instance.
When looking for the Adapter pattern, search the C++ files of an existing software system for triples of Target, Adapter, and Adaptee that have methods corresponding to Request and SpecificRequest and have the association and delegation mentioned above. Any such constellation may represent an instance of the Adapter pattern to be found and should hence be output.

2.3 The Pat architecture

The fundamental design idea of Pat is to represent both patterns and designs in PROLOG and let the PROLOG engine do the actual search. The basic design information itself is extracted from source code by the structural analysis mechanism of a commercial CASE tool: Paradigm Plus [15] is an object-oriented CASE tool that supports several methods and notations, one of them OMT. Modeling information is stored in an object repository and accessed by textual and graphical editors and an internal programming language. Paradigm Plus provides a structural analysis facility called “import” that extracts information about classes directly from C++ header files.

More concretely we proceed as follows (see also Figure 2):

1. Each pattern is represented as a static OMT class diagram (see Figure 1). These diagrams constitute the repository $P$ (for “patterns”).
2. A straightforward program $P2prolog$ converts $P$ into PROLOG rules. The generated form is one rule for each pattern, representing design properties that are necessary but not sufficient to diagnose the pattern; see Section 3.2.
3. The structural analysis mechanism of Paradigm Plus extracts design information from C++ header files and stores it in the repository in OMT form. The resulting part of the repository is called $D$ (for “design”).
4. Another straightforward program $D2prolog$ converts $D$ into PROLOG facts; see Section 3.1.
5. A PROLOG query $Q$ detects all instances of design patterns from $P$ in the examined design $D$. Duplicates of design patterns that often occur in the PROLOG output are removed. Manual postprocessing removes false positives; see Section 4.

Note that this approach searches for specific implementation forms of patterns, which are sometimes called design templates. Most design patterns can be implemented by several different design templates. Pat as presented in this article uses only one template per pattern, but alternatives whose occurrence is expected could be added at will.
2.4 Implementation details

For the implementation we used Paradigm Plus 2.01 [15] and Visual Prolog 4.0 Beta (Professional Version) [14].

The programs P2prolog and D2prolog are written in the BASIC dialect provided by Paradigm Plus and are executed directly by Paradigm Plus with direct access to its repository.

The resulting PROLOG program performs the search and generates one output line per pattern instance found. Each line has the form of a \LaTeX macro call such as \\
\texttt{adapter\{Target\\{Adapter\\{Adaptee\}}}\\n
The pattern instance list is then filtered for duplicates. Suitable \LaTeX macros convert the resulting instances into graphical OMT form as shown above to provide a basis for a reverse-engineered design document.

3 PROLOG representation

3.1 Source code to PROLOG mapping

We represent the relevant information of C++ header files by PROLOG facts. Implementation files need not be consulted. As an example, the class declaration

```cpp
class zPane:public zChildWin {
    zDisplay* curDisp;
    /* ... */
public:
    virtual void show(int=SW\_SHOWNORMAL);
    /* ... */
};
```

would result in these PROLOG facts:

- `class(concrete, zPane).
- `inheritance (zChildWin, zPane).
- `association (zPane, zDisplay).
- `operation(virtual, zPane, show).

The following rules are applied for generating such facts:

Any class declaration of the form `class C { declarations } results in a fact `class(ca, C). ca has the value `concrete if there is at least one public constructor among the `declarations and abstract otherwise. Any class declaration of the form `class C : B { declarations } results in the same fact plus another fact `inheritance(B, C), or multiple such facts if multiple base classes are given. Paradigm Plus always treats inheritance as public.

A method declaration of the form `virtual Resulttype Methodname(parameterlist) in the public part of the class C results in a fact of the form `operation(virtual, C, Methodname). If the virtual modifier is missing, `normal is used instead of `virtual. Abstract functions (called "pure virtual" in C++) are also treated as `virtual. If the static modifier is used, no fact is generated at all. Other method modifiers are ignored.

