Mining Design Patterns from C++ Source Code

Zsolt Balanyi and Rudolf Ferenc

Research Group on Artificial Intelligence, University of Szeged, Hungary
zsoca@rgai.inf.u-szeged.hu, ferenc@cc.u-szeged.hu

Abstract

Design patterns are micro architectures that have proved to be reliable, easy-to implement and robust. There is a need in science and industry for recognizing these patterns. We present a new method for discovering design patterns in the source code. This method provides a precise specification of how the patterns work by describing basic structural information like inheritance, composition, aggregation and association, and as an indispensable part, by defining call delegation, object creation and operation overriding. We introduce a new XML–based language, the Design Pattern Markup Language (DPML), which provides an easy way for the users to modify pattern descriptions to suit their needs, or even to define their own patterns or just classes in certain relations they wish to find. We tested our method on four open-source systems, and found it effective in discovering design pattern instances.

Keywords

Design Patterns, DPML, C++, UML, ASG, Schema, Columbus

1 Introduction

Design patterns [9] are micro architectures that have proved to be reliable, easy-to implement and robust. Hence they can be a measure of the quality of an object oriented software system. So a software system can be characterized among other things by the number of the design patterns used. Of course, one must fully understand the design patterns one would like to use because improperly used, they can result in unnecessarily huge class structures that can in the worst case even decrease the quality of the code.

There are three kinds of design patterns:

- *Creational* – these patterns are concerned with the creation of objects. They can decide the type of the object and its multiplicity. Here it is not enough to match the pattern structure because the real functionality is hidden in the function implementations. Though it is difficult to recognize these patterns it is fortunately not impossible because object creations can be identified in the source code.

- *Structural* – these patterns deal with the composition of classes or objects. They define class hierarchies and different relations. In these patterns most features are described with the declarations of the operations and attributes, so they are easier to recognize than the creational ones.

- *Behavioral* – these patterns describe how classes interact and distribute responsibility. Hence the behavior is defined in the bodies of the operations, the knowledge of the declarations is insufficient. This makes these patterns the most difficult to recognize.

The recognition of design patterns is a crucial question in reverse engineering, since they represent a high level of abstraction in OO design. As mentioned above, one possible usage might be in measuring the quality of a software system. This can help the project managers to decide whether a code is good enough to be used in the project. Good enough means that the code should be readily understandable, and it should be easy to modify parts of the code without needing to modify the whole code. So if the design patterns are well documented, it should be much easier to understand the source, and to make appropriate modifications on a well-defined part of it.

Another possible usage is in helping documenting a source without proper comments on patterns for gaining advantages described above. Well-commented program code is much easier to maintain than the one without comments or with poor comments. We can find pattern instances and this way help inserting comments where it is necessary. Yet another possible usage is in forward engineering, when the

system designers inspect the source to see if the coders implemented the pattern correctly.

Design patterns are described by listing the intents, motivations, applicability, structure (with UML diagrams [16]), participants, collaborations, consequences, implementation details, sample code, known uses and related patterns. All of these except the sample code are written for humans, they do not prescribe how the pattern will be implemented. Even the sample code is only useful for comparing structures, as function names and implementations may differ. So we must find a way to efficiently describe the patterns and then find a way to compare these pattern descriptions to the code. It is obvious, that the direct comparison to the pure code will not work. The problem is to find an intermediate format in which patterns can be described and to which the code can be relatively easily converted, and to find an algorithm that can efficiently find patterns in the transformed code.

There have only been a few publications on this topic, most of them search only the *structure* of the patterns. We have developed a new method which tries to solve the part of the problem above which can be solved based on the information collected from the source code. Our approach tries to provide as much information as possible from the source. First, we analyze the C++ [11] source code with the Columbus system [5] which builds an internal representation, called the Abstract Semantic Graph (ASG). Next, we load our pattern descriptions which are stored in Design Pattern Markup Language (DPML), a new language based on XML [20] that we designed especially for this purpose. These pattern descriptions are easy to modify to suit the needs of the users. Finally our algorithm binds classes found in the source code to pattern classes that are part of the pattern description and checks whether they are related in a way that is described in the pattern. Here we use composition, aggregation, association and inheritance relationships for classes, and call delegation, object creation and operation redefinition (overriding) for operations. The results of function-body analysis is what gives us more precision compared to others in detecting design pattern occurrences in the source code.

Our system offers methods to the users to define their patterns in a very precise way. This means that one can define patterns in the sense of functionality. This way only the pattern instances that fulfill these fine-grained requirements will be found. This precise definition is unfortunately not applicable for every design pattern. Some patterns are so general, that they can not even be described in this way, like the Facade pattern, which defines a higher-level interface to a set of interfaces.

The paper is organized as follows. In the next section we discuss approaches having similar objectives as ours. In Section 3 we will describe the Columbus reverse engineering tool. Next, in Section 4 we present the design pattern mining algorithm. This section contains also the description of the Design Pattern Markup Language through an example instance of it. In Section 5 we present experimental results of four real-life, publicly available software systems. Finally, in Section 6 we draw some conclusions and outline directions for future work.

2 Related Work

Only few works have been published on the topic of recognizing design patterns from C++ source code, and we have found even fewer papers with concrete results.

