Using Metrics to Identify Design Patterns in Object-Oriented Software

G. Antoniol, R. Fiutem and L. Cristoforetti
ITC-IRST
Istituto per la Ricerca Scientifica e Tecnologica
Povo (Trento), Italy I-38050
e-mail: antoniol|fiutem|cristofo@itc.it

ABSTRACT

Object-Oriented design patterns are an emergent technology: they are reusable micro-architectures, high level building blocks.

This paper presents a conservative approach, based on a multi-stage reduction strategy using OO software metrics and structural properties to extract structural design patterns from OO design or code. Code and design are mapped into an intermediate representation, called Abstract Object Language, to maintain independence from the programming language and the adopted CASE tools.

To assess the effectiveness of the pattern recovery process a portable environment written in Java, remotely accessible by means of any WEB browser, has been developed. Based on this environment, experimental results obtained on public domain and industrial software are discussed in the paper.

1 INTRODUCTION

Object-Oriented (OO) design patterns are an emergent technology: they represent well-known solutions to common design problems in a given context. The most famous OO design patterns collection is contained in the book of Gamma et Oth. [4]: 23 design patterns were collected and documented by the authors who also presented pattern implementation in Smalltalk and C++.

This paper presents a conservative approach, based on a multi-stage reduction strategy using software metrics and structural properties to extract structural design patterns from OO design or code. Code and design are mapped into an intermediate representation, called Abstract Object Language (AOL) [8], to maintain independence from the programming language and the adopted CASE tools. Such representation contains information about classes, their methods and attributes as well as relations among classes. While for forward engineering the benefit of using design patterns is clear, from program understanding and reverse engineering perspective the discovery of patterns in a software artifact (e.g., design or code) represents a step in the program understanding process. In other words, it helps in understanding a piece of design or code: a pattern provides knowledge about the role of each class within the pattern, the reason of certain relationship among pattern constituents and/or the remaining parts of a system.

Moreover, a system which has been designed using well-known, documented and accepted design patterns is also likely to exhibit good properties such as modularity, separation of concerns and maintainability, thus design patterns can also give some indications to managers about the quality of the overall system.

The presence of patterns in a design should be reflected also in the corresponding code: being able to extract pattern information from both design and code is fundamental in identifying traceability links between different documents, explaining the rationale of the chosen solution in a given system and thus simplifying the activity of building its conceptual model.

Being design patterns a relatively young field, few works in program understanding and reverse engineering have addressed design pattern recovery. Kramer and Prechelt [6] have proposed an approach and developed a system, called Pat, that localize instances of structural design patterns by means of structural information. It exploits the reverse engineering capability of a CASE tool repository to extract design information and uses Prolog facts to represent it and rules to express patterns. A Prolog query searches the fact database for all pattern instances.

Shull, Melo and Basili [10] have developed an inductive method to help discovering custom, domain-specific design patterns in existing OO software systems. The method however is performed manually, although it could be greatly assisted by tools.

Different approaches, exploiting software metrics, were used in previous works to automatically detect design concepts and function clones [5, 7] in large software systems.

In the approach we present, a design pattern is represented as a tuple of classes and relations among classes. When examining potential pattern instances, to avoid combinatorial explosion in checking all possible class

combinations, OO software metrics are used to determine pattern constituents candidate sets. Software metrics, as well as structural properties, are extracted from an Abstract Syntax Tree (AST) produced by parsing the AOL representation. Pattern structure is then exploited to further reduce the search space. In fact, a pattern can be conceived as a graph in which nodes are classes and edges correspond to relations. Once a pattern element is chosen, remaining classes are related to it by the number of in between relations. Thus, the problem of further reducing the search space is mapped into a shortest path problem where the pattern imposes the existence of certain number of edges between candidate classes. Shortest path filtering effectively further reduces the number of candidates tuples determined by the first application of software metrics. On these reduced sets, exact structural design pattern constraints are applied.

Since necessary conditions are applied in pattern recovery, our approach is conservative, i.e. if a pattern is in the code it is surely reported in the results. Being conservative, sometimes extra patterns are reported: to further reduce false alarms, design pattern constraints in term of method delegation have been exploited. Method delegation means that a class implements one operation by simply calling an operation of another class to which it is associated, thus delegating the responsibility to it.

The proposed approach has been evaluated on C++ public domain and industrial systems and results are presented in the paper. A system implementing the described approach has been prototyped. The entire recognition process, is driven by a WEB interface. The recognizer can be accessed by remote users through a WWW address. The user, by means of a WWW browser, connects to a WWW server where the entire pattern recovery process is performed. He/she then chooses the system to be analyzed, the pattern/s to be recognized, and starts the recognizer.