A pointer or reference data member declaration of the form `Typename *Membername or `Typename &Membername for any named type `Typename in any part of the class C results in a fact of the form `association(C, Typename). The same is true for pointers to pointers and so on.
A non-reference data member declaration of the form Typename Membername for any named type Typename in any part of the class C results in a fact of the form aggregation(C, Typename, exactlyone). If the member explicitly is an array, i.e., the declaration has the form Typename Membername[] or Typename Membername[constexpression], it results in a fact of the form aggregation(C, Typename, many). The same is true for multidimensional arrays. Paradigm Plus cannot detect associations or aggregations that are implemented by other than the above means, for instance by container classes or temporarily by method parameters.

All other header file content is ignored.

### 3.2 Patterns to Prolog mapping

The Prolog rule for each pattern gathers the facts required to diagnose a pattern instance. As an example, see again the Adapter pattern in Figure 1. P2prolog converts this OMT class diagram into the following Prolog rule:

```prolog
adapter(Target, Adapter, Adaptee):-
    class(_, Target),
    class(concrete, Adapter),
    class(concrete, Adaptee),
    operation(virtual, Target, Request),
    operation(_, Adapter, Request),
    operation(_, Adaptee, SpecificRequest),
    inheritance(Target, Adapter),
    association(Adapter, Adaptee).
```

This Prolog rule describes necessary but not sufficient properties of classes for forming one kind of Adapter pattern instance. In addition, there may be multiple alternative rules for diagnosing different design templates representing the pattern.

The derivation of the Prolog rules is straightforward, except for the following cases: First, classes that are abstract in the pattern are allowed to be either abstract or concrete in the software. Methods that are abstract in the pattern are required to be virtual, but not necessarily abstract ("pure virtual" in C++). Methods that are concrete in the pattern are allowed to be either virtual or normal.

Second, call delegation is not modeled at all, as Paradigm Plus does not detect it. For instance the Adapter pattern demands that there exists a delegation from a method Adapter::Request to Adaptee::SpecificRequest. However, because Paradigm Plus cannot extract delegations, the delegation must not be modeled in our Prolog rule or else the rule could never be matched.

Third, the client is not modeled because an Adapter is still an Adapter if it occurs stand-alone without any actual client, e.g., in a library. Similar considerations apply for the Prolog rules of the other design patterns.

Fourth, in patterns where there may be an arbitrary number of subclasses of a particular kind, only one such class is modeled in the rule, because that is sufficient to detect the pattern. See the Bridge, Composite, and Decorator patterns as examples.

Fifth, the actual names of classes, attributes, and methods are ignored.

Technically, the actual Prolog rules used in Pat have two additions over the ones shown here. First, they contain local cuts (getbacktrack/cutbacktrack pairs) to restrict the rules to match
one method per operation clause and ignore the rest. Second, classes are checked for inequality with clauses like Adapter <> Adaptee etc. to avoid senseless matches and combinatorial explosion.

Here are the Prolog representations of the other four structural design patterns. If you are not familiar with these patterns, you may first want to read the short description of their purpose at the end of section 5.

A Bridge consists of at least four classes: the abstract Abstraction superclass, one or several Refined Abstraction subclasses, the abstract Implementor superclass, and one or several Concrete Implementor subclasses. Bridges can often be re-interpreted as instances of the Strategy pattern and vice versa.

![OMT diagram of the design pattern “Bridge”](image)

```
bridge(Abstraction,RefinedAbstr,Impl,ConcreteImpl):-
class(_,Abstraction),
class(concrete,RefinedAbstr),
class(_,Impl),
class(concrete,ConcreteImpl),
operation(_,Abstraction,Op),
operation(virtual,Impl,OpImpl),
operation(_,ConcreteImpl,OpImpl),
inheritance(Abstraction,RefinedAbstr),
inheritance(Impl,ConcreteImpl),
aggregation(Abstraction,Impl,exactlyone).
```

This rule does not model the delegation from Abstraction::Op to Impl::OpImpl.

A Composite consists of at least three classes: the abstract Component superclass, one or several Leaf subclasses, and one or several Composite subclasses.

```
composite(Component,Leaf,Composite):-
class(_,Component),
class(concrete,Leaf),
class(concrete,Composite),
operation(virtual,Component,Op),
operation(_,Leaf,Op),
```


This rule ignores the fact that `Composite::Op` must have a loop of `Op` calls for all children. The semantics of `add`, `remove` and `getChild` are also ignored, because no such information is available. If several of the classes have many operations, the combinatorial explosion in the operation clauses of this rule makes the rule impractical. We then actually drop all operation clauses from the rule. As the method semantics are not checked anyway, this omission makes rarely any difference.

