JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 5, No. 1, January–February 2006

An efficient tool for recovering Design Patterns from C++ Code

Marek Vokáč, Software Engineering Department, Simula Research Laboratory, Oslo, Norway

Design Patterns are informal descriptions of tested solutions to recurring problems. Most design tools have little or no support for documenting the presence and usage of patterns in code. Reverse engineering is therefore often required to recover Design Patterns from code in existing projects. Knowledge of what Design Patterns have been used can aid in code comprehension, as well as support research.
Since pattern descriptions are abstract and informal, they offer no algorithmic translation into concrete code. Some patterns prescribe class structures that are easy to recognize, while others lead to structures that are difficult or impossible to recognize. This work presents a tool that can recover five different design patterns from C++ code with high precision and at a speed of 3×10^6LOC/hr. This makes it suitable for analysis of large (multi-millon LOC) systems.

1 INTRODUCTION

Software Design Patterns, as first formalized by Gamma *et al.* [10], have become popular in the object-oriented software community. Some of the patterns have been incorporated into widely used architectures and frameworks. Examples of this are the Iterator pattern in the C# language (coupled to the `foreach` keyword), the Observer pattern in event-based user interfaces, and the Factory pattern in Microsoft COM, MFC and J2EE.

Recovery of Design Patterns from existing code is important in several situations. Code maintenance should be made easier if any patterns used during the design can be recovered. Also, empirical research on the effects of using Design Patterns is severely limited if it cannot make use of existing code bases.

Hopefully, the emergence of integrated development environments that fully support UML modelling and pattern application will reduce the need for reverse-engineering tools; however, maintenance of code designed without such support will remain significant for many years.

To support our ongoing research into correlations between the use of Design Patterns and defect frequency, we needed a tool to extract patterns from a large amount of C++ code. A survey of academic literature uncovered a number of recovery tools, but none of them were documented to be able to handle the patterns and code sizes we needed. We therefore created and validated our own tool for this

purpose.

The tool looks for structural signatures, i.e., class and method structures that result from the implementation of certain design patterns. We developed a semi-formal, graphic notation to describe Design Pattern structures in greater detail than the original diagrams given by Gamma *et al.*, but not so rigid as to overspecify and thereby miss recovery of implementations that are not identical to the "ideal" structure.

This paper is organized as follows: Section 2 discusses existing tools, as documented in the academic literature. Pattern structures and our diagrammatic notation for expressing them are described in Section 3. The goals and construction of our tool are in Section 4, and its performance, recovery and precision on a 500 000 LOC system are in Section 5. Section 6 concludes.

2 EXISTING TOOLS AND RELATED WORK

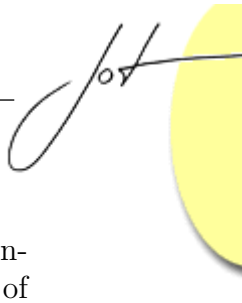
In this section we summarize related pattern-extraction tools. Our starting point is executable code, not UML designs or other specifications.

The justification for this starting point is as follows. A Design Pattern is a description, using prose and semi-formal diagrams, of a way to structure classes in a program. However, the ultimate expression of Design Patterns is in executable code, whether generated from a model or by hand, and the code is the ultimate reference (as opposed to UML models or other documents, which tend to become out of date if not used with good tool support, or rigorously maintained). Our survey of related work is thus restricted to tools that use code as their input.

It is in the very nature of Design Patterns that they are abstract, general prescriptions of solutions. Their translation into actual code necessarily involves judgement, and is not a task that can be performed mechanically without regard to context. This is especially true when analyzing existing code, the design of which was inspired by patterns yet did not have "compliance" with pattern specifications as its goal. This means that we should not expect any tool to recover all patterns with 100% precision.

An early work was by Kramer and Prechelt [16]. Patterns were drawn in an OMT design tool and translated into Prolog rules; source code was parsed using the Paradigm Plus tool and converted into Prolog facts. Then, queries were run to determine what facts matched the rules, ie., what patterns were present in the code.

The parsing tool had significant limitations. It did not extract information that would have been useful, such as whether a method is a constructor, or whether a class is abstract or concrete. Further, the tool looked only at header files, and thus had no information on the function call hierarchy. This made recovery of patterns more difficult, since essential parts of the signature of a pattern that depended on these concepts could not be expressed. Nevertheless, the tool achieved reasonable



recall and precision rates on source code of moderate size (150–350 classes).

Florijn et al. [8] constructed a tool that was integrated in a Smalltalk environment, which supported development at several abstraction levels, including that of Design Pattern. With this tool, it is possible to create new classes as instances of patterns, connect existing classes with patterns and roles, and check whether pattern invariants are being upheld by classes in the code. The real-life test example cited involves about 150 classes.

Bansiya [6] presented a tool in 1998 that used the Microsoft MSVC compiler to parse the code, and relied on the "browse information" database generated. The tool seems to have been based on a structural rule-based matching method, but there is little information on its precision, or its ability to handle large systems.

Antoniol *et al.* used a different approach [3]. The code was analysed in terms of tuples of classes and their relations, and metrics were used to reduce the number of candidate classes and avoid the combinatorial explosion.

The metrics calculated were the number of attributes and operations, further divided into public, private and protected; the number of association, aggregation and inheritance associations for each class; and the total numbers of attributes, methods and relations.