The following sections present our pattern extraction process: next section introduces pattern concepts in the framework of the structural patterns subset proposed by [4]. Section 3 describes the pattern identification process while Section 4 presents the experimental results. Finally Section 5 draws conclusions and outline ideas for future work.

2 DESIGN PATTERNS

Design patterns are well-known and frequently reused micro-architectures: they provide proven solutions to design recurring problems within certain contexts. A pattern description encompasses its static structure, in terms of classes and objects participating to the pattern and their relationships, but also the behavioral pattern dynamics, in terms of participants exchanged

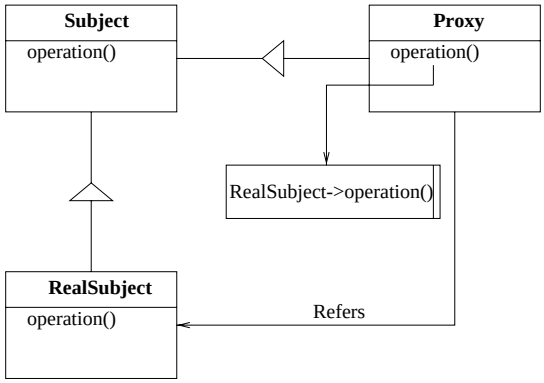

Figure 1: Proxy design pattern.

messages. The description of the solution tries to capture the essential insight which the pattern embodies, so that others may learn from it, and make use of it in similar situations: patterns help create a shared language for communicating insight and experience about these problems and their solutions.

According to the tassonomy proposed by Gamma design patterns can be classified into **creational**, **structural** and **behavioral**.

Creational patterns concern object creation, structural patterns capture classes or object composition and behavioral patterns deal with the way in which classes and objects distribute responsibility and interact.

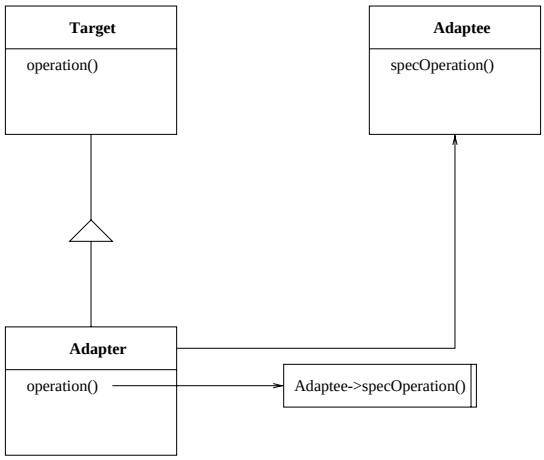

Figure 2: Adapter design pattern.

From a program understanding and reverse engineering perspective, the complexity of extracting information from a design or source code is not the same for the different pattern categories. While for structural design patterns most information is explicit in their syntactic representation, for the other two pattern families

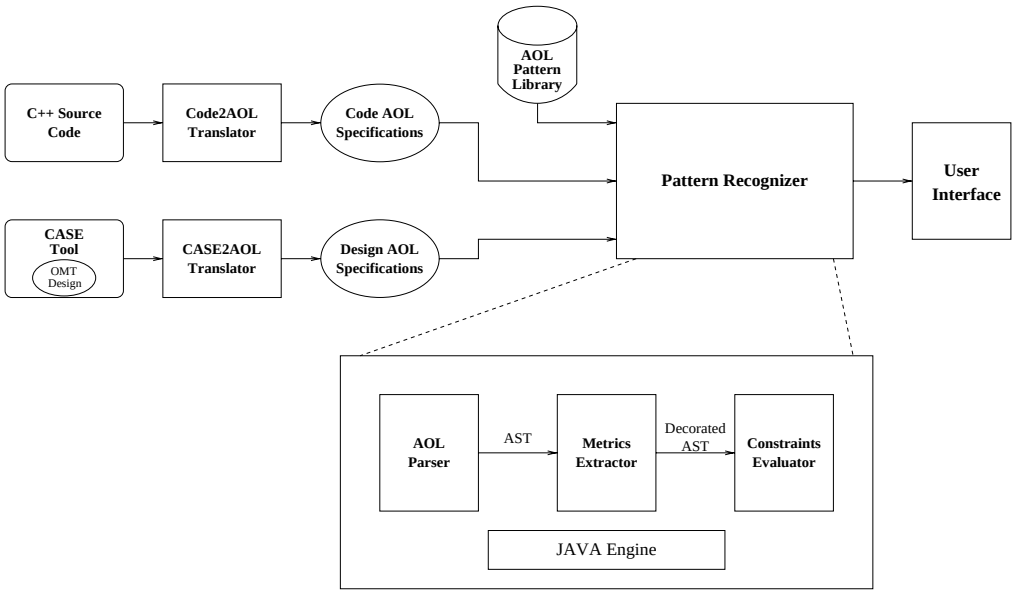

Figure 3: Pattern Extraction Process.

