# Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software

Christian Krämer (ckraemer@ctec-sw.com)
Computec GmbH
Software Engineering Dept.
D-76133 Karlsruhe, Germany *

Lutz Prechelt (prechelt@ira.uka.de)
Fakultät für Informatik
Universität Karlsruhe
D-76128 Karlsruhe, Germany
+49/721/608-4068, Fax: +49/721/694092

## Abstract

*The object-oriented design community has recently begun to collect so-called design patterns: cliches plus hints to their recommended use in software construction. The structural design patterns Adapter, Bridge, Composite, Decorator, and Proxy represent packaged problem/context/solution/properties descriptions to common problems in object-oriented design. Localizing instances of these patterns in existing software produced without explicit use of patterns can improve the maintainability of software. In our approach, called the Pat system, design information is extracted directly from C++ header files and stored in a repository. The patterns are expressed as* Prolog *rules and the design information is translated into facts. A single* Prolog *query is then used to search for all patterns. We examined four applications, including the popular class libraries zApp and LEDA, with Pat. With some restrictions all pattern instances are found; the precision is about 40 percent. Since manual filtering of the output is relatively easy, we consider Pat a useful tool for discovering or recovering design information.*

*Keywords: design patterns, object-oriented, CASE,* Prolog, *maintainability*

## 1 Design patterns for understanding

The structural design patterns introduced by Gamma et al. [4] are concepts to improve the understanding of object-oriented designs. A design pattern packages expert knowledge; it represents a solution to a common design problem and can be reused frequently and easily. Each pattern is a microarchitecture on a higher abstraction level than classes. Design patterns are meant to be design building blocks for better software construction and designer communication. In a pattern-based methodology a pattern not only consists of a solution (a cliche) but of a description of the problem, problem context, terminology, one or several solutions, and solution properties and constraints.

In this work, we will use the term *pattern* to refer only to (one of) the solution(s). Therefore, for our purpose, a pattern is just a particular kind of design cliche [13].

It would be useful to find instances of such patterns in designs were they were not used explicitly or where their use is not documented. This could improve the maintainability of software, because larger chunks could be understood as a whole.

We present a tool, the *Pat* system, that searches for design pattern instances in existing software. We describe in order the general approach taken, related work, the CASE tool used, the Prolog representation for the search, and a quantitative evaluation of the system.

As an example of a pattern, see the description of an Adapter in the OMT diagram [12] of Figure 1. The purpose of an Adapter is to provide an additional usage 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. To be able to do this, an *Adapter* instance needs an association (e.g. a pointer) to an *Adaptee* instance.

*Appeared in Proc. Working Conf. on Reverse Engineering, IEEE CS press, Monterey, November 1996.

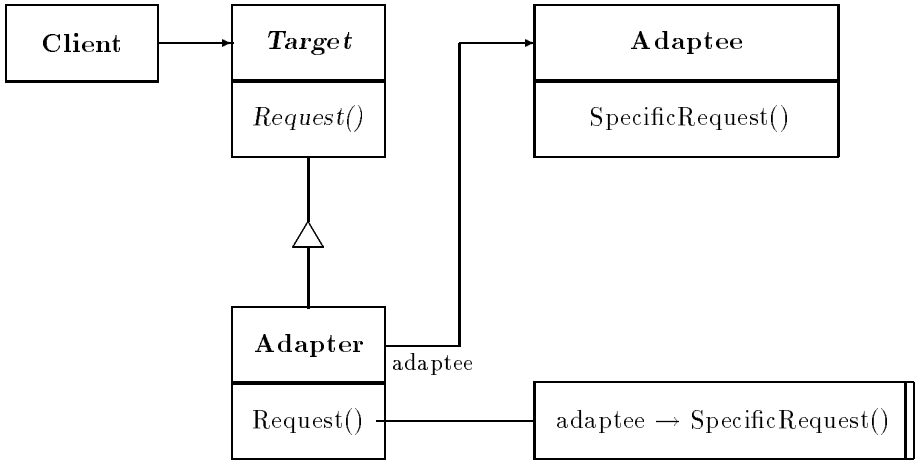


Figure 1. OMT diagram of the design pattern Adapter

With respect to the Adapter pattern, the purpose of the *Pat* system is to find triples of *Target*, *Adapter*, and *Adaptee* that have the required *Request* and *SpecificRequest* methods and the required association and delegation. Of course, the classes and methods can have arbitrary names. Any such constellation represents an instance of the Adapter pattern to be found by *Pat*.