Using the metrics, classes were eliminated that did not show the "right" signature for the pattern in question, such as inheritance or associations that are part of the given patterns' structure. In the final stage, the exact pattern signature was sought among the classes that survived the metrics selection process.

Their method was tested on public-domain and industrial software in the 5 000–50 000 LOC range. It performed well in terms of speed (minutes), but did not achieve high precision. In the industrial software analyzed, there were so few pattern instances that it was difficult to judge the precision of the process.

The method has been further developed and was last presented in 2001 [2]; however, the precision of the system is still quite low (3%–50%).

A potential weakness is the fact that some metrics, such as inheritance, are not reliable indicators of a pattern structure. For instance, the Template Method pattern specifies a base class with a template method, and concrete subclasses that override and implement the primitive operations. However, this whole hierarchy might well be embedded in a larger context, so that the abstract base class is itself a subclass of one or more classes. Thus, it would be incorrect to conclude that a class has to be at the top of the hierarchy to be at the root of the Template Method pattern. However, it is correct to require that the subclass in the Template Method really be a subclass; though it may be deeper than a direct subclass of the base class.

Keller et al. [15] designed a pattern extraction mechanism for use within a larger reverse-engineering system. Based on the structure of the pattern, they constructed an appropriate query that searched their metadata repository for corresponding occurrences. Performance figures are not given, but one of the systems tested consisted

of more than 470 000 lines of code. This work is a continuation of earlier work by Schauer and Keller [18].

An interesting point is that there are some patterns that are difficult or impossible to recover, because their structural signature is weak or variable. The Bridge pattern is cited as an example; while the original definition of Bridge specifies a certain combination of inheritance and aggregation associations, in practice these are not always followed. Relaxing the criteria to accommodate this causes large numbers of false positives, while keeping them strict, means that design constructs intended to work like Bridge are missed. We believe this is an inherent property of the pattern, rather than of any particular approach to recovery.

Albin-Amiot et al. [1] have proposed and implemented in prototype a system that searches for, and recognizes, pattern signatures in Java code. Their system relies on a constraint solver, which attempts to solve the problem given by matching the actual code structure to the structure of the design pattern. Their system can recognize partial or distorted implementations and can even recommend possible refactorings.

They do not state running times or give examples of the application of their system to non-trivial systems. However, they state that they intend to test their system on the package JHotDraw; a companion paper [12] states that this package "... contains more than 125 classes and identifies several design patterns".

Perhaps the most promising method to date for design pattern recovery from large-scale projects is that of Balanyi and Ferenc [5]. They use a reverse engineering framework to convert C++ code into metadata (termed Abstract Semantic Graph), and express patterns in an XML-based language. They then perform a multi-step algorithm to identify candidate class structures, match them to the pattern descriptions, and filter out mismatches. One of their major contributions is to look at information from function bodies, such as function calls and object creations, in addition to the more traditional static structure.

They seem to be the only group so far that has tested a method on million-line code collections, and they give running times for extraction of different patterns. In two large projects of 1 200 KLOC and 1 500 KLOC size, they found about 440 and 520 pattern instances in total, in five and nine hours' running time. However, the time spent on code parsing is not given. They cite fairly low rates of false positives (falsely indicating the presence of a pattern), but do not give any evaluation of false negatives (failing to identify a pattern that is actually present).

3 PATTERN STRUCTURES AND DESCRIPTIONS

As a first level of abstraction from concrete code, we adopted an entity-reference model, similar to the class/relation tuples used by Antoniol et al. [3]. An *entity* is anything that is not a language keyword or operator, i.e., any named class, variable,

method, macro or parameter. A *reference* links two entities, and is of a certain type, such as "Calls", "Is declared by" or "Overrides". Entity types also express attributes such as "virtual" or "public". During the parsing step, the code to be analyzed is reduced to a set of entities interconnected by multiple references.

Using these two concepts, we constructed a graphic notation to define Design Patterns at a sufficient level of detail for the analysis. In our view, formal, rigorous definitions as used by France et al. [9], are not suitable in this context, since our aim is to be able to recover patterns that have been applied imprecisely. Further, the "rigorous" application of a pattern may simply not be the best design decision in every case, and we expect software designers to exercise judgement.

In its first version, our tool recovers the patterns Observer, Decorator, Factory, Singleton and Template Method. Our concepts for pattern structures, and the structural signature left imprinted in the final code, are discussed for the Template Method and Observer patterns. They are good illustrations of both simple and more complex problems. We should note that during most of its running time, our tool performs parsing and preparatory indexing that is independent of the actual patterns to be extracted. Adding new patterns is thus fairly easy, and does not require the whole parsing process to be re-run.

Template Method

This is a relatively simple Design Pattern. It is used in situations where the major flow of a process or algorithm is given and is reusable, but there may be differences in the detailed steps. In this case, the algorithm is implemented in a base class, and calls overridable methods for the detail steps. These methods may be abstract or have a default implementation in the base class. Derived classes provide their own implementations of these methods as required, thus customizing the algorithm to their particular needs.