behavioral information must be recovered, which might involve the analysis of messages exchanged and the code of class methods. This also implies that recovery of such kind of patterns is not always feasible at design stage, where in most cases the only information available is the bare static information obtainable from class diagrams. Even the class diagram information could be poor: it is not unusual, in the authors' experience, that neither data member type nor method signature is specified. Moreover, almost always no information is available on method body neither as pseudocode nor as the set of messages sent to other objects.

For these reasons, we focused our recovery effort on five of the seven structural design patterns proposed in [4]: the *adapter*, the *proxy*, the *composite*, the *bridge* and the *decorator*. Being focused on structural design patterns does not mean that non structural design pattern like *iterator* or *observer* cannot be represented or searched at all: it simply means that the extraction process does not exploit pattern behavioral constraints, thus a larger number of false positives can occur with respect to structural patterns.

In Figure 1 and 2 the OMT [9] class diagram of the *proxy* and the *adapter* patterns are shown as an example. The *proxy* design pattern packages a common technique used in networks: a local object needs to communicate with a remote process but we want to hide the details about the remote process location or the communication protocol. The *proxy* object allows to access remote services with the same interface of local processes. In fact, when an *Operation* is required to the *Proxy* object, it delegates the implementation of the required operation to the *RealSubject* object. Being both *Proxy* and *RealSubject* subclasses of *Subject*, this guarantees that they export the same interface for *Operation*. To be able to call *RealSubject* methods, *Proxy* needs an association to it.

On the contrary, in the *adapter* pattern, the *Adapter* object adapts the interface exported by the *Adaptee* object, so that the *Adaptee* services can be called with the different calling conventions of the *Target* object. This implies that a subclass of *Target*, the *Adapter* is created, which actually implements the abstract operation *Operation* exported by *Target*, by delegating the task to the *SpecificOperation* of *Adaptee*.

3 RECOVERY PROCESS

The whole design pattern recovery process is represented in Fig. 3. Both design and source code can be the object of the recovery process. The process consists of the following activities:

1. AOL Representation Extraction: in this phase an AOL representation is recovered from both design and code through respectively a `CASE2AOL` and a `Code2AOL` extractors;

2. AOL Representation Parsing: an `AOL Parser` builds an AST which subsequent phases rely on. The parser also produces the AOL ASTs for the design patterns of the library;

3. Class Metrics Extraction: a `Metrics Extractor` computes metrics on the class declarations contained in the AOL AST and decorating the AST with such information;

4. Pattern Recognition: a multi-stage `Constraint Evaluator` process localizes design pattern instances where the first stage exploits single class metrics to reduce the search space for the subsequent stages, which apply the structural constraints.

5. GUI: the AOL input file selection, the recognition process, extracted metrics and pattern results visualization are implemented using WEB technology and HTML pages displayed by WEB browsers.

It is worth noticing that the constraints that are applied are all **necessary** conditions for the tuple of classes to be an actual instance of a pattern. For this reason, the output of the recovery process may contain *false positives*, and must be manually inspected and verified by the user.

```
CLASS  Subject
        OPERATIONS
                PUBLIC operation();

CLASS  RealSubject
        OPERATIONS
                PUBLIC operation();
CLASS  Proxy
        OPERATIONS
                PUBLIC operation();

GENERALIZATION Subject
        SUBCLASSES RealSubject;

GENERALIZATION Subject
        SUBCLASSES Proxy;

RELATION  Refers
        ROLES
                CLASS Proxy         MULT One,
                CLASS RealSubject   MULT One
```

Figure 4: AOL proxy representation.

3.1 AOL Representation

AOL has been designed to capture OO concepts both from code and design in a formalism independent of programming languages and tools. The language is a general-purpose design description language, capable of expressing concepts available at the design stage of OO software development. This language is based on the Unified Modeling Language [2], a notation that is becoming the standard in object oriented design. UML is a visual description language with some limited textual specifications. Hence we designed from scratch many parts of the language, while remaining adherent to UML where textual specifications were available. Figure 4 shows the AOL description of the object model correspondent to the *proxy* design pattern in Figure 1. At present AOL covers only the UML part related to class diagrams. Since, for class diagrams, the UML and OMT notations are almost identical, AOL is compatible with OMT designs. An extract of the AOL BNF grammar is given in Appendix A.