A **Decorator** consists of at least four classes: the abstract `Component` top class with a Concrete `Component` subclass and an abstract `Decorator` subclass; the latter has one or several further subclasses called Concrete Decorators.

```prolog
decorator(Component,ConcreteComp,Decorator,ConcreteDeco):-
class(_,Component),
class(concrete,ConcreteComp),
class(_,Decorator),
class(concrete,ConcreteDeco),
operation(virtual,Component,Op),
operation(_,ConcreteComp,Op),
operation(virtual,Decorator,Op),
```
operation(_,ConcreteDeco,Op),
inheritance(Component,ConcreteComp),
inheritance(Component,Decorator),
inheritance(Decorator,ConcreteDeco),
aggregation(Decorator,Component,exactlyone).

This rule ignores the delegations from Decorator::Op to Op of the decorator’s aggregated component and the implementation of ConcreteDeco::Op as a call to Decorator::Op plus some decorator behavior.

A Proxy consists of three classes: a Real Subject class, its Proxy class and their abstract Subject superclass.

proxy(Subject,RealSubject,Proxy):-
class(_,Subject),
class(concrete,RealSubject),
class(concrete,Proxy),
operation(virtual,Subject,Op),
operation(_,RealSubject,Op),
operation(_,Proxy,Op),
inheritance(Subject,RealSubject),
inheritance(Subject,Proxy),
association(Proxy,RealSubject).

This rule ignores the implementation of Proxy::Op as a delegation to RealSubject::Op plus some proxy behavior.
Figure 6: OMT diagram of the design pattern “Proxy”

4 Evaluation

Three questions arise, given a design recovery system such as Pat:

1. What fraction of pattern instances is found?
2. What fraction of the output consists of false positives (spurious instances)?
3. How useful is the output for actual program understanding and maintenance tasks?

We cannot answer the third question at this time, as it requires a rather costly empirical study; a first experiment is described in Section 6, however.

We use the information retrieval measures called precision and recall [8] to answer the first two questions.

Assume that Pat outputs $P$ distinct pattern instance candidates. Assume further that the design analyzed actually contains $T$ true pattern instances that are implemented by one of the design templates we search for and that $F$ of these $T$ are found by Pat. Then \( \text{precision} := \frac{F}{P} \) and \( \text{recall} := \frac{F}{T} \). Note that these numbers ignore pattern instances using design templates not in our rule set. We also measured the execution time needed for the automatic search and the time needed for human filtering of the results to remove false positives.

4.1 The benchmarks

Four different sets of classes were examined: Network Management Environment Browser (NME), Library of Efficient Datatypes and Algorithms (LEDA 3.0, [10]), the zApp class library (zApp, [9]), and Automatic Call Distribution (ACD). NME and ACD are telecommunication software developed at Computec, the other two are widely used class libraries.

None of these four benchmarks included explicit design information; all data was extracted from C++ header files as described above. Table 1 characterizes the size of the benchmark applications.
as found by the structural analysis step and as obtained from the D2prolog conversion. Except for NME, the size of the benchmarks is considerable.

<table>
<thead>
<tr>
<th></th>
<th>classes</th>
<th>attrib.</th>
<th>operat.</th>
<th>aggr.</th>
<th>assoc.</th>
<th>inherit.</th>
<th>kByte facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>NME</td>
<td>9</td>
<td>34</td>
<td>131</td>
<td>0</td>
<td>10</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>LEDA</td>
<td>150</td>
<td>501</td>
<td>4084</td>
<td>91</td>
<td>151</td>
<td>67</td>
<td>243</td>
</tr>
<tr>
<td>zApp</td>
<td>240</td>
<td>1176</td>
<td>3590</td>
<td>143</td>
<td>155</td>
<td>145</td>
<td>205</td>
</tr>
<tr>
<td>ACD</td>
<td>343</td>
<td>1306</td>
<td>2879</td>
<td>457</td>
<td>461</td>
<td>191</td>
<td>204</td>
</tr>
</tbody>
</table>

Table 1: Number of classes, attributes, operations, aggregations, associations, and inheritances found by Paradigm Plus in each of the applications and size of the generated Prolog facts file in kByte.