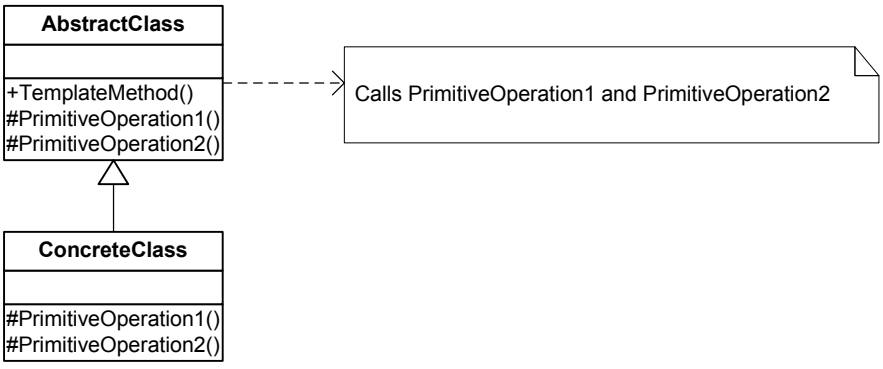


Figure 1: Structure of Template Method, from [10].

Gamma *et al.* use an informal, UML-like notation with explanatory prose to describe the structure of this design pattern, as shown in Figure 1. We see that

it involves an abstract class and a concrete class, and one or more "primitive" operations that are called by the template method in the base class; these methods are overridden in the concrete class.

Some elements of this structure diagram should not be taken too literally. For instance, the inheritance may span multiple levels; there may be more than one template method in the abstract class, and concrete classes will not always override all of the primitive operations.

Our translation into the entity/reference model modifies and formalizes the structure to take into account these properties.

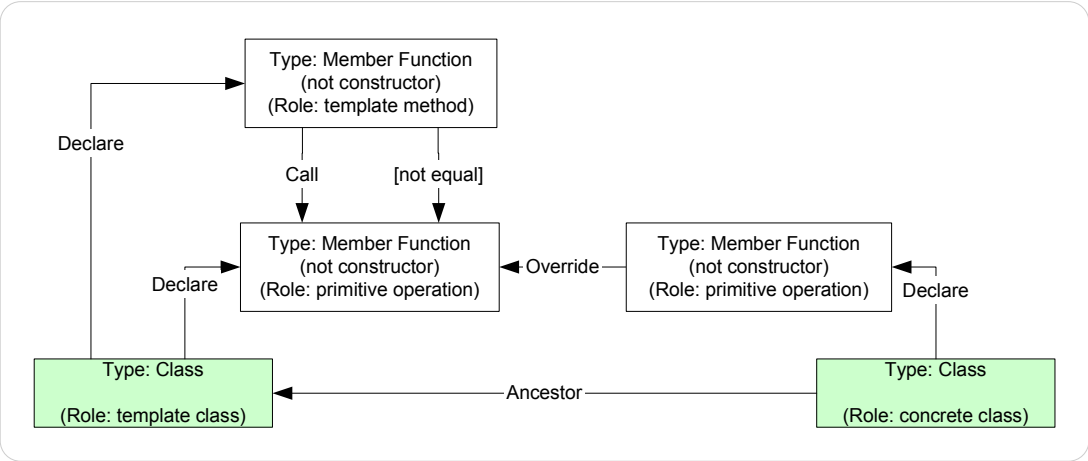


Figure 2: Structural signature of Template Method

In our notation, boxes represent entities such as classes or methods, and arrows between the boxes represent references. The top line of text in a box denotes the entity type, and the bottom line denotes the role that the entity plays in the pattern. Restrictions are placed in the middle of the box. Similarly, the text on a reference defines the reference kind.

The signature diagram for the Template Method pattern is shown in Figure 2, and expresses the expected structural signature in a more complete way than the original notation. Our diagrams can be translated by hand or semi-mechanically into executable queries against an entity-reference database derived from actual code.

The two main entities in the signature diagram are the template class and the concrete class (shaded boxes). The template class is a (possibly indirect) ancestor of the concrete class, and may itself be a subclass of some other class (this is not specified and thereby not restricted). The template class declares at least two member functions, which have the additional restrictions of not being the constructor. We require a "call" reference from the template member function to one or more primitive operation functions, and we require the functions to be distinct.

The concrete class must declare one or more member functions that are also not constructors, and which override the template class methods that act as primitive operators.

Observer

The Observer pattern illustrates how it is possible to implement a design pattern in at least two, radically different ways. The core concept is simple: a class (the observer) may observe changes to another class (the subject), and react to those changes in some way; there may be multiple types and instances of Observers for any Subject.

The differences in implementation are related to the way in which notifications from subjects to observers are implemented. In the classic, closely coupled model, described in the original pattern, each observer keeps track of its subjects and directly notifies them of changes. The structure described in [10] and shown in the upper part of Figure 3 portrays the closely coupled model.

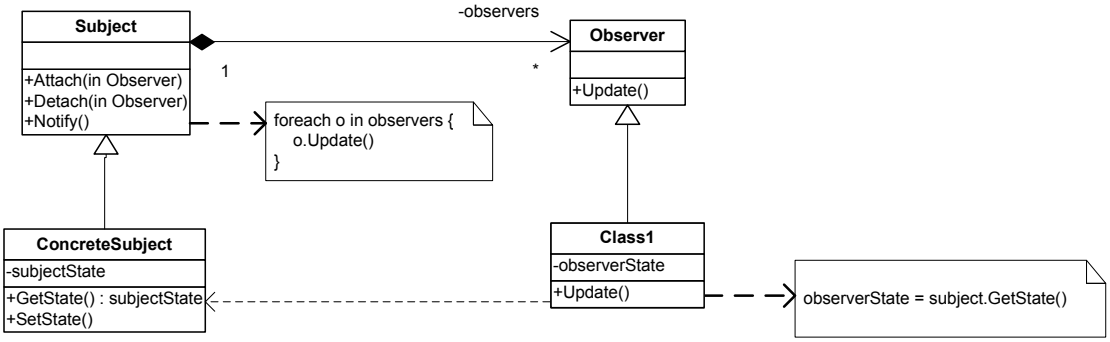


Figure 3: Structure of Observer, from [10].