3.2 Class Metrics Extraction

Once the AOL parsing phase has been performed, the resulting AST is traversed and decorated with class level metrics computed on the AOL AST. Software metrics are usually exploited to characterize artifact properties. In our approach we are mostly interested in object classes specific properties that can be used to avoid the design pattern recovery process combinatorial complexity explosion. Consequently, software metrics are displayed to the user at the end of the recognition process more as a by-product, since the goal of our process is to recover design patterns.

Software metrics are the essential means which allows to reduce the search space dimension: they have to be functional to the design pattern recovery process. In other words, we are interested in those software metrics which can be directly derived by the design patterns structure or properties and furthermore could be regarded as constraints to be satisfied by candidate patterns extracted from design or code. Hence, the number of relations (aggregations, associations, inheritances) and methods are the key metrics of interest, more precisely, the metrics computed for each class are the following:

- number of public, private and protected attributes;
- number of public, private and protected operations;
- number of association, aggregation and inheritance (or generalization) relations in which a class is involved;
- total number of attributes, methods and relations.

Other metrics, such as depth of inheritance tree or number of derived (directly or not) classes can be easily computed. Anyhow, the set of the above metrics suffices to perform recognition process for the structural design pattern as described in this paper.

3.3 Pattern Recognition

Finding an instance of a structural design pattern involves identifying a set of classes which exhibits the ex-

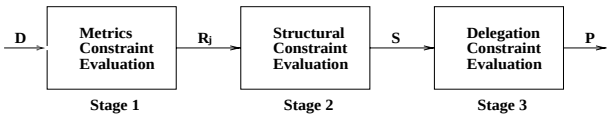

Figure 5: Multi-stage Filtering

act relationships, in terms of aggregations, associations and inheritances, with a given prototype. Operation names correspondence must hold, and finally, some patterns require delegation of operation between two classes through method calls.

Given a pattern $p = (< e_1, \ldots, e_k >, \mathcal{R})$ with cardinality k, where $e_1, \ldots, e_k$ are the involved classes and $\mathcal{R}$ is the set of relations existing among them, a brute force approach to identify all possible pattern candidates in a design containing n classes would require to compute all the dispositions without repetition of the design classes k by k (i.e., $n(n-1)\ldots(n-k+2)(n-k+1)$) and check the validity of the relations in $\mathcal{R}$ for each of them. The resulting worst case complexity is therefore $\mathcal{O}(n^k)$. For example, to search instances of the *adapter* pattern, which involves three classes, in a design containing 200 classes, would require to examine about 8 million possible configurations.

To reduce the complexity a multi-stage recognition process has been adopted, as shown in Fig. 5.

3.3.1 Metrics Constraint Evaluation The input of this stage is the whole design D. Suppose we are searching instances of a given design pattern p. Pattern candidates can be selected, according to class roles and related metrics. Consider the *adapter* pattern: the *Target* must have at least one relation of inheritance, the *Adapter* must have at least one inheritance and one association and the *Adaptee* must participate at least at one association, independently of the specific classes they are related to in these relations, thus such constraints can be verified simply by checking the number of relations in which a class is involved.

Each element belonging to a given design pattern is then characterized by a tuple of metrics, which allows to extract, with linear complexity, a candidate set for each pattern element.

In other words, let $p = (< e_1, \ldots, e_k >, \mathcal{R})$ a pattern (with elements $e_1, \ldots, e_k$) belonging to a pattern collection $\mathcal{P}$, let $M_p =< m_1, \ldots, m_k >$ be the tuple of metrics characterizing pattern p, where each m_i is an array of the metrics listed in Section 3.2 and corresponding to each pattern element, then candidates for the elements e_i are elements of the set:

$$C_i \stackrel{\text{def}}{=} \{x | x \in D \ \wedge \ \forall m_i \in M_p : \forall m_{ij} \in m_i \ m_{ijx} \geq m_{ij}\}$$

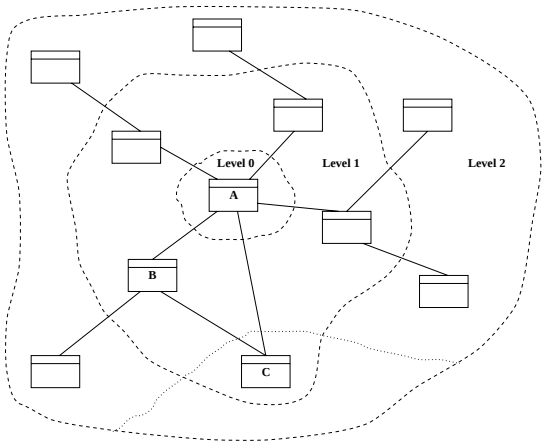

Figure 6: Relation metrics based filtering

Clearly, the same design element x could be in more than one C_i. However, actual pattern instances are surely within the tuples extracted from the set collection $C_1, \ldots, C_n$.