Kraemer et al. [13] used the Pat system that was developed by Computec GmbH in cooperation with the University of Karlsruhe, Germany. It works on the output of the Paradigm Plus ooCASE tool [1], which is converted to Prolog facts. It gives the structural analysis of the code based on C++ header files. The relevant extracted information are class names, attribute names, method names and properties, inheritance relations, association and aggregation relations. Some relevant information is not extracted by Paradigm, such as the category of a class (abstract or concrete; all classes are considered concrete), the semantic kind of a method (constructor, destructor, etc.), and delegation of method calls (not visible from header files). They have found true instances of Adapters and Bridges, but have also found false instances of Adapters, Bridges, Composites, Decorators and Proxies. Their precision was between 14-50%.

They examined four real-life projects: Network Management Environment Browser (NME), the Library of Efficient Datatypes and Algorithms (LEDA), the zApp class library and Automatic Call Distribution (ACD). None of these four benchmarks included explicit design information; all data was extracted from C++ header files as described above. The structural analysis and the conversion to Prolog facts took about two hours, while the actual search for the patterns took only a few seconds. All real pattern occurrences were found although the pattern rules could have overlooked some pattern instances because the structural analysis might have mistaken some aggregations (implemented by pointers) for associations. However, it was verified that, in the four benchmarks, none of these cases would have revealed another correct pattern instance. This seems to be a result of good programming style.

The second work [8] describes a method based on a multi-stage reduction strategy using software metrics and structural properties to extract *structural design patterns* from OO design model or source code. Code and design are mapped to an intermediate representation, called Abstract Object Language (AOL). To support the first case (finding patterns in design) an extractor module called CASE2AOL has been implemented for the StP/OMT CASE tool to ob-

tain an AOL specification of the internal object models of the case tool repository. In this case the information extracted is completely trustworthy, in that it really represents design information and no assumptions have to be made about the validity of class relationships. To extract the AOL representation from source code the Code2AOL Extractor module has been developed for the C++ language. Extracting information about class relationships from code is much more difficult than from the design and the result may have some degree of imprecision.

The authors have found true instances of Adapters. They have also found false instances of Adapters, Bridges and Proxies. The testing included three design patterns: Adapter, Bridge and Proxy. Six public domain code systems were analyzed: LEDA, galib, groff, libg++, mec and socket. The first two stages were executed together (metric-based filtering, structural filtering). The third stage (delegation filtering) was executed separately to compare intermediate results. The first stage reduced the input by three to four orders of magnitude. The second stage gave a reduction of one-two orders of magnitude, while the third stage reduced the input two to three times. The precision after the first two steps was about 55%, and an increase of 35% was obtained using the delegation constraint with respect to the use of structural constraints alone. The correctness was 100% because of the conservative approach adopted, that is no correct instances were missed.

In [6], design pattern detection from C++ source code was accomplished with the integration of two existing tools called Columbus [5] and Maisa [17]. The method combines the extraction capabilities of the Columbus reverse engineering tool with the clause-based pattern mining ability of Maisa. First the C++ code is analyzed by Columbus, and a plug-in was written to export the knowledge collected in the abstract semantic graph (ASG) to a form understandable for Maisa. This form is a clause-based design notation in Prolog. Afterwards, it is analyzed by Maisa, and instances are searched that match the previously given design pattern descriptions. No real-world projects were analyzed, but the reference implementations of seven different design patterns were identified.

DP++ [2] detects most of the *structural patterns* primarily based on the detection of structural relationships. It also identifies clusters of functionally related classes, which may not necessarily represent any known pattern, but do represent an abstraction in the program. DP++ successfully identified patterns in several commercial and public-domain object-oriented packages, ranging in size from 30 to 400 classes.

PTIDEJ [10] (Pattern Trace Identification, Detection and Enhancement for Java) was developed to perform the search using a constraint satisfaction problem (CSP). The authors analyzed the Java AWT and net libraries and found occurrences of Composite and Facade design patterns.

Brown [3] developed a method for detecting design patterns in SmallTalk. He encoded the pattern detection methods for Composite, Decorator, Template Method and Chain of Responsibility into his algorithm. He performed testing on four projects in which he found pattern instances.

3 Columbus

Columbus [5] is a reverse engineering framework that has been developed in cooperation between the Research Group on Artificial Intelligence in Szeged, the Software Technology Laboratory of the Nokia Research Center and FrontEndART Ltd [7]. Columbus is able to analyze large C/C++ projects and to extract data according to the Columbus Schema (see Section 3.1).

The main motivation behind developing the Columbus system was to create a tool which implements a general framework for combining a number of reverse engineering tasks, and to provide a common interface for them. Thus Columbus is a framework which supports project handling, data extraction, data representation, data storage and filtering. All these basic tasks of the reverse engineering process are accomplished by using the appropriate modules (*plug-ins*) of the system. Some of these plug-ins are present as basic parts of Columbus, and the system can be extended to include other reverse engineering functionality as well.

3.1 The Columbus Schema

The *Columbus Schema for C++* [4] describes a common *format* which prescribes the form of the information extracted from C++ source code. The schema captures the C++ language at low detail (ASG) and also contains higher-level elements (e.g. semantics of types). The description of the schema is given using standard UML Class Diagrams, which permits its simple implementation and easy physical representation. It is modular, hence it provides additional flexibility for any future extension/modification.

This schema is used as a basis for the Application Programming Interface (API), which allows easy access to facts stored in the schema. This API can be employed in various reverse engineering applications such as front ends and metrics tools. We used it among other things to extract UML class models, build conventional call graphs and, of course, to detect design patterns.

4 Design Pattern Mining

Design pattern mining is a process where the structure of a design pattern is searched in the source code. The structure should include the main properties of the design pattern and it should be flexible enough at the same time to

describe the slightly distorted occurrences as well because in real-world systems patterns are usually modified to serve the solution of the problem. Because of this the structure should be easy to modify and to adapt them to the needs of the user. To meet this commitment, we use an XML–based language to describe design patterns. The language is easy to understand, and the descriptions are easy to modify.