In this model, there are base classes for the concepts of Subject and Observer, and concrete classes derived from them. The subject class maintains references to its current set of observers, and the observers call the subject directly to retrieve information. It is explicitly specified that the subject state is not part of the notification message, but must be retrieved separately.

However, the Observer pattern is also known under the name "Publish-Subscribe", which points us to a different, loosely coupled model for its implementation. In this model there is a third party, a broker, that keeps track of subjects, observers and notifications. The concrete subject class is relieved of this task and does not have to be a subclass of a common "subject" base. The observer class will, in most instances, still have some relation to the subject (it is, after all, interested in what happens to the subject) but its registration interaction will be with the broker.

There are other ways of communicating as well. [7] describes several possible schemes. The Reactor pattern [19] describes an inter-process variant with Singleton

multiplexers and demultiplexers in the sending and receiving processes. A similar scheme, though without the multiplexing, appeared in the OMG Event Service Specification [11]. It involves a proxy publisher and a proxy subscriber to hide the process boundary, and an event channel to transfer the notifications. Varying degrees of buffering, asynchronicity and further decoupling are also possible.

The tightly coupled and loosely coupled observer implementations have quite different signatures, as seen in Figure 4. The loosely coupled model involves a total of nine entities and 12 references, while the tightly coupled model uses six entities and seven references. Most importantly, we will not recognize a tightly coupled observer while looking for a loosely coupled one, and vice versa.

This range of potential implementations is a fundamental and intended strength of the concept of Design Patterns; the designer is able to implement a pattern in the way that best fits the context and problem at hand. However, by the same token, that same range of potential implementations makes it much more difficult to reconstruct patterns from code, unless there are some "standard" implementations whose structural signatures will match most of the actual usage. As a result, it must be made clear regarding a particular reverse-engineering tool what variants of patterns it is designed to recover.

Language-specific features

Different languages implement concepts in different ways. One concept that is central to many Design Patterns is that of aggregation, generally implemented as a collection; a set of objects or references to objects. The Composite pattern is a typical example, where each composite object may contain or reference any number of child objects. To recognize a Composite, we therefore need to start by recognizing a collection or aggregation relation.

In C++ in general, this is almost impossible. A C++ pointer can point to one object, or it can point to an array of objects; but there is no way to know which without analyzing the code in great detail.

Before the advent of the Template mechanism and the Standard Template Library [20], every developer or group had to implement its own data structures, so there was no generally accepted standard (unlike Java and its Collections hierarchy of objects). To make matters even worse, macros in C++ can be used to perform almost any textual substitution, making parsing almost impossible. An example of this is a case where a Container base class is inherited through a macro that defines a subclass; correctly parsing and recognizing this as a collection is beyond most tools [4].

In other languages the situation is simpler. Java has a well-defined set of container classes [21], and also does not have the fine division between embedded objects, references and pointers present in C++. Thus, it is much easier to detect collections with reasonable accuracy in Java.