In this study, we limited the set of considered design patterns to five *structural object patterns* in the sense of Gamma et al., namely Adapter, Bridge, Composite, Decorator, and Proxy. Additional structural patterns could be used as well.

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 were encapsulated in separate classes, so that both can be changed independently. This gives a hint to an area in the program where much change or reuse was expected. Composite and Decorator instances signal easily extensible areas of a program. Proxy instances allow to use a surrogate of an object instead of the object itself; they often hint to points in the program were very large objects are handled or objects that are expensive to create but may never be used after creation.

## 2 Approach

The fundamental idea for the automated search in *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 object-oriented CASE tool. More concretely we proceed as follows (see also Figure 2):

1. Each pattern is represented as a static OMT diagram as seen in Figure 1 — much like in the book by Gamma et al. These diagrams constitute the repository $P$ (for patterns); see Section 5.2.
2. A program `P2prolog` is used to convert $P$ into Prolog representation. The generated form is one rule for each pattern, representing design properties that are required but not sufficient to diagnose the pattern; see Section 5.
3. The structural analysis mechanism of the ooCASE tool is used to extract design information from C++ header files and represent it in the repository in OMT form. The resulting part of the repository is called $D$ (for design).
4. Another program `D2prolog` converts $D$ into Prolog representation; see Section 5.
5. A Prolog query $Q$ detects all instances of design patterns from $P$ in the examined design $D$. Simple automatic postprocessing is used to remove duplicates of design patterns that often occur in the Prolog output. Manual postprocessing is required to remove false positives; see Section 7.

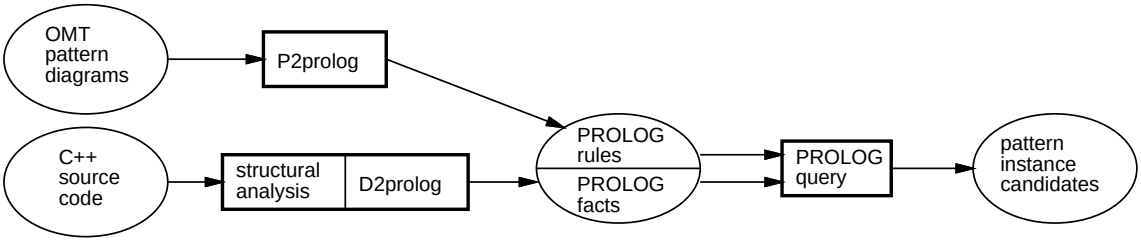


Figure 2. Architecture of the *Pat* system

### 2.1 Limitations

One important limitation of the approach is that some characteristics of design patterns require too much semantic information about the behavior of methods to be modeled by a CASE tool, let alone to be extracted from source code automatically today. An example of this is the design pattern Composite which requires operations for adding and deleting elements and for iterating over all of these elements. For the same reason it is also difficult to find behavioral patterns instead of structural ones.

## 3 Related work

Design patterns are a young field and until now, they are mostly used for understanding and communicating during the *production* of designs. Seven 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.

In contrast to much other work in reverse engineering, *Pat* does not strive for detailed program understanding [3, 11] or design recovery [2]. In those cases, a wide gap has to be closed between the syntactic representation of the program (and maybe other artifacts) and the understanding of semantics and pragmatics that is to be gained. On the other hand, 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 structural (architectural) features of design patterns. 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 relatively easy to recognize (except for the problem of false positives). Third, a small set of possible pragmatic intentions is packaged in the description of a design pattern; see Section 1. Therefore, structural design patterns allow to infer program pragmatics from syntactic source code features with moderately complex machine deduction and only a modest amount of additional interpretation by the user. In particular, design pattern search does not call for automatic concept assignment [3] and the output is useful without a domain model.

Koenig [7] suggests the notion of *antipatterns* that capture bad or non-working designs that are frequently "reused". Should structural antipatterns exist, our approach could be used as a design quality checker by searching for antipattern instances.

## 4 The *Paradigm Plus* ooCASE tool

*Paradigm Plus 2.01* [10] by Platinum (formerly ProtoSoft) is an object-oriented CASE tool for all phases of the software lifecycle. Several methods and notations are supported, one of them OMT. Modeling information is stored in an object repository and is accessed by textual and graphical editors and an internal programming language, which is a BASIC dialect with extensions.

The most interesting aspect of *Paradigm Plus* for our purpose is structural analysis facility called "import" that extracts information about classes directly from C++ header files. The extracted information relevant for us is class names, attribute names, method names and properties, inheritance relations, and association and aggregation relations. In a class `A`, members of the form `B x` or `B x[n]` or `B x[]` etc. are considered aggregations; members of the form `B *x` or `B &x` etc. are considered associations.