To further reduce the search space, let's choose one of the C_i sets, for example the set C_{min} with minimum number of elements; e_{min} (the corresponding pattern element) can be regarded as a node in a graph representing the pattern, where the remaining $n-1$ pattern constituents must be reachable from C_{min} elements in a number of steps constrained by the pattern structure. As depicted in Figure 6 this could be regarded as a neighborhood property defining a shortest path relation among pattern constituents. Reduced candidate sets $R_j(y)$ with $j \neq min$, $j = 1, \ldots, n$ and $y \in C_{min}$ can be defined as:

$$R_j(y) \stackrel{\text{def}}{=} \{x | x \in D \ \wedge \ ShPath(x, y) = ShPath(e_j, e_{min})\}$$

where $ShPath(x, y)$ is the shortest path between two classes x and y in a design, measured as the number of relations on the design that must be traversed to reach y from x, independently on the nature of the relations. Figure 7 depicts the relation between $R_j(y)$ sets, C_i and C_{min} for a design pattern consisting of three elements.

Notice that, even if theoretically speaking, $R_j(y)$ computation could require the all pairs shortest path computation, with cubic complexity [1], in practice, since design patterns are micro-architectures, typical values for $ShPath(e_j, e_{min})$ are limited and usually below 4. As a consequence the complexity observed in practical cases is almost linear with the size of C_{min}.

3.3.2 Structural and Delegation Constraints Evaluation The previous stages have only imposed, for each pattern constituent, the presence of certain pattern rela-

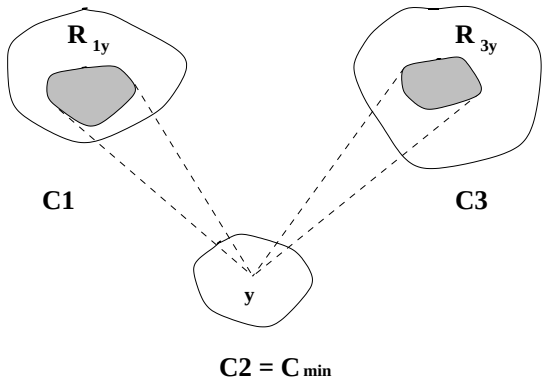

Figure 7: Relations between R sets and C sets

tionships, without worrying about the exact identity of the classes involved in the relations.

In the last two stages, exact design pattern constraints are applied to tuples $< r_1(y) \ \ldots \ y \ \ldots \ r_n(y) >$ where $r_k(y) \in R_k(y)$. In the first stage structural constraints are applied and the set S of the candidate tuples formed with the sets $r_k(y) \in R_k(y)$ that satisfy those constraints is produced as output. Let $R = \{p | p =< r_1(y) \ldots r_n(y) > \wedge y \in C_{min} \wedge \forall r_i(y) \in R_i(y)\}$, a tuple in S satisfies all the relations in $\mathcal{R}_\mathcal{S}$, where $\mathcal{R}_\mathcal{S}$ is a subset of $\mathcal{R}$ representing all structural relations (i.e. without delegation constraints):

$$\begin{aligned} S \ &\stackrel{\text{def}}{=} \ \{x | x =< x_1 \ldots x_n > \in R \ \wedge \\ &\forall \quad r \in \mathcal{R}_\mathcal{S} : \ r(e_p, e_s, \ldots e_t) \Rightarrow r(x_p, x_s \ldots x_t)\} \end{aligned}$$

where p, s and t are generic indices of a subset of pattern elements that participates at a given relation r.

The set S is taken as input of the second stage in which the delegation constraints are checked and the final set P of tuples that satisfy also these constraints is produced as output of the whole process.

Being the `Code2AOL Extractor` capable to safely identify supersets of methods invoked by a given method, verifying the presence of an actual method delegation allows to eliminate discovered patterns that only satisfy structural requirements. For example, consider the *adapter* design pattern in Figure 2. Once *Target*, *Adapter*, *Adaptee* candidates have been identified, we further impose the delegation constraint: the *Adapter* must issue a call from one of his methods to an *Adaptee*'s method.