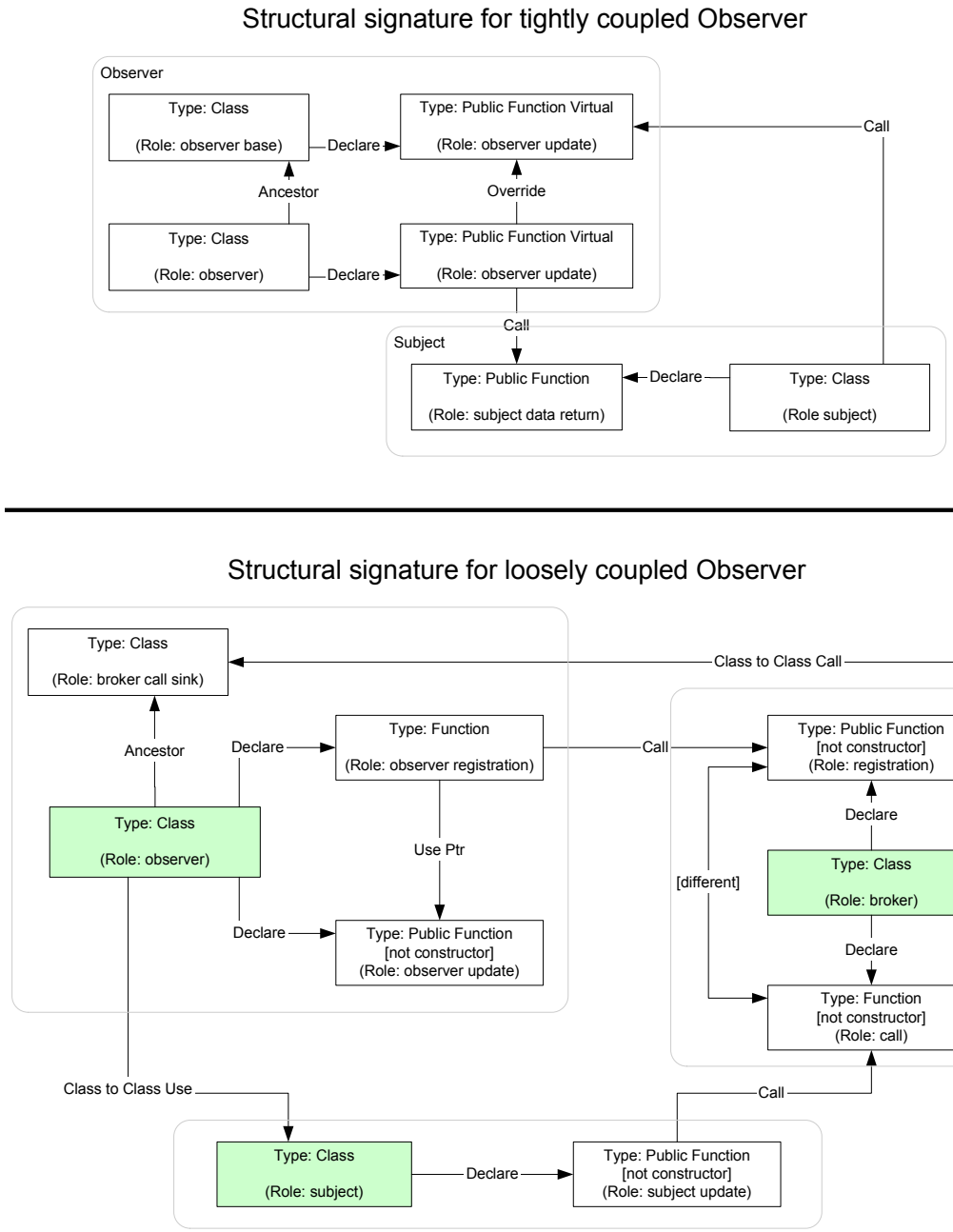


Figure 4: Structural signatureof Observer, tightly and loosely coupled.

In C# there is an `IEnumerator` interface in the standard library set, which encourages developers to implement their own iterator concepts wherever appropriate. It is coupled to the `foreach` keyword in the language [17], yielding an easily recognized, simple and elegant syntax for traversal of any array or collection.

Our tool and the exact structural signature for the chosen patterns were designed in the context of C++, and to some extent adjusted for the programming style used in the software analyzed in the case study. However, the underlying concepts are

transferable to other, similar languages.

4 TOOL GOALS AND DESIGN

We formulated the following goals for a pattern recovery tool:

- It must be possible to describe a structural signature of a pattern, and to extract from a C++ code base the set of classes that correspond to this signature.
- The tool must be flexible, because some design patterns have complicated signatures that are not easily expressible as simple rules. It must be reasonably easy to express a structure, so that the specification can be checked, revised and debugged.
- The tool must scale extremely well, and be able to handle amounts of code in the 10^8 LOC range with running times in the order of hours, or at least within a weekend. Preferably, much of the processing time should be spent on data preparation that is independent of what patterns are being sought, so that the addition of new patterns does not force a re-run of the whole process. If possible and when necessary, the method should lend itself to optimization using parallel hardware (storage, CPU) or clustering.
- The input data should be in the form of "untreated" code files, i.e., whatever structure the source project is already in. Output should be in a form that is easily transferable to statistical packages for further analysis.

Tool construction

Due to the lack of a satisfactory, off-the-shelf tool, we decided to construct our own. The tool was built using several sub-components to handle different stages of the process. During our research, it satisfied all the goals, with the exception of parallel processing, which was not pursued due to lack of suitable hardware.

Figure 5 shows an outline of how our tool is constructed. The first stage consists of extraction of code snapshots from a Version Control System (VCS). If we only want to perform a single analysis of a system, this stage can be performed manually (and does not require the presence of a VCS at all). However, for analyses of trends over time we need to take multiple snapshots, one for each time point we wish to analyze.

The resulting snapshot is a collection of C++ source files, both header and class body. The method makes no assumptions as to the location of classes or correspondence between classes and files, but since the VCS generally works at the

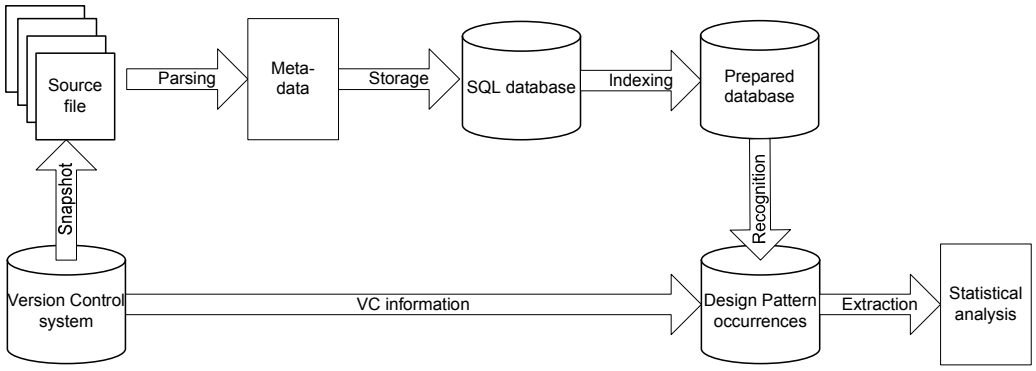


Figure 5: Outline of the structure extraction tool. The process starts at the lower left.

file level, analysis becomes simpler if the convention of "one class, one file pair" is followed.