### 4.2 Evaluation procedure and results

Each of the four resulting Prolog facts files was used in a separate pattern search run. The results are summarized in Table 2. For each application and for each design pattern the table gives the number of pattern instances found by the search mechanism (“found”) and the number of these that were not spurious (“true”). Below are total recall and precision values over all patterns and the runtime in seconds taken by the Prolog program. As for the runtimes, the structural analysis and D2prolog steps take up to two hours, i.e., much longer than the actual pattern search.

<table>
<thead>
<tr>
<th></th>
<th>NME found</th>
<th>LEDA found</th>
<th>zApp found</th>
<th>ACD found</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>Adapter</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>Bridge</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Proxy</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Recall</th>
<th>Precision</th>
<th>Prec. w. deleg.</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>100%</td>
<td>50%</td>
<td>100%</td>
<td>1 sec</td>
</tr>
<tr>
<td>Bridge</td>
<td>≈100%?</td>
<td>≈14%</td>
<td>≈43%</td>
<td>2 sec</td>
</tr>
<tr>
<td>Composite</td>
<td>100%</td>
<td>≈53%</td>
<td>100%</td>
<td>3 sec</td>
</tr>
<tr>
<td>Decorator</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>36 sec</td>
</tr>
</tbody>
</table>

Table 2: Number of pattern instances found by Pat and approximate number of true instances, resulting recall, precision, and Prolog runtimes measured on a PCI-Bus PC with Pentium P133 and 32 MByte RAM under Windows 95.

**Recall:** Obviously, when computing recall, structurally different alternative implementations of the patterns have to be taken out of account, as Pat cannot possibly find them in the given setup, but would find them with appropriate additional rules. Given this restriction, how high is Pat’s recall? Except for NME, we had no definitive information about the set of patterns actually used in the programs. However, for NME we know and for zApp we believe, judging from the documentation, that Pat achieved full recall. In LEDA and ACD recall is unknown, because Pat may have missed some patterns for the following reason.

The recognition of a pattern may fail (only) due to an undetected aggregation. An aggregation
will go undetected if either (1) it looks like an association or (2) it is completely hidden due to the use of a container type for its implementation, such as a Vector or Bag class. We have checked\(^1\) that case (1) does not lead to any undetected pattern, but case (2) may have occurred. Even so, case (2) is probably relevant only for the Composite pattern, as it is the only one that requires an aggregation of multiple objects. Thus, recall is most probably close to 100 percent even for LEDA and ACD.

**Precision:** Because our pattern rules do not represent sufficient conditions for pattern instances, precision is not perfect. Some constructions will lack required (but unchecked) properties, yet be returned, incorrectly, as pattern instances.

How does one decide what is such a false positive and what is a true pattern instance? We took the following approach: (1) In many cases false positives are obvious when the class and method names clearly indicate unrelated classes. (2) In other cases correct pattern instances are obvious via class and method names indicating the semantics required by the pattern. In the remaining cases, the pattern instance has to be checked by manually consulting (3) available documentation or (4) source code. It turns out that plausibility checks of pattern instance candidates can often be done quite rapidly using only methods (1) through (3). We applied methods (1) and (2) for all projects and also method (3) for LEDA and zApp. We did not check source code at all.

Our evaluation approach implies that the precision values in Table 2 are approximations. The line labeled ‘precision’ in Table 2 gives the precision values that result directly from dividing ‘true’ by ‘found’. The line labeled ‘prec. deleg.’ shows precision values that would result if Paradigm Plus were able to detect all simple call delegations and our pattern rules contained checks for them, making most of the false positives disappear — all spurious Adapters and Bridges lack the correct delegations. The following sections discuss additional aspects of interest for each of the four benchmarks.

**NME:** The original designer of the software confirmed that Pat found one true and one spurious Adapter. The spurious Adapter would have been rejected had delegations been checked.