3.3.3 WEB Interface The entire recognition process has been conceived to fulfill the WWW computation model and to be automated as much as possible: a WWW browser, connects to the WWW server where the pattern recognition process is carried out. The user chooses a design pattern from the design pattern library (known by the recognizer) and, by browsing the local (client side) file system, he/she selects the AOL artifact to be processed. A snapshot of the GUI is shown in Figure 8.

Once the recognition process is completed results are shown in the lower part of the browser page.

4 EXPERIMENTAL RESULTS

The process for extracting design patterns shown in Figure 3 has been completely automated.

A `CASE2AOL Extractor` module has been implemented for the StP/OMT[3] CASE tool, to obtain an AOL specification of internal object models from its repository. In this case the information extracted is completely trustable, in that it really represents design information and no assumption has to be made about the validity of class relationships.

To extract the AOL representation from code a `Code2AOL Extractor` module has been developed, working for the C++ language. Extracting information about class relationships from code may be more difficult than from design and the result might have some degree of imprecision. In fact, there are intrinsic ambiguities, given two or more classes and a relation among them, due to the choice left to programmers implementing OO design. Associations can be instantiated in C++ by means of pointer data members or by inheritance. Furthermore, aggregation relations could result either from templates (e.g., `list<tree>`), arrays (e.g., `Heap a[MAX]`) or pointers data member (e.g.,`Edges=new GraphEdge[MAX]`). In the present work, an aggregation is recognized from code if and only if an instance of an object is stored as a data member of another object or a template or object array data member is declared, even if a pointer could be used to create an object chunk. All the remaining cases, i.e. object pointers and references both as data members and formal parameters to methods, give origin to associations.

Since the above choice may be over-restrictive, structural design patterns containing aggregations, have been represented in two forms: the canonical one and a more flexible (referred to as *soft* version) in which aggregations are substituted by associations, since an aggregation is actually a special form of association.

The AOL specification derived either from the code or from the design is then parsed by the `AOL Parser`, producing an AST representing the object model. The parser also resolves references to identifiers, and performs some simple consistency checking (e.g. names referenced in associations have been defined).

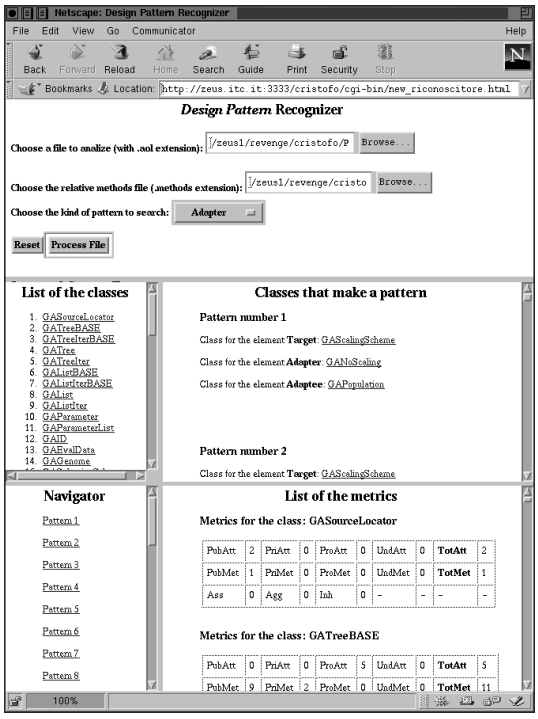

Figure 8: User interface screen-shot.

The `Metrics Extractor` traverses the AOL AST decorating it with the single class metrics used by the subsequent module, the `Constraint Evaluator`, which implements the multi-stage recognition process described in the previous section.

The `AOL Parser`, `Metrics Extractor` and the `Constraint Evaluator` have all been implemented in Java and are plugged into a WEB server by means of a CGI bin.

The extraction process described in the previous sections has been applied on public domain and industrial software.

4.1 Public Domain Code

There are some advantages in analyzing public domain software: source code can be easily obtained and results can be compared with those of related works. Unfortunately, in the authors knowledge, no public domain OO system has both code and design available. Furthermore, even if design is available, quite often it is not documented in terms of design patterns, regardless being it from industry or in the public domain. As a consequence, tedious work is required to assess retrieved patterns.

Table 1 lists the public domain applications that have been analyzed and their main characteristics: lines of code (LOC), number of classes, total number of class attributes and operations and total number of class relationships.

`LEDA` (version 3.4) and `libg++` (version 2.7.2) are two well known C++ libraries of foundation classes. The first, Library of Efficient Data types and Algorithms, is developed and distributed by Max-Planck-Institut für Informatik, Saarbrücken, Germany, the second is part of the GNU Free Software Foundation C++ development environment.