The source files are parsed using a commercial tool called UNDERSTAND FOR C++ [14]. This tool is flexible and scalable, and parses C++ code into metadata. Its main shortcoming is that it does not currently handle templates (generics), and this causes some problems later on with design patterns that rely on collections.

The output from UNDERSTAND is a file, in proprietary format, that contains mainly two kinds of data: entities and references. An entity is any named concept that is not a keyword, for instance a class, variable, type or file. A reference is a link between two entities, such as "declares", "calls", or "dereferences". Both entities and references are classified into predefined kinds.

The "storage" stage of the processing transfers the entity and reference data from the proprietary UNDERSTAND format into an SQL database, without materially changing the data. The database contains the following: (i) tables that correspond to the entity and reference concepts, and (ii) supporting tables for entity and reference kinds, and links to files and metrics.

The tables are then indexed. Insertion of large amounts of data is much faster if there are no indexes, so index generation is postponed until data loading is completed. During the indexing process, some extra reference kinds are computed in addition to those generated by Understand. To optimize performance, the database schema is slightly denormalized by converting some relations (such as whether an entity is a class member function) into attributes directly in the entity table.

At this stage, the database is ready to perform recognition of structural design patterns. The recognition is actually done by a series of SQL statements designed to look for the given structure; a structural signature translates quite readily into one or more `select` statements. Complicated or irregular structures may be recovered by chaining multiple SQL statements in a stored procedure. This provides more expressive power than single statements, and also provides an opportunity

for optimizing performance where necessary. The results are stored in intermediate tables.

Metadata is also extracted from the VCS, in the form of information about submitted changes. This information is added to the database with the code metadata, which enables us to combine it with the class structure and see how it evolved.

Finally, the results are condensed and transferred to a statistical package for further analysis. An example of such analysis is to generate indicator variables for each pattern, and use logistic regression to look for correlations between changeability and pattern membership for classes.

5 TOOL PERFORMANCE, RECOVERY AND PRECISION

Performance

The tool was used to analyze a Customer Relationship Management system, written in C++. Table 1 gives some size metrics for the product.

Metric	Value
Total lines	1 114 092
Lines of code (LOC)	505 367
Number of classes	2 047
Number of code files	2 809
Number of methods	30 823
Declarative statements	150 685
Executable statements	194 625

Table 1: Descriptive metrics for SuperOffice CRM5

The system consisted of approximately 3 700 files (including scripts, resources, graphics, etc), totalling 90 MB. However, as our study intended to analyze the system's evolution over time, a total of 153 weekly snapshots were extracted from the VCS. The snapshots grew slightly over time as code was added to the system; in sum, they consisted of about 500 000 files totalling 14 GB.

The pattern extraction tool was applied to each snapshot, to parse all the code, load the metadata into the SQL Server, index it and extract Design Patterns. Typical running times are shown in Figure 6. Each snapshot had its own, preallocated empty database in the server. Analysis of all the snapshots, 76×10^6 LOC, took slightly less than 26 hours on a 2.8 GHz PC with 4 GB of RAM (max. 800 MB actually in use) and standard IDE disk drives.

Table 2 summarizes the frequency of occurrence of the patterns in the code. Participation of a class in a pattern is not a simple 1:0 or 1:1 relation, as it is quite possible for a class to participate in multiple patterns. Our tool considers each

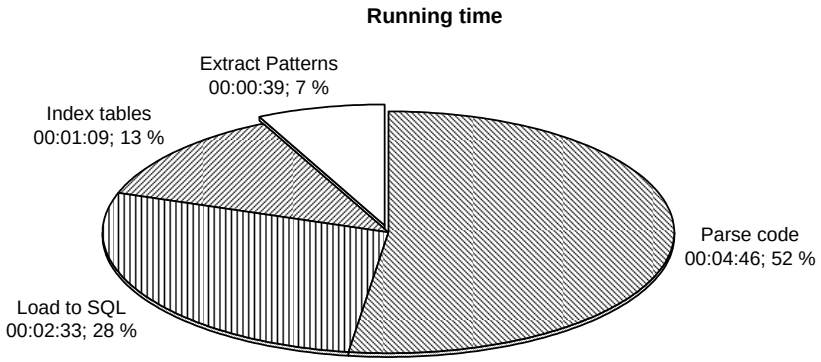


Figure 6: Length and phase distribution of running times for the Pattern recognition tool, on 500 000 LOC.

pattern separately, but since the entities that represent classes in the metadata have unique identifiers, it is easy to identify classes that participate in more than one pattern.

Patterns	Occurrences	%
No Pattern	183 634	77.5 %
Factory	20 237	8.5 %
Singleton	3 331	1.4 %
Observer	16 061	6.8 %
Template Method	5 381	2.3 %
Decorator	1 513	0.6 %
Factory + Observer	612	0.3 %
Factory + Singleton	2 279	1.0 %
Observer + Singleton	2 390	1.0 %
Observer + Template	953	0.4 %
Factory + Observer + Singleton	485	0.2 %

Table 2: Frequencies of pattern occurrences, and percentage of code covered by the patterns

Error rates

Since a Design Pattern is an informal specification of a recommended structure, it will be translated into program code differently in different projects. Any discussion of error rates must, therefore, be seen in relation to the application of the method to a particular set of software artifacts.

It is also necessary to define exactly what we mean by an instance of a certain Design Pattern. Consider Factory as an example: one Factory class may have methods to create one or more Product classes. Should we count every combination

of Factory and Product as an instance, or should one Factory class count as a single instance, regardless of the number of products? Similar situations occur in most patterns, since they specify relationships between multiple classes.