The search for patterns is performed by trying to match source classes to pattern classes. The search is divided into two stages. The first stage is concerned with filtering the candidates for the pattern classes. In the second stage source classes are bound to pattern classes, and the constraints are checked to see if they form a pattern instance. Our method shows which source class, operation and data member plays the role of which pattern class, operation and data member respectively, so it is easy to locate them in the source code.

4.1 Design Pattern Markup Language – DPML

Our algorithm uses an XML–based language for design pattern description. This is the Design Pattern Markup Language – DPML (see the DTD in Figure 1). We will explain the grammar of DPML by an example, the description of the Proxy pattern (see Figures 2 and 3).

The root element is the *<DesignPattern name='...'>* element. Within this element are the classes of the pattern and the type representations. Classes are represented by the *<Class id='id...' name='...' isAbstract='...' isChangeable='...'>* element. The *id* and *name* properties are required, while *isAbstract* and *isChangeable* are optional. The last property tells that the class can have multiple incarnations in a pattern instance. There is no limit for the number of classes but, unfortunately, the algorithm cost grows exponentially with each class. In our example the root element of the pattern description is the DesignPattern element at line 4, which stores the name of the pattern.

The specification of the classes starts with their relations. *<Composition ref='id...' accessibility='...' multiplicity='...'/>* is the element for the composition relationship. The *accessibility* and *multiplicity* properties can be omitted. The other three relations have the same form: *<Aggregation ...>*, *<Association ...>* and *<Base ...>*. When specifying the *Base* (inheritance relation), it makes no sense to specify the multiplicity as it is always 1. In our example, there are class definitions at lines 6, 13 and 27.

The operations of a class can be defined with the *<Operation id='id...' name='...' accessibility='...' storageClass='...' kind='...' isVirtual='...' isPureVirtual='...'>* element. The *id* and *name* properties are required, but the others can be omitted. It can have a *<defines ref='id...'/>* child element to specify which operation it needs to redefine (override). The *ref* property is required. Additional child elements can be *<calls ref='id...'>* and *<creates='id...'>* for describing which operations it calls and which objects it creates, respectively. The *ref* property is required. The operation has a *<hasTypeRep ref='id...'/>* child element which refers to its type representation (see below). The *ref* property is required. The operation can have parameters that are specified by the *<Parameter id='id...' name='...'>* element. Both properties are required. The parameter element also has a *<hasTypeRep ref='id...'/>* child element which refers to the type representation of the parameter.

Class attributes can be defined with the *<Attribute id='id...' name='...' accessibility='...' storageClass='...'>* element. The *id* and *name* properties are required, but the other two can be omitted. A class attribute – like operations – has a *<hasTypeRep ref='id...'/>* child element which refers to its type representation (see below).

In our example the first class (lines 6-11) called *Subject* is abstract, and has an operation *Request* which is pure virtual. Its type (line 9) is represented by TypeRep id50. The second class *Proxy* (lines 13-25) is derived (line 14) from class *Subject*–id10 and aggregates the class *RealSubject*–id30 (line 15). It also has an operation *Request* (lines 16-21) which is virtual, but not pure virtual. It defines/overrides the inherited operation *Request*–id11 (line 18) and calls the operation *Request*–id31 from class *RealSubject* (line 19). Its type is represented by TypeRep id50 (line 20). This class has an attribute called *realSubject*as well (lines 22-24) whose type is represented by TypeRep id52 (line 23). The third class *RealSubject* (lines 27-34) is derived (line 28) also from class *Subject*–id10. It also has an operation *Request* (lines 29-33) which is virtual but not pure virtual. It also defines/overrides the inherited operation *Request*–id11 (line 31). Its type is represented by TypeRep id50 (line 32).

The *<TypeRep id='id...'>* element represents a type. It can have different child elements which modify the referred type, like *<TypeFormerArr/>* for arrays, *<TypeFormerPtr/>* for pointers/references and *<TypeFormerFunc>* for functions. The reference to the type is stored in a *<TypeFormerType* ref='id...'/> element. The *<TypeFormerFunc>* element has a *<hasReturnTypeRep ref='id...'/>* child element for referring to the return type and can have several *<hasParameterTypeRep ref='id...'/>* child elements for referring to the types of the parameters.

In the example the first type representation (lines 36-40) shows a function type (line 37) and its return type is described in TypeRep id51 (line 38). The second type representation (line 42) is empty, which means that it can represent any type. The third type representation (lines 44-47) indicates a pointer (line 45) to an object of type id30–*RealSubject* (line 46).

4.2 The Pattern Miner Algorithm

First, the Columbus framework analyzes the C++ source code and builds an abstract semantic graph (ASG) from it. Second, the appropriate DPML pattern description file is loaded into a standard XML DOM [19] tree. The pattern miner algorithm (see Figure 4) then matches the DOM tree to the ASG. This process is similar to graph matching where the vertices are classes and the edges are relations between the classes. Moreover, Columbus calculates the class diagram and call graph from the ASG.

Since class diagrams are reconstructed from source code differently by different reverse engineering tools, we will explain here what we mean under the concept of inheritance, composition, aggregation and association. *Inheritance* means that a class derives from another class, the base class (for design pattern detection, inheritances are stored transitively as well). *Composition* means that a class directly contains another class instance by a data member. *Aggregation* means that a class indirectly contains another class by having a pointer or a reference to it. *Association* means that a class uses another class by getting it as an operation parameter or by returning it as a return type. The class diagram and call graph are used later in the algorithm.