`galib` (release 2.4) is a C++ Genetic Algorithm Library to solve optimizations problems, developed at the Massachusetts Institute of Technology. `groff` (version 1.10) is the GNU Free Software Foundation version of Unix `troff` utility. `mec` (release 0.3) is trace-and-replay program developed by Michael Chastain, *mec-trace* runs a target program and traces it. `socket` (release 1.10) is a library for inter-process communication developed by Gnanasekaran Swaminathan at the University of Virginia .

4.2 Industrial Code

To asses the use of design patterns in an industrial environment and to verify consistency between code and

	galib	groff	LEDA	libg++	mec	socket
LOC	20507	127762	115882	40119	21006	3078
Classes	55	25	187	165	30	29
Attributes	206	89	426	308	94	17
Operations	1111	173	4332	1768	396	281
Aggregations	10	4	166	20	33	2
Associations	97	26	334	91	103	23
Inheritances	36	4	85	95	6	25

Table 1: Public domain code characteristics.

	Sys1	Sys2	Sys3	Sys4	Sys5	Sys6	Sys7	Sys8
LOC	47057	19863	10424	54300	15267	5807	22980	31011
Classes	53	16	17	103	25	5	21	44
Attributes	504	69	14	241	15	21	162	313
Operations	708	347	201	973	158	79	328	609
Aggregations	70	10	4	33	1	2	30	33
Associations	14	1	10	80	4	5	13	23
Inheritances	23	9	13	78	20	4	7	23

Table 2: Industrial code characteristics.

design, a second experiment was conducted on design and code of industrial software for telecommunications. Eight complete systems were analyzed, for which both OO design models and the corresponding code were available. All systems were developed in the same environment, using the same language (C++).

Table 2 shows the size of each system and the other relevant OO characteristics. The sizes are spread fairly evenly across a broad range, from about 5,000 to 50,000 lines of code.

4.3 Discussion of Results

The proposed pattern extraction approach can be assessed in the framework of information retrieval systems including: memory requirements, execution efficiency and retrieval effectiveness. Being the system depicted in Figure 3 almost entirely developed in Java, with the present level of efficiency of Java environments (even exploiting JIT compile technology) time execution and memory requirements result sacrificed in favor of portability. However, even for the `LEDA` or `galib` packages, which are the largest in our program test suite, the extraction process requires only few minutes, which is compatible in the authors' opinion with the pattern recovery conceivable applications.

The most commonly used measure of retrieval effectiveness are *recall* and *precision*. *Recall* is the ratio of relevant document retrieved for a given query over the number of relevant documents for that query in the "database". *Precision* is the ratio of the number of relevant documents retrieved over the total number of documents retrieved. For both the public domain and industrial software, the verification of the actual pattern instances present in the code has been done manually, starting from the results of the extraction process and checking which of the identified patterns was an actual pattern. To do this we based mainly on the names of the involved classes, which gave in most cases the semantics of their usage, but we also checked the source code to verify the exact nature and direction of the relations between classes or the intended functionality of a method. In the present work, each time a doubt on a pattern instantiation arose we considered the Gamma book [4] as the reference in deciding whether or not that instantiation was actually representing a design pattern or not. This process took us about 10 hours, which is reasonable being the total size of the analyzed code about 520 KLOC.

Clearly, all pattern instances present in a software artifact are retrieved by a conservative approach, hence recall is always 100% (i.e. no false negatives are generated), independently on the specific filtering stage of the process in Figure 5. Precision and precision ratios between different stages allow to assess both the overall system and single stage effectiveness.

Table 3 shows the number of pattern candidates after each of the stage filters for the public domain systems analyzed, with respect to the *adapter* and *bridge* (soft version) patterns. Similar results were obtained for the other patterns considered in this paper. Last row gives

	Adapter				BridgeSoft			
System	Initial	Stage 1	Stage 2	Stage 3	Initial	Stage 1	Stage 2	Stage 3
galib	157410	1018	52	33	8185320	19366	46	40
groff	13800	108	7	4	303600	54	0	0
LEDA	6434670	2545	110	8	1183979280	57506	74	23
libg++	4410780	483	15	1	714546360	3889	18	12
mec	24360	312	30	27	65720	120	0	0
socket	21924	269	10	1	570024	4298	27	5
Avg Ratio		2336	21.1	3		22381	516	2

Table 3: Reduction of candidates through the stage filters for the Adapter and Bridge (soft version) patterns.