In our evaluations, we have adopted a simple definition, according to which one Factory class counts as a single instance. Similarly, we count one instance of the Observer pattern for every Subject; one Template Method for each template method; one Decorator for each Decorator class; and one Singleton for each Singleton class.

It was necessary to adjust the pattern-recovery tool for two particular patterns: Observer and Decorator. The Observer pattern can be implemented in two different ways, as discussed in section 3. The "classic" structure specifies that each Subject should keep track of its Observers directly, using a collection of object references. An alternative approach is to use a generalized message broker to handle the relations between subjects and observers, and this is the approach adopted globally in the studied code. The tool was adapted to this Subject/Broker/Observer structure.

For Decorator, a related problem exists, that of aggregation (see section 3). The tool was adapted to the kind of aggregation generally used in studied code.

False positives

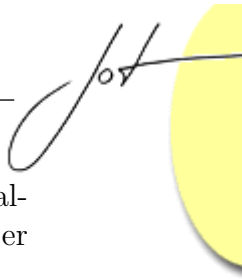
The rate of false positives was determined by manually examining all detected instances of all patterns. The results for all patterns are given in Table 3.

Pattern	Instances	False	Error rate
Factory	53	8	15.1 %
Singleton	45	0	0.0 %
Observer	20	3	15.0 %
Template Method	163	2	1.2 %
Decorator	9	0	0.0 %

Table 3: False positives

Most of the false positives identified for the Factory pattern were classes that used inner (nested) classes. From the outside this looks like a Factory instance, since the inner class is created by the outer class and by nothing else. However, this usage does not correspond to the intent of Factory, so it was classed as a false positive. It is feasible to add the condition "product class must not be nested within factory class" to the structural signature for Factory in a refined version of the rules.

In the Observer cases, we are dealing with a "loosely coupled" version of Observer, in which all notifications are handled by a centralized message broker class. This differs somewhat from the classical, simple Observer pattern, in which every Subject keeps track of its Observers separately. There are other forms of interaction through the Message Broker than just those that accord with the Observer pattern,



and the three false positives are such cases. Since the structure of the pattern is already quite complex, further refinement is difficult without risking a greater number of false negatives.

The structure for the Template Method allows for multiple levels of inheritance, and does not require more than one call to an underlying, virtual primitive method to consider the caller a parent method. It is a matter of taste whether one would want to tighten the definition, i.e., to demand that there is more than one implementation of the primitive method, or more than one primitive method for each template method. The number of false positives would most probably decline, but the number of false negatives might increase.

Decorator has a somewhat problematic structure, in that it contains an aggregation (the set of Decorators for one decorated class) that can be implemented in many ways. The signature was adapted to the known kind of aggregation implementation in this code, and so we should not take the zero error rate as being guaranteed in a different setting. Also, false negatives are more probable for this pattern.

Singleton is a pattern that is fairly easy to recognize, and so the low false positive rate is as expected.

False negatives

To determine the actual rate of false negatives, we would need to evaluate all classes and find all cases in which a class participates in a pattern but has not been detected by the tool. With more than 2 000 classes this is not realistically possible.

Instead, we chose to evaluate a random sample of classes, to get an estimate of the false negative rate. Judging from prior knowledge of the code (the author previously served as a developer and architect for the studied product), a low rate of false negatives was expected. To calculate the necessary sample size, the following criteria were set:

Required power: 90%. Hypothesized proportion (false negative rate): 20%. Alternative proportion (rate to be tested for): 10%.

This yielded a required sample size of 109.[1] To guard against randomly choosing classes that are trivially small or otherwise nonrepresentative, the actual sample size was increased to 125.

Classes were chosen using a uniformly distributed random number generator. The chosen classes covered all major modules of the program. Figure 7 shows the distributions of class sizes, for the full system and the sample.

Out of the 125 classes inspected, a total of nine were false negatives. Of these, one was part of a Decorator, one an Observer and the rest were Template Methods (six of the seven were actually part of the same instance of Template Method, missed

[1] MiniTAB [13] version 13.32 was used for all statistical calculations.

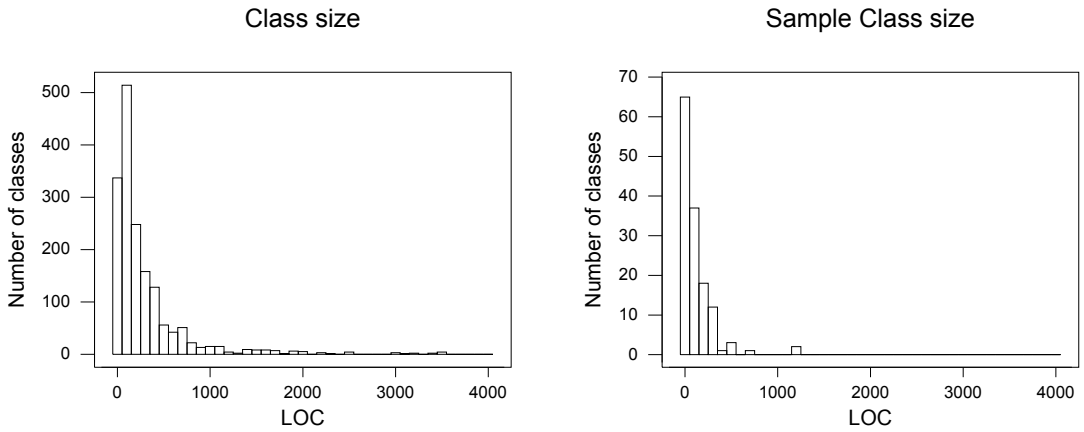


Figure 7: Histogram of class sizes, of the full system (left pane) and 125-class sample (right pane)

due to macros in the code that hid the virtual method declarations).