The algorithm starts by collecting *source class* (a class found in the source code) candidates for each *pattern class* (a class found in the pattern description). This is accomplished by searching classes with an appropriate number of attributes and operations with desired properties[1]. Only the properties given in the pattern description are checked. The number of different relations[2] of the class is checked as well. If a source class has all the required attributes, operations and relations it is then stored as a candidate for the appropriate pattern class. One source class can become a candidate for multiple pattern classes.

In the second stage the candidates are filtered. This is done by checking the connections with other pattern class candidates: if two pattern classes are related in some way, then all candidate classes of the first pattern class have to be connected in the same way to at least one candidate class of the second pattern class. If a class does not have the necessary relations, it is removed from the candidates list. This step is repeated iteratively until no further classes are removed.

Next, all combinations of the candidate classes have to be tested to find design pattern instances. The algorithm searches for these combinations recursively to check every possibility (procedure *bindSourceClassToPatternClass*). If a combination of classes has all the required connections, the attributes and operations of the source classes have to be matched to the attributes and operations of the pattern classes.

These attributes and operations are also matched recursively to try out every combination (function *bindAttributesAndOperations*). A source class attribute is bound to a pattern class attribute, if it has all the required properties, checking this time its type as well. For operations, beside the basic properties, the return types and parameters have to be checked too.

After a combination was found, a further check has to be performed to see if the bound functions contain the needed call delegations, object creations and whether they redefine/override the appropriate operations (function *checkCallsCreatesDefines*). This is performed sequentially, for each class and within them for each function.

If this last check was successful, then the current bound source classes form a design pattern instance. These classes, their path and line information and the role they play in the design pattern instance are then displayed (together with the bound operations and attributes).

4.3 Optimizations

The algorithm outlined above essentially describes how our algorithm works. Obviously, the basic algorithm is not optimal, so we will mention two ideas which increase its performance.

The first optimization (see Figure 5) further reduces the candidate lists using the information provided by the class diagram built at the beginning of the algorithm. The problem is the following: let us take a big system with about 10000 classes. An average pattern consists of 3 classes. Let us presume that for each class there are about 1000 candidates after the filtering. Then about $1000*1000*1000 = 10^9$ combinations have to be checked, which is too expensive.

This number can be dramatically reduced by using the following technique. First, a graph is built where the vertices are the pattern classes and the edges are the relations among them. Second, a topological ordering is performed on it. In this way an ordering of pattern classes is carried out where the latter ones depend on a previous one. The procedure *makeSmallCandidateSets* creates new smaller candidate class subsets for a given pattern class from the original candidates set. These will take the place of the original (large) candidate sets in the *bindSourceClassToPatternClass* procedure. The union of these small sets forms the original set.

The small sets are created in the following way. The first relation is taken between two pattern classes where the second class depends on the first one (the first pattern class occurs before the second one in the topological order). For each candidate class of the first pattern class the small set

[1] Only simple properties like virtuality and accessibility are checked, no type-checking is done at this stage.

[2] Relations can be: inheritance, composition, aggregation and association.

consists of those candidate classes of the second pattern class which are in the same relation with the first candidate class as the second pattern class is with the first one. Each pattern class can play the role of the second pattern class in the previous discussion only once. This way a small set is unambiguously defined by two pattern classes and a source class that plays the role of the first pattern class.

These small sets are used in procedure *bindSourceClassToPatternClass* in the following way. If during class binding previously created small sets exist which are defined by the actual pattern class, the source class bound to it and the dependent pattern classes, then the candidate sets of the dependent ones are replaced by the small ones. If the small sets are 100 times smaller, say, than the original ones then the number of combinations will be 1000*10*10 = 10^5, which is an improvement of four orders of magnitude (this is of course strongly depends on the system being analyzed).

The second optimization is a cut in the operation and attribute matching. If a pattern element is considered which says that an operation must create an object, then it can be checked whether the operation being matched is indeed creating an object of the given type. If it does not, this combination of operations and attributes is not adequate and the matching will continue with another source operation. In the case of call delegation and operation redefinition this checking is possible only if the called or redefined operation has already a bound source class element.

To be concrete (see Figure 6), this means that the *bindAttributesAndOperations* function is modified so that it calls a new *checkOperationConstraints* function, which checks the above described conditions.

5 Experiments

In this section we demonstrate the design pattern detection capabilities of our system. The experiments were performed on four real-life, publicly available C++ projects listed below:

- *Jikes* [12]. Open-source Java compiler system from IBM.
- *Leda* [14, 15]. Library of efficient data types and algorithms.
- *StarOffice Calc* [18]. The spreadsheet application of StarOffice, a large C++ project that consist of 6,307 source files (more than 1.2 million non-preprocessed non-empty lines of code).
- *StarOffice Writer* [18]. The word processing application of StarOffice, a large C++ project that consist of 6,794 source files (more than 1.5 million non-preprocessed non-empty lines of code).

The following table summarizes the size information of the projects.

Size info	Jikes	Leda	Calc	Writer
No. of files	77	488	6 307	6 794
Size	2.7MB	2.7MB	44.5MB	52.2MB
LOC[3]	75 485	85 606	1 270 303	1 512 972
No. of classes	329	1 617	5 051	6 729

The next table shows the number of different design pattern instances found in the test projects. For most design patterns we have also prepared their "soft" versions in which we slightly relaxed the original specifications in [9]. The patterns are grouped as in [9]: creational, structural and behavioral patterns.