	galib			groff			LEDA			libg++			mec			socket		
Pattern	ND	D	T	ND	D	T	ND	D	T	ND	D	T	ND	D	T	ND	D	T
Adapter	52	33	6	7	4	0	110	8	5	15	1	1	30	27	19	10	1	0
Bridge	0	0	0	0	0	0	7	7	0	6	0	0	0	0	0	0	0	0
Composite	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Decorator	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Proxy	11	0	0	0	0	0	28	0	0	1	0	0	0	0	0	0	0	0
Time(sec)	92	100	-	4	9	-	207	216	-	17	25	-	6	11	-	12	20	-
Precision (%)	9.2	18.2	-	0	0	-	3.4	33.3	-	4.5	100	-	63.3	70.3	-	0	0	-

Table 4: Results of pattern instances recovery on public domain code: columns labeled with ND report values computed without using the delegation constraint, those labeled with D using the delegation constraint. T columns report the actual patterns present in a system.

the average ratio of the patterns retrieved by two subsequent stages. An effective reduction of several orders of magnitude can actually be observed for each filtering stage. The first filter reduces the input from three to four orders of magnitude, showing the effectiveness of the use of metrics to prune the search space. Without using the metrics based filter, checking directly the structural constraints on systems like LEDA for example would result too demanding in terms of computation time. The structural filter reduces from one to two orders of magnitude, depending on the specific pattern, while the delegation filter reduces from two to three times the input.

pattern	Sys3	Sys4	Sys5	Sys7	Sys8
Adapter	0	6	1	24	1
Bridge	0	6	0	0	0
BridgeSoft	12	9	0	0	1
Time(sec)	4	4	7	4	4

Table 5: Results of pattern instances recovery on industrial design.

Table 4 gives the number of pattern instances retrieved for each system in the public domain test suite for each structural design pattern. Columns labeled with ND report values computed without using the delegation constraint, those labeled with D using the delegation constraint. T columns report the actual patterns present in a system. It can be observed that the most frequently found pattern is the *adapter*, while the *composite* and *decorator* patterns are not present at all in the test suite. However, since both the *composite* and *decorator* contain an aggregation relation between two of their elements and, as stated before, the distinction between aggregations and association in source code is not straightforward given the multiplicity of possible implementations and the inherent ambiguity of pointers in C++, the information extracted by the `Code2AOL Extractor` module could be not completely trustable as regards the capability of distinguishing aggregations from associations. Hence, we also tried to recover the *soft* version of such patterns (DecoratorSoft, CompositeSoft) in which aggregations are substituted by associations. Doing this, we found some instances of these patterns, but when we checked manually the validity of the retrieved pattern candidates, none of them was found as an actual pattern instance.

Last row of the column gives the precision values of the pattern recovery process, both without and with

	Sys1			Sys2			Sys4			Sys5			Sys8		
Pattern	ND	D	T	ND	D	T	ND	D	T	ND	D	T	ND	D	T
Adapter	3	0	0	1	1	0	2	1	0	3	0	0	6	0	0
BridgeSoft	0	0	0	0	0	0	7	0	0	0	0	0	0	0	0
CompositeSoft	0	0	0	0	0	0	42	31	0	0	0	0	0	0	0
Proxy	0	0	0	1	1	1	1	1	1	3	0	0	0	0	0
Time(sec)	4	8	-	4	8	-	7	14	-	4	8	-	4	8	-
Precision (%)	0	100	-	50	50	-	33	50	-	0	100	-	0	100	-

Table 6: Results of pattern instances recovery on industrial code.

the delegation constraint. Precision is computed over all patterns, by summing the numbers of each column ND,D and T and then computing T/ND and T/D. As clearly shown by the table, by imposing also the delegation constraint, precision increases, at the expense of a limited increase of execution times (on average 55%). Removing the systems which do not contain design pattern instances (`groff` and `socket`) for which precision is obviously zero, we obtain a 35% average precision increase.

As regards the industrial systems, both design and code have been analyzed. Design information has been recovered from the corporate database using the `CASE2AOL Translator` we developed for StP/OMT. Delegation constraint could not be checked on design because the information about the methods called by each method at design stage was not available. Therefore, in Table 5 the retrieved patterns correspond to the output of the structural constraint evaluation stage. In the table, the systems for which no pattern instance has been found have been removed. The same holds for the patterns for which no instance has been found in any of the systems. In this case the corresponding row has been removed. In general, patterns seem seldom used in the design of such systems.

Table 6 shows the pattern candidates identified from the source code of each of the eight industrial systems analyzed, respectively without and with the Delegation stage filtering. Also here, systems and patterns with no instances have been removed from the table. As in Table 4, last row give the precision for each system over all patterns: also in this case, BridgeSoft and CompositeSoft instances have not been considered in the precision count since we verified in the code that the information extracted by the `Code2AOL Extractor` was correct