### 4.1 Limitations

*Paradigm Plus* cannot find associations or aggregations

that are implemented in any other way, such as with tables or graph managers. *Pat* can thus only find pattern instances whose aggregation or association relations are based on the direct means listed above. Thus, in all subsequent discussion we will consider this set of pattern instances only.

Unfortunately, other information that would also be relevant for a precise search for pattern instances is *not at all* extracted by the structural analysis of *Paradigm Plus*. This missing information is the category of a class (abstract or concrete; all classes are considered concrete), the semantic kind of a method (constructor, destructor, selector, iterator, or modifier; no methods are considered iterators or modifiers), complete discrimination of aggregation and association (as described above), call compatibility of parameter lists, and delegation of method calls (this is not visible from header files).

Some of this information is usually difficult to compute — even impossible in the general case. This is true for the semantic kind of methods, discrimination of aggregation and association, and call delegation. Heuristic methods could handle the most frequent cases, though.

# 5 Prolog representation

## 5.1 Source code

We represent the C++ header files by PROLOG facts. As an example, the class declaration

```
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).
attribute (zPane, curDisp).
operation(virtual, selector, zPane, show,
          public, "int,", "void").
```

The information that is not reliably available from *Paradigm Plus* will have to be ignored (using PROLOG wildcard variables) in the pattern rules. For instance, the above `show` is called a selector although it is a modifier. The encoding used for these facts is

`class(Category, Name)`: There is a class `Name` of the given `Category` (`abstract` or `concrete`).

`attribute(Class, Name)`: There is a data member `Name` in `Class`.

`operation(Category, Kind, Class, Name, Scope, Params, ReturnType)`: There is a method `Name` of visibility `Scope` (`public`, `protected`, or `private`) in `Class`. No reliable results for `Category`, `Kind`, `Params`, and `ReturnType` will be available from *Paradigm Plus*; therefore, these items should be ignored later on.

`aggregation(Class1, Multiplicity1, Class2, Multiplicity2)`: There is an aggregation of `Class2` in `Class1`, i.e., `Class1` contains either one data member (`Multiplicity2` is `exactlyone`) of type `Class2` or an array of such members (`Multiplicity2` is `many`).

`association(Class1, Class2)`: There is a reference to a `Class2` instance in a `Class1` instance.

`inheritance(Class, Subclass)`: `Subclass` is a subclass of `Class`.

## 5.2 Patterns

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. This OMT object model is converted into the following PROLOG rule:

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

This rule describes necessary but not sufficient properties of classes to form an Adapter pattern instance. Gamma's Adapter pattern demands that there exists a delegation from the method `Adapter::Request` to `Adaptee::SpecificRequest`. However, because the structural analysis of *Paradigm Plus* cannot extract delegations, the delegation must not be modeled in our PROLOG rule or else the rule could never be matched. 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. Besides the above-mentioned structural properties that remain unchecked in *Pat*, there are also further semantic and pragmatic aspects in a pattern that cannot be detected reliably by automated tools. Similar restrictions apply for the PROLOG rules of the other design patterns.

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 only and ignore the rest. Second, classes are checked for inequality with clauses like `Target <> Adapter` etc. to avoid senseless matches.

Here are the PROLOG representations of the other four structural design patterns. You may want to skip the rest of this section if you are not familiar with design patterns.

A **Bridge** consists of four classes: abstraction, refined abstraction, implementor, and concrete implementor.

```
bridge(Abstr,RefAbstr,Impl,ConcrImpl):-
  class(_,Abstr),
  class(concrete,RefAbstr),
  class(_,Impl),
  class(concrete,ConcImpl),
  operation(_,_,Abstr,Op,_,_,_),
  operation(_,_,Impl,OpImpl,_,_,_),
  operation(_,_,ConcrImpl,OpImpl,_,_,_),
  inheritance(Abstr,RefAbstr),
  inheritance(Impl,ConcrImpl),
  aggregation(Abstr,exactlyone,
              Impl,exactlyone).
```

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

A **Composite** consists of three classes: the component superclass, a leaf subclass (usually there are several of these), and the composite subclass.

```
composite(Cpnt,Leaf,Compos):-
  class(_,Cpnt),
  class(concrete,Leaf),
  class(concrete,Compos),
  operation(_,_,Cpnt,Op,_,_,_),
  operation(_,_,Leaf,Op,_,_,_),
  operation(_,_,Compos,Op,_,_,_),
  operation(_,_,Cpnt,Add,_,_,_),
  operation(_,_,Cpnt,Remove,_,_,_),
  operation(_,_,Cpnt,GetCh,_,_,_),
  operation(_,_,Compos,Add,_,_,_),
  operation(_,_,Compos,Remove,_,_,_),
  operation(_,_,Compos,GetCh,_,_,_),
  inheritance(Cpnt,Leaf),
  inheritance(Cpnt,Compos),
  aggregation(Compos,exactlyone,Cpnt,many).
```

This rule ignores the realization of `Compos::Op` as a loop of `Op` calls for all children and ignores the semantics of `Add, Remove` and `GetCh` — all because no such semantic 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. In such cases, we drop all `operation` clauses from the rule.