Statistics	Jikes	Leda	Calc	Writer
Abstract Factory	-	-	-	-
Builder	-	-	2	7
Builder soft	-	-	17	9
Factory Method	-	-	-	-
Factory Method soft	-	-	1	9
Prototype	1	-	-	1
Prototype soft	1	-	-	1
Singleton	-	-	-	-
Adapter Class	-	-	-	16
Adapter Class soft	-	-	13	16
Adapter Object	54	-	27	62
Adapter Object soft	62	-	153	135
Bridge	-	-	-	-
Bridge soft	-	-	73	80
Decorator	-	-	-	-
Decorator soft	-	-	-	-
Proxy	36	-	-	4
Proxy soft	44	-	-	5
Chain of Responsibility	-	-	-	-
Iterator	-	-	-	-
Iterator soft	-	-	1	-
Strategy	4	1	10	5
Strategy soft	12	2	20	32
Template Method	5	-	94	101
Visitor	-	-	-	-
Visitor soft	-	-	-	5
Sum total	235	6	442	525

The next table shows the percentage of true pattern instances compared to the found ones. All found instances (except the instances of *Adapter Object* and its soft version) were checked manually in the source code (those design patterns are not shown for which no instances were found).

[3]Lines of non-empty non-preprocessed code.

Statistics	Jikes	Leda	Calc	Writer
Builder	-	-	0%	14%
Builder soft	-	-	0%	11%
Factory Method soft	-	-	100%	100%
Prototype	100%	-	-	100%
Prototype soft	100%	-	-	100%
Adapter Class	-	-	-	0%
Adapter Class soft	-	-	0%	0%
Bridge soft	-	-	100%	100%
Proxy	0%	-	-	50%
Proxy soft	18%	-	-	60%
Iterator soft	-	-	0%	-
Strategy	100%	100%	100%	100%
Strategy soft	100%	100%	100%	100%
Template Method	100%	-	100%	100%
Visitor soft	-	-	-	0%

The next table shows the pattern mining time for different patterns (all tests were performed on an Intel P4-2000 machine with 512MB RAM running Windows 2000). It does not include the time needed for source code analysis.

Time[4]	Jikes	Leda	Calc	Writer
Abstract Factory	0:00:00	0:00:00	0:00:02	0:00:10
Builder	0:00:01	0:00:00	0:00:11	0:00:29
Builder soft	0:00:02	0:00:02	0:32:41	0:47:05
Factory Method	0:00:00	0:00:00	0:00:02	0:00:12
Factory Method sf	0:00:00	0:00:00	0:02:00	0:04:06
Prototype	0:00:43	0:00:00	0:00:22	0:00:47
Prototype soft	0:00:44	0:00:00	0:14:57	0:30:12
Singleton	0:00:00	0:00:00	0:00:00	0:00:00
Adapter Class	0:00:00	0:00:00	0:00:01	0:00:01
Adapter Class soft	0:00:00	0:00:00	0:00:01	0:00:16
Adapter Object	0:00:01	0:00:00	0:01:47	0:02:12
Adapter Object sft	0:00:04	0:00:00	0:20:31	0:30:22
Bridge	0:00:00	0:00:00	0:00:00	0:00:23
Bridge soft	0:00:00	0:00:01	0:49:19	1:53:08
Decorator	0:00:00	0:00:00	0:00:00	0:00:01
Decorator soft	0:00:00	0:00:00	0:05:05	0:11:56
Proxy	0:00:00	0:00:00	0:00:00	0:00:00
Proxy soft	0:00:00	0:00:00	0:00:01	0:00:02
Chain of Resp.	0:00:00	0:00:00	0:00:00	0:00:01
Iterator	0:00:00	0:00:00	0:00:00	0:00:00
Iterator soft	0:00:00	0:00:00	0:00:00	0:00:00
Strategy	0:00:33	0:00:00	0:00:42	0:01:34
Strategy soft	0:00:38	0:00:30	1:27:01	2:27:56
Template Method	0:00:10	0:00:00	0:00:11	0:00:51
Visitor	0:00:00	0:00:00	0:00:00	0:00:01
Visitor soft	0:00:00	0:00:00	0:00:01	0:00:04
Sum total	0:04:08	0:01:04	5:02:47	9:00:53

We will discuss the obtained results in the next section.

[4]Time format: h:mm:ss.

6 Conclusions and Future Work

In this paper we presented a method for discovering design pattern instances in C++ source code. We tried out a new approach to the problem of pattern detection which includes the detection of call delegations, object creations and operation redefinitions. These are the elements that identify pattern occurrences more precisely. The pattern descriptions are stored in an XML–based format, the Design Pattern Markup Language (DPML). This gives the user the freedom to modify the patterns, adapt them to his or her own needs, or create new pattern descriptions. We tested our method on four public-domain projects. Except for LEDA, the other three are newer projects, and it was noticeable that there were more patterns in those projects than in LEDA. Leda and Jikes are medium-sized projects where the search took only a few minutes, which is acceptable. StarWriter and StarCalc are huge projects, where the search time took several hours. This is reasonable if we consider that they contain more than a million lines of code, but the time also indicates that we should make even more effort on optimizing our algorithm.

The main advantage that this work offers is that it provides design pattern instance detection using a user-friendly language for design pattern description. Thus, it will be straightforward to identify pattern instances with this tool, and it can be helpful in code comprehension, code documentation, correct pattern implementation testing, and in other fields which require pattern mining.