**LEDA:** We decided which of the pattern instances to consider correct by consulting the LEDA manual. This work took about one hour for a programmer without prior knowledge of LEDA. 56 of the 59 Bridges occur because each of the 8 classes circle, line, p_dictionary, point, polygon, real, segment, string (all subclasses of handle_base) seems to form a Bridge with each of the 7 classes circle_rep, line_rep, point_rep, polygon_rep, rrep, segment_rep, and string_rep (all subclasses of handle_rep). If Pat could check for the correct delegations, only the correct 7 of these 56 pairs should remain. The 6 false Composites were found by a relaxed rule (without operation clauses) as described in Section 3.2.

**zApp:** The evaluation of the output for zApp was also done with the manual. This work took one hour. All of the false positives could have been suppressed by checking for correct delegations in the Adapter candidates. Surprisingly, there is neither a Composite nor a Decorator in zApp, although it is a GUI library. But zApp does indeed not use the Composite concept of handling containers and basic components in the same way nor the Decorator concept of transparently attaching additional functionality to an object.

**ACD** is a large project and created so much output that we were unable to check correctness completely. Instead, we relied on conservative common sense judgement from the class names

---

\(^1\)The check was made by re-running all four experiments with an additional rule that allowed to interpret any association as an aggregation. This led to more than twice as much output, none of which contained any additional pattern instances.
combined with another plausibility check: In the case of the Adapters we assumed that exactly those are correct where the name of Request is similar to the name of SpecificRequest. In the case of proxies we drew conclusions from the class names alone; no Proxy seems to be in ACD. Evaluating the solutions for ACD in this manner took 30 minutes.

5 Design alternatives

While Pat performs quite well, there are other ways of recovering design patterns from code. In this section, we discuss and relate the fundamentally different approaches. Basically, there are three kinds of information to identify design pattern instances.

First, information on declarations and the “uses”-relation can be extracted using basic compiler techniques (scanning, parsing, name resolution). This is the approach used by the Pat system and allows to rely on CASE tool functionality. Such information can be used for finding structural matches. Structural comparison is insufficient for finding those design patterns that rely on particular behavior of methods (behavioral patterns). It is sufficient, however, for design patterns that rely mostly on static aspects of software composition (structural patterns).

Second, the semantics of the program can be partially analyzed using advanced compiler and program analysis techniques (interprocedural data dependence and data flow analysis, alias analysis etc.). In principle, semantic information could identify instances of behavioral patterns, not only structural patterns. However, there are two problems. (1) As program semantics are in general not computable, the analysis techniques will consist of complex heuristics that make the search program difficult to create and extend, make it dependent on programming style, and make its performance unpredictable. (2) Some of the concepts underlying design patterns are vague (for instance “call delegation”). Therefore, whichever heuristic formalization is used, counterintuitive results may occur. Furthermore, C++ is notoriously difficult to analyze, due to its weird syntax, overly complex semantics, and the preprocessor. This approach exhibits more of the complexity of general program understanding and may produce less leverage from the specificity of design patterns.

Third, the names of classes, methods, and attributes expected in a particular design pattern could be matched against software using a heuristic abbreviation recognizer and thesaurus etc. For some programs, names may provide a useful shortcut to program semantics so that heuristic name analysis could be a useful complement to the above techniques. In particular, name analysis would work best for behavioral aspects of the program. However, name analysis is inherently unreliable and extremely sensitive to the naming conventions used, if any. Therefore, it might make performance even more unpredictable.

We chose to use only the first approach, which achieves the maximum ratio of performance to construction effort. It results in a simple, efficient, and relatively robust system. In particular, we can use commercially available building blocks for the analysis and deduction parts, thus further reducing the complexity and increasing the quality of the implementation. This architecture limits the covered set of design patterns to structural patterns; behavioral patterns cannot be found this way.

However, such a reduced search space is still useful. From a reverse engineering point of view, finding instances of these patterns yields the following information. Adapter instances signal where classes are used in multiple contexts, requiring different interfaces. Bridge instances show where the interface and the implementation of a module are encapsulated in separate classes, so
that both can be changed independently; Bridges indicate reuse or places where much change is expected. Bridges may also indicate instances of the closely related Strategy pattern, which allows changing the implementation of an operation at run time. Composite and Decorator instances signal easily extensible areas of a program where collections of components or extended versions of components are handled like basic components alone. Proxy instances indicate where a surrogate of a (very large?) object was used instead of the object itself or where functionality was added to an object transparently, for example, firewall, encryption, or caching functionality.

Nevertheless from the reverse engineering point of view, it is clearly desirable to extend the scope of the pattern search. Semantic information and name analysis may both be viable and useful means for recovering behavioral design patterns and should be pursued in further work.

6 Related work

As discussed above, Pat does not strive for detailed program understanding or general design recovery as some other advanced work in reverse engineering does, for instance that of Biggenstaff, Mithander, and Webster [2, 3] or the classical work of Rich and Waters [16]. In those cases, a wide gap has to be closed from the syntactic representation of the program (and maybe other artifacts) to the understanding of semantics and pragmatics. When searching for structural design patterns, this gap is much smaller for three reasons. First, the rich syntax of object-oriented languages contains much information about architectural features of design patterns, such as inheritance relations, associations and aggregations. We do not attempt to analyze software in non-oo languages. Second, the semantics of a structural design pattern are closely coupled to its syntactic representation and therefore easy to recognize (except for the problem of false positives, see Section 4). Third, a small set of possible pragmatic intentions is packaged in the description of a design pattern. Therefore, structural design patterns allow for inferring program pragmatics from syntactic source code features with simple machine deduction and a modest amount of additional interpretation by the user. In particular, design pattern search, at least for structural patterns, does not call for automatic concept assignment and the output is useful without a domain model. For these reasons we find it worthwhile to investigate the leverage that can be gained from design patterns in program understanding, using much simpler program analysis techniques than were required previously; searching for structural design patterns has a good price/performance ratio.

Design patterns are a young field and currently they are mostly used for understanding and communicating during the invention of designs. Work is also beginning towards creating tool support for handling design pattern instances as explicit design and program entities. Such tools can be focused on patterns [4] or incorporate patterns within a broader perspective [6]. In the far future, programs constructed with such tools may make design pattern recovery much easier, because they can reliably document pattern instances created in the code. The tool of [6] also supports semi-automatic reverse engineering of non-documented pattern instances. Forward engineering using patterns is particularly popular in the context of component architectures [11].

Several pattern practitioners [1] agree that one of the largest benefits of design patterns is their use as a means of communication and understanding. This observation suggests that finding patterns in existing designs should make understanding these designs easier.

The conservative, controlled experiment mentioned above has recently corroborated this assumption [12, 13]. Two groups of subjects received the same program and were asked to outline
how they would make a certain program change. The program for one of the groups had its design patterns documented in addition to thorough, conventional program documentation. For the other group, the pattern documentation was missing. The experiment was repeated with two different programs in a counterbalanced design. For one program, the solutions of both groups were of similar quality, but the group with pattern documentation finished significantly faster. For the other program, the group with pattern documentation produced a completely correct solution twice as often as the group without pattern documentation.

7 Conclusion

Automated search for instances of structural design patterns can be implemented by a rather simple software architecture, Pat. The key to this simplification is building on structural analysis capabilities of a commercial ooCASE tool and the search capabilities of Prolog, thereby making the implementation small, reliable, and efficient. The approach restricts the search capabilities to structural (as opposed to behavioral) design patterns, but exhibits an extremely good effort to performance ratio.

Often all design pattern instances within the system’s search space are recovered from the C++ source code. In addition, the Pat output contains a number of false positives. In our four benchmark applications, detection precision is between 14 and 50 percent. Overall, this precision is acceptable. The remaining false positives can be sorted out with a modest amount of manual work (typically a few minutes per resulting pattern instance). We conclude that our approach is a fast and simple partial solution to recovering design pattern information from source code.

Automatic detection of design pattern instances is probably a useful aid for maintenance purposes — for quickly finding places where extensions and changes are most easily applied. How useful automatic pattern finding is should be the subject of further study.

Further work should also explore how name analysis and/or semantic analysis can be employed to detect behavioral patterns in addition to structural ones.

References