A **Decorator** consists of four classes: the component top class with a concrete component subclass and a decorator subclass; the latter has one or several further subclasses called concrete decorators.

```
decorator(Cpnt,ConcrCpnt,Deco,ConcrDeco):-
  class(_,Cpnt),
  class(concrete,ConcrCpnt),
  class(_,Deco),
  class(concrete,ConcrDeco),
  operation(_,_,Cpnt,Op,_,_,_),
  operation(_,_,ConcrCpnt,Op,_,_,_),
  operation(_,_,Deco,Op,_,_,_),
  operation(_,_,ConcrDeco,Op,_,_,_),
  inheritance(Cpnt,ConcrCpnt),
  inheritance(Cpnt,Deco),
  inheritance(Deco,ConcrDeco),
  aggregation(Deco,exactlyone,
              Cpnt,exactlyone).
```

This rule ignores the delegations from `Deco::Op` to `Op` of the decorator's aggregated component and the implementation of `ConcrDeco::Op` as a call to `Deco::Op` plus some added behavior.

A **Proxy** consists of three classes: a real subject class, its proxy class and their common subject superclass.

```
proxy(Subj,RealSubj,Proxy):-
  class(_,Subj),
  class(concrete,RealSubj),
  class(concrete,Proxy),
  operation(_,_,Subj,Op,_,_,_),
  operation(_,_,RealSubj,Op,_,_,_),
  operation(_,_,Proxy,Op,_,_,_),
  inheritance(Subj,RealSubj),
  inheritance(Subj,Proxy),
  association(Proxy,RealSubject).
```

This rule ignores the delegation from `Proxy::Op` to `RealSubj::Op`.

## 6 Implementation details

The *Pat* system was developed with the *Paradigm Plus 2.01* ooCASE tool [10] and the *Visual Prolog 4.0 Beta (Professional Version)* compiler system [9].

The programs `P2prolog` and `D2prolog` are written in the BASIC dialect provided by *Paradigm Plus* and are executed directly by *Paradigm Plus*. Therefore they have direct access to the repository. The PROLOG rules and facts generated by the programs are written into text files to be consulted by *Visual Prolog*.

These rules and facts files are complemented by another file containing declarations and the generic query that starts the search for the pattern instances. This file is compiled into executable code, the other two are consulted at run time.

The executable 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 for instance

```
\adapter{zchildwin}{zpane}{zdisplay}
```

These pattern instance candidate lists are then filtered for duplicates. A file with suitable definitions for the LaTeX macros is used to convert the resulting instances into graphical OMT form (Figure 1 is an example) to provide a basis for a reverse-engineered design document.

# 7 Evaluation

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

1. What fraction of all 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.

The first two questions are answered in terms of the information retrieval quality measures called *precision* and *recall* [5]. Assume that *Pat* outputs $C$ pattern instance candidates after the automatic postprocessing. Assume further that the design analyzed actually contains $I$ true pattern instances and that $F$ of them are found by *Pat*. Then $precision = F/C$ and $recall = F/I$. Furthermore, we measured the execution time needed for the automatic search and the time needed for human filtering of the results to remove false positives.

## 7.1 The benchmarks

Four different sets of classes were examined: Network Management Environment Browser (NME), Library of Efficient Datatypes and Algorithms (LEDA, [8]), the zApp class library (zApp, [6]), 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.

## 7.2 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 the tables gives for each design pattern the number of pattern instances found by the search mechanism ("found") and the number of these that were not spurious ("true"). Below that you find 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.

Perfect recall occurs *although* our pattern rules could have missed some pattern instances because the structural analysis may mistake some aggregations (implemented by pointers) for associations. However, we have checked[1] that in our four benchmarks none of these cases would reveal another correct pattern instance. This seems to be a result of good programming style. Recall would have been below 100 percent had more aggregations been realized using pointers instead of arrays in the benchmarks' classes.