From these observations, we calculate the upper bound of a 95% confidence interval for the proportion of false negatives. The results are given in Table 4.

Pattern	N_{false}	95% CI	P
Factory	0	2.3 %	0.000
Singleton	0	2.3 %	0.000
Observer	1	3.7 %	0.000
Template Method	7	10.3 %	0.000
Decorator	1	3.7 %	0.000

Table 4: False negatives, from code sample

When interpreting these results, we must keep in mind that they apply to the studied code only. Given the number of possible ways to implement the structure described by a pattern, the validity of the tool must be tested for each new coding style.

6 SUMMARY AND FUTURE WORK

We have successfully built and validated a tool that efficiently recovers selected Design Patterns from C++ code. The tool was used to analyze a 500 000 LOC commercial system. During processing of historical data from the VCS, the tool evaluated a total of 76×10^6 LOC.

The tool is based on descriptions of structural signatures associated with the chosen Design Patterns. The signatures are described using semi-formal diagrams, which can be translated into queries mechanically, or hand-coded in the case of complicated or irregular structures. The code to be analyzed is parsed into metadata

using a commercially available tool, and this metadata is placed in a relational database where the queries are executed.

The tool has been empirically validated to have a high rate of recovery (few false negatives) and precision (few false positives). However, validation has been performed only on one major set of code. Since Design Patterns are usually not mechanically applied and translated into concrete code, it is necessary to revalidate the tool when applying it to new software.

Most of the tool's running time is spent on tasks that are independent of the specific patterns recovered. This, combined with its speed, means that adding further patterns is relatively straightforward, and can be done while using a full code base and not just trivial examples.

We plan to pursue the tool in four possible directions:

- Add more patterns—many of the patterns in [10] are candidates, though not patterns such as Bridge, which seem inherently to leave a very imprecise signature.
- Define a UML Profile for our pattern description notation, and use a code generator to transform pattern structure diagrams into SQL queries
- Validate the tool on more software—the Open Source community should be a good source of nontrivial C++ software.
- Extend the tool to other languages—since the first stage is to translate from C++ into an abstract entity-reference model, an equivalent parser for other languages can be substituted. While it may be unrealistic to find another parser with exactly the same output, the entity-reference model is so general that it should be easy to transform other formats into it.

REFERENCES

[1] H. Albin-Amiot, P. Cointe, Y. G. Gueheneuc, and N. Jussien. Instantiating and detecting design patterns: putting bits and pieces together. In *16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 26–29, San Diego, CA, USA, 2001. Ecole des Mines Nantes France.

[2] G. Antoniol, G. Casazza, M. Di Penta, and R. Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

[3] G. Antoniol, R. Fiutem, and L. Cristoforetti. Using metrics to identify design patterns in object-oriented software. In *Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*, pages 23–34, 1998.

[4] G. J. Badros and D. Notkin. A framework for preprocessor-aware c source code analyses. *Software-Practice & Experience*, 30(8):907–924, 2000.

[5] Zsolt Balanyi and Rudolf Ferenc. Mining design patterns from c++ source code. In *International Conference on Software Maintenance (ICSM'03)*, page 305, Amsterdam, The Netherlands, 2003. IEEE.

[6] J. Bansiya. Automating design-pattern identification. Dept. of Comput. Science, Alabama Univ., Huntsville AL, USA, 1998.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture*. Wiley, Chichester, 1996.

[8] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Ecoop'97: Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495. 1997.

[9] R.B. France, D.-K. Kim, Sudipto Ghosh, and E. Song. A uml-based pattern specification technique. *Software Engineering, IEEE Transactions on*, 30(3): 193–206, 2004.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[11] Object Management Group. Corbaservices: Common object services specification, 1995.

[12] Y.-G. Gueheneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Technology of Object-Oriented Languages and Systems, 2001. TOOLS 39. 39th International Conference and Exhibition on*, pages 296–305, 2001.

[13] MiniTAB Inc. Minitab 13.32, 2003.

[14] Scientific Toolworks Inc. Understand for c++, 2003.

[15] R.K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 226–235, 1999.

[16] C Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 208–215, 1996.

[17] Microsoft. C# programmer's reference: `foreach`, `in`, 2004.

[18] R. Schauer and R.K. Keller. Pattern visualization for software comprehension. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pages 4–12, 1998.

[19] D. C. Schmidt. Reactor: An object behavioural pattern for concurrent event demultiplexing and event handler dispatching. In *PLoP 94*, pages 529–545, 1994.

[20] SGI. Standard template library programmer's guide, 2004.

[21] Sun. Java api documentation: Interface collection, 2004.

ABOUT THE AUTHORS

Marek Vokáč completed his MSc degree in computer science at the University of Oslo in 1992, and is currently working on his PhD in the same field, at the Simula Research Laboratory in Oslo. He has been working as a professional software developer since 1985, on both minicomputers, PC's and client/server systems. His interests center on software engineering, software design, databases and design patterns. Combining research with practical, industrial work is a personal long-term goal.