The testing we performed revealed several things. Firstly, the larger a project is, more likely it will contain design patterns. Since the patterns are implemented to solve a specific problem in a specific environment, they rarely follow the strict descriptions in [9]. The implementations usually violate some rules, so the pattern description must be simplified to be able to recognize them. But the more we simplify the pattern description, the more false positives we will get. Sometimes it is almost impossible to determine whether a pattern instance we have found is a real or false instance. The structure is proper, but the pattern instance does not do what we expect. This means that we can find source classes that are connected in the right way, but we cannot check whether they fulfill the intent of the original pattern. This can be checked only manually. Except for the instances of the *Adapter Object* pattern (because of their high number) we have manually checked the found instances. The results are collected in the third table in the previous section.

The names of the operations and classes usually suggest the role they play, like operation *Clone* in a *Prototype* pattern instance found in *Jikes*, or *CreateNew* in a *Prototype* found in *StarWriter*. Again, some names indicated that it was a false positive we were looking at, but we could not eliminate them. These false positives also have the required

structure, but they have a different intent from the design pattern they are similar to. In the cases of trivial patterns like the *Template Method*, there is no chance of finding false positives. On the other hand, all the instances of *Adapter Class* are false positives, because we could not describe the delegation of the important part of the job, so we got results where only marginal parts of the jobs were delegated to the *Adaptee* class. In the case of the *Proxy* pattern where we found true and false instances also, we accepted only the instances where the call delegation did the main part of the job.

The search time is long in cases when many operations and attributes in the source classes need to be bound to pattern operations and attributes. This part of the algorithm should be optimized.

We have encountered an interesting problem during the pattern formalization process. The structure of the *State* and *Strategy* patterns are so similar that we could not distinguish them in the pattern description. The differences are in motivation and intent, which we could not formalize. So the tables in the previous section do not contain the results for the *State* pattern.

For the future, we need to take into account standard template containers. Developers rarely use simple arrays, but normally use standard containers instead, and we need to find out what kind of data is stored in them. This would allow us to find many more relations, and to find patterns that use lists like Flyweight, Mediator or Observer. We need to write pattern descriptions for simplified pattern instances which do not use every class from the original pattern description. For example, if there is only one *Implementor* in the *Bridge* pattern, there is no need for an abstract parent class.

As soon as the extractor for Java is available in Columbus, we will develop the corresponding version of our algorithm for this language as well. This requires a modified algorithm, because the actual algorithm checks pointers, say, that are not available in Java, but it does not check interfaces which are widely used in Java. Most design pattern descriptions need not be modified, because they are intended to be language independent.

We plan to build a benchmark on which this and other systems could be tested. This way different design pattern miner systems could be compared and the deficiencies of the algorithms could be easily discovered.

References

[1] AllFusion Component Modeler (formerly Paradigm Plus) Homepage. `http://www3.ca.com/Solutions/Product.asp?ID=1003`.

[2] J. Bansiya. DP++ is a tool for C++ programs. In *Dr. Dobb's Journal*, June 1998.

[3] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. In *Master's thesis*. Department of Computer Engineering, North Carolina State University, 1996.

[4] R. Ferenc and Á. Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, Mar. 2002.

[5] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[6] R. Ferenc, J. Gustafsson, L. Müller, and J. Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. In *Acta Cybernetica journal vol. 15*, pages 669–682. University of Szeged, 2002.

[7] Homepage of FrontEndART Software Ltd. `http://www.frontendart.com`.

[8] R. F. G. Antoniol and L. Cristoforetti. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *Proceedings of the Fifth International Symposium on Software Metrics (METRICS98)*, pages 23–34, Nov. 1998.

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

[10] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design patterns identification. In *Proceedings of IJCAI Workshop on Modelling and Solving Problems with Constraints*, pages 57–64, Aug. 2001.

[11] International Standards Organization. *Programming languages – C++*, ISO/IEC 14882:1998(E) edition, 1998.

[12] IBM Jikes Project. `http://oss.software.ibm.com/developerworks/opensource/jikes`.

[13] C. Kraemer and L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'96)*, pages 208–215, Nov. 1996.

[14] The LEDA Homepage. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[15] K. Mehlhorn and S. Naeher. Leda: A platform for combinatorial and geometric computing. In *Cambridge University Press*, 1997.

[16] Object Management Group Inc. *OMG Unified Modeling Language Specification*, version 1.3 edition, 1999.

[17] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332, 2000.

[18] The StarOffice Homepage. `http://www.sun.com/software/star/staroffice`.

[19] World Wide Web Consortium (W3C). *Document Object Model (DOM)*, 2000.

[20] World Wide Web Consortium (W3C). *Extensible Markup Language (XML)*, version 1.0 edition, 2000.

```
<!ENTITY % Boolean "(true|false)">
<!ENTITY % AccessibilityKind "(private|protected|public
                              |notPrivate|notProtected|notPublic)">
<!ENTITY % StorageClassKind "(static)">
<!ENTITY % FunctionKind "(normal|constructor|destructor)">


<!ELEMENT DesignPattern (Class*, TypeRep*)>
<!ATTLIST DesignPattern name CDATA #REQUIRED>
<!ELEMENT Class ((Base|Composition|Aggregation|Association)*,
                 (Operation|Attribute)*)>
<!ATTLIST Class id ID #REQUIRED
                name CDATA #REQUIRED
                isAbstract %Boolean; #IMPLIED
                isChangeable %Boolean; #IMPLIED>
<!ELEMENT Operation (defines?, (calls|creates)*, hasTypeRep, Parameter*)>
<!ATTLIST Operation id ID #REQUIRED
                    name CDATA #REQUIRED
                    accessibility %AccessibilityKind; #IMPLIED
                    storageClass %StorageClassKind; #IMPLIED
                    kind %FunctionKind; #IMPLIED
                    isVirtual %Boolean; #IMPLIED
                    isPureVirtual %Boolean; #IMPLIED>
<!ELEMENT Attribute (hasTypeRep)>
<!ATTLIST Attribute id ID #REQUIRED
                    name CDATA #REQUIRED
                    accessibility %AccessibilityKind; #IMPLIED
                    storageClass %StorageClassKind; #IMPLIED>
<!ELEMENT Parameter (hasTypeRep)>
<!ATTLIST Parameter id ID #REQUIRED
                    name CDATA #REQUIRED>
<!ELEMENT Base EMPTY>
<!ATTLIST Base ref IDREF #REQUIRED
               accessibility %AccessibilityKind; #IMPLIED
               multiplicity CDATA #IMPLIED>
<!ELEMENT Composition EMPTY>
<!ATTLIST Composition ref IDREF #REQUIRED
                      accessibility %AccessibilityKind; #IMPLIED
                      multiplicity CDATA #IMPLIED>
<!ELEMENT Aggregation EMPTY>
<!ATTLIST Aggregation ref IDREF #REQUIRED
                      accessibility %AccessibilityKind; #IMPLIED
                      multiplicity CDATA #IMPLIED>
<!ELEMENT Association EMPTY>
<!ATTLIST Association ref IDREF #REQUIRED
                      accessibility %AccessibilityKind; #IMPLIED
                      multiplicity CDATA #IMPLIED>
<!ELEMENT defines EMPTY>
<!ATTLIST defines ref IDREF #REQUIRED>
<!ELEMENT calls EMPTY>
<!ATTLIST calls ref IDREF #REQUIRED>
<!ELEMENT creates EMPTY>
<!ATTLIST creates ref IDREF #REQUIRED>


<!ELEMENT TypeRep (TypeFormerType|TypeFormerArr
                   |TypeFormerPtr|TypeFormerFunc)*>
<!ATTLIST TypeRep id ID #REQUIRED>
<!ELEMENT TypeFormerType EMPTY>
<!ATTLIST TypeFormerType ref IDREF #REQUIRED>
<!ELEMENT TypeFormerArr EMPTY>
<!ELEMENT TypeFormerPtr EMPTY>
<!ELEMENT TypeFormerFunc (hasReturnTypeRep, hasParameterTypeRep*)>
<!ELEMENT hasTypeRep EMPTY>
<!ATTLIST hasTypeRep ref IDREF #REQUIRED>
<!ELEMENT hasReturnTypeRep EMPTY>
<!ATTLIST hasReturnTypeRep ref IDREF #REQUIRED>
<!ELEMENT hasParameterTypeRep EMPTY>
<!ATTLIST hasParameterTypeRep ref IDREF #REQUIRED>
```

Figure 1. Design Pattern Markup Language

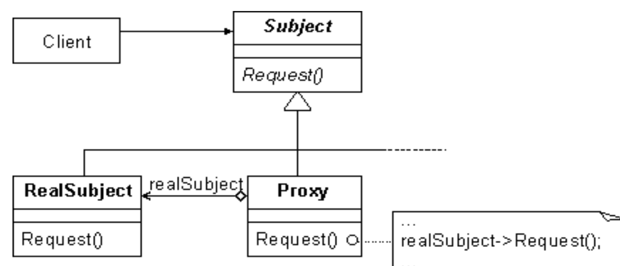

Figure 2. The Proxy design pattern

```
01<?xml version='1.0'?>
02<!DOCTYPE DesignPattern SYSTEM 'dpml-1.6.dtd'>
03
04<DesignPattern name='Proxy'>
05
06  <Class id='id10' name='Subject' isAbstract='true'>
07    <Operation id='id11' name='Request' kind='normal'
08               isVirtual='true' isPureVirtual='true'>
09      <hasTypeRep ref='id50'/>
10    </Operation>
11  </Class>
12
13  <Class id='id20' name='Proxy'>
14    <Base ref='id10'/>
15    <Aggregation ref='id30'/>
16    <Operation id='id21' name='Request' kind='normal'
17               isVirtual='true' isPureVirtual='false'>
18      <defines ref='id11'/>
19      <calls ref='id31'/>
20      <hasTypeRep ref='id50'/>
21    </Operation>
22    <Attribute id='id22' name='realSubject'>
23      <hasTypeRep ref='id52'/>
24    </Attribute>
25  </Class>
26
27  <Class id='id30' name='RealSubject'>
28    <Base ref='id10'/>
29    <Operation id='id31' name='Request' kind='normal'
30               isVirtual='true' isPureVirtual='false'>
31      <defines ref='id11'/>
32      <hasTypeRep ref='id50'/>
33    </Operation>
34  </Class>
35
36  <TypeRep id='id50'>
37    <TypeFormerFunc>
38      <hasReturnTypeRep ref='id51'/>
39    </TypeFormerFunc>
40  </TypeRep>
41
42  <TypeRep id='id51'/>
43
44  <TypeRep id='id52'>
45    <TypeFormerPtr/>
46    <TypeFormerType ref='id30'/>
47  </TypeRep>
48
49</DesignPattern>
```

Figure 3. The Proxy pattern in DPML

```
function checkCallsCreatesOverloadsDefines() : bool
  foreach pattern class P1 ∈ PC
    foreach operation O1 ∈ O(P1)
      foreach relation R between operations O1 and O2
        if R ∉ RO(actBoundElement(O1),actBoundElement(O2)) then
          return false
      foreach relation R between operation O1 and pattern class P2
        if R ∉ ROC(actBoundElement(O1),actBoundClass(P2)) then
          return false
  return true
endfunc

function bindAttributesAndOperations(PC index i, PE index j) : bool
  if j ≤ PC[i].size then
    foreach element E ∈ actBoundClass(PC[i])
      if E is not yet bound then
        if P(PE(PC[i],j)) ⊆ P(E) and Type(PE(PC[i],j)=Type(E) then
          actBoundElement(PE(PC[i],j)) := E
          if bindAttributesAndOperations(i,j+1) then return true
          clear binding of PE(PC[i],j)
  else
    if i ≤ PC.length then
      if bindAttributesAndOperations(i+1,1) then return true
    else
      if checkCallsCreatesDefines() then return true
  return false
endfunc

procedure bindSourceClassToPatternClass(PC index i)
  if i ≤ PC.length then
    foreach C ∈ Candidates(PC[i])
      if C is not yet bound then actBoundClass(PC[i]) := C
      bindSourceClassToPatternClass(i+1)
  else
    foreach pattern class P1 ∈ PC
      foreach relation R between pattern classes P1 and P2
        if rel(R,actBoundClass(P1),actBoundClass(P2)) then
          if bindAttributesAndOperations(1,1) then
            the bound source classes form a design pattern instance
    clear all bindings for operations and attributes
endproc

program PatternMiner
  analyze the subject system and build the ASG
  calculate the class diagram
  calculate the call graph
  load pattern classes from DPML to PC
  foreach pattern class P ∈ PC
    foreach source class S
      if P(P) ⊆ P(S) and A(P) ⊆ A(S) and O(P) ⊆ O(S) and R(P) ⊆ R(S) then
        Candidates(P) += S
  repeat
    foreach pattern class P1 ∈ PC
      foreach relation R between pattern classes P1 and P2
        foreach candidate class C ∈ Candidates(P1)
          if not relSet(R,C,Candidates(P2)) then
            Candidates(P) -= C
  until no further classes are removed
  bindSourceClassToPatternClass(1)
endprog
```

PC – *list of pattern classes*
A(C) – *set of attributes of class C*
O(C) – *set of operations of class C*
R(C) – *set of class-class relations of class C (inheritance, composition, aggregation, association)*
RO(F1,F2) – *set of operation-operation relations between operations F1 and F2 (call, redefine)*
ROC(F,C) – *set of operation-class relations between operation F and class C (creates)*
Candidates(C) – *set of candidate source classes for pattern class C*
rel(R,C1,C2) – *checks if a relation R exists between classes C1 and C2*
relSet(R,C,S) – *checks if a relation R exists between class C and at least one of the classes from set S*
PE(P,i) – *i. element (operation or attribute) of the P pattern class*
actBoundClass(P) – *source class bound to the P pattern class*
actBoundElement(E) – *source class element bound to pattern class element E*
P(X) – *set of properties of X (X can be a class or class element)*
Type(E) – *type of class element E*

Figure 4. The Pattern Miner Algorithm

```
procedure makeSmallCandidateSets()
  perform topological ordering of the pattern classes and store them ordered to TO
  foreach pattern class P1 ∈ TO
    Visited(P1) := true
    foreach relation R between pattern classes P1 and P2
      if not Visited(P2) then
        Visited(P2) := true
        foreach candidate class C1 ∈ Candidates(P1)
          foreach candidate class C2 ∈ Candidates(P2)
            if rel(R,C1,C2) then
              Candidates(P1,C1,P2) += C2
endproc

procedure bindSourceClassToPatternClass(PC index i)
  if i ≤ PC.length then
    foreach C ∈ Candidates(PC[i])
      if C is not yet bound then actBoundClass(PC[i]) := C
→     foreach PC[j]: j>i and Candidates(PC[i],C,PC[j])≠∅
→       Candidates(PC[j]) := Candidates(PC[i],C,PC[j])
      bindSourceClassToPatternClass(i+1)
  else
    ...
endproc

program PatternMiner
  ...
  until no further classes are removed
→makeSmallCandidateSets()
→PC:=TO
  bindSourceClassToPatternClass(1)
endprog
```

TO – *topologically sorted list of pattern classes*
Candidates(P1,C,P2) – *set of candidate source classes for pattern class P2 if C plays the role of P1*

Figure 5. Candidates lists reduction

```
function checkOperationConstraints(pattern element O1) : bool
  if O1 is not an operation then return true
  foreach relation R between operations O1 and O2
    if actBoundElement(O2)=∅ then
      return true
    if R ∉ RO(actBoundElement(O1),actBoundElement(O2)) then
      return false
  foreach relation R between operation O1 and pattern class P2
    if R ∉ ROC(actBoundElement(O1),actBoundClass(P2)) then
      return false
  return true
endfunc

function bindAttributesAndOperations(PC index i, PE index j) : bool
  if j ≤ PC[i].length then
    foreach element E ∈ actBoundClass(PC[i])
      if E is not yet bound then
        if P(PE(PC[i],j)) ⊆ P(E) and Type(PE(PC[i],j)=Type(E) then
          actBoundElement(PE(PC[i],j)) := E
→         if checkOperationConstraints(PE(PC[i],j)) then
            if bindAttributesAndOperations(i,j+1) then return true
          clear binding of PE(PC[i],j)
  else
    ...
endfunc
```

Figure 6. Cut in the operation and attribute matching