Because our pattern rules do not represent sufficient conditions for pattern instances, precision is not perfect. Some constructions will be detected incorrectly as pattern instances because they lack required properties that were not tested.

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

[1] The check was made by re-running all of the 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 more correct pattern instances than the output obtained without the rule.

| | classes | attrib. | operat. | aggr. | assoc. | inherit. | kByte facts |
|---|---|---|---|---|---|---|---|
| NME | 9 | 34 | 131 | 0 | 10 | 6 | 13 |
| LEDA | 150 | 501 | 4084 | 91 | 151 | 67 | 243 |
| zApp | 240 | 1176 | 3590 | 143 | 155 | 145 | 205 |
| ACD | 343 | 1506 | 2879 | 457 | 461 | 191 | 204 |

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.

| | NME | | LEDA | | zApp | | ACD | |
|---|---|---|---|---|---|---|---|---|
| | found | true | found | true | found | true | found | true |
| *Adapter* | 2 | 1 | 1 | 0 | 20 | ≈12 | 150 | ≈69 |
| *Bridge* | 0 | 0 | 59 | ≈10 | 7 | 0 | 0 | 0 |
| *Composite* | 0 | 0 | 6 | 0 | 0 | 0 | 0 | 0 |
| *Decorator* | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |
| *Proxy* | 0 | 0 | 1 | 0 | 1 | 0 | 17 | 0 |
| Recall | 100% | | 100% | | 100% | | 100% | |
| Precision | 50% | | ≈14% | | ≈43% | | ≈41% | |
| Prec. deleg. | 100% | | ≈53% | | 100% | | 100% | |
| Runtime (sec) | 1 | | 2 | | 3 | | 36 | |

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 running under Microsoft Windows 95.

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 using a relaxed rule (without `operation` clauses) as described in the modeling section.

### 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.

### ACD

ACD is a large project and created so much output that we were unable to check correctness completely. Instead, we relied on common sense judgement from the class names combined with another plausibility check: In the case of the Adapters we assumed that exactly those are correct where the name of `Request` is equal to the name of `SpecificRequest`. In the case of proxies we drew conclusions from the class names alone; no Proxy seemed to be in ACD. Evaluating the solutions for ACD in this manner took 30 minutes.

## 8 Conclusion

Automated search for instances of the five structural design patterns Adapter, Bridge, Composite, Decorator, and Proxy is feasible. The approach we propose, the *Pat* system, builds on structural analysis capabilities of a commercial ooCASE tool and the search capabilities of PROLOG, making the implementation simple and efficient.

Often all design pattern instances that are present are recovered from the C++ source code (i.e., recall is 100 percent). In addition, the *Pat* output contains a number of false positives. In our four benchmark applications, detection precision is between 14 and 50 percent. This precision is acceptable. It would be much higher if *Pat* could also check for correct method call delegations. This would require that the structural analysis detects such delegations. Precision would then be between 53 and 100 percent. The remaining false positives, if any, can be sorted out with a modest amount of manual work (typically a few minutes per pattern instance).

We conclude that the *Pat* system is a fast and simple way to recover design 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 could also try to apply the approach presented here to a larger number of more detailed patterns, probably using a staged recognition approach to achieve high precision and recall. We should also explore how well a similar approach can be used to detect behavioral patterns instead of structural ones.

## References


[1] K. Beck, J.O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlissides. Industrial experience with design patterns. In *18th Intl. Conf. on Software Engineering*, pages 103–114, Berlin, March 1996. IEEE CS press.

[2] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[3] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–83, May 1994.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] H.S. Heaps. *Information Retrieval*. Academic Press, 1978.

[6] Inmark Development Corporation, Mountain View, CA. *zApp Programmer's Guide*, 1994.

[7] Andrew Koenig. Patterns and antipatterns. *Journal of Object Oriented Programming*, 8(1):46–48, March 1995.

[8] Stefan Näher. *LEDA User Manual Version 3.0*. Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany, 1992.

[9] Prolog Development Center, Brøndby, Danmark. *Visual Prolog 4.0 (Professional Version)*, 1996.

[10] ProtoSoft Inc., Houston, TX. *Paradigm Plus 2.01 Reference Manual*, 1994.

[11] Charles Rich and Richard C. Waters. *The Programmer's Apprentice*. Frontier Series. acm press, Addison-Wesley, New York, NY, Reading, MA, 1990.

[12] James Rumbaugh. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[13] Peter G. Selfridge, Richard C. Waters, and Elliot J. Chikofsky. Challenges to the field of reverse engineering. In *Proc. Working Conf. on Reverse Engineering*, 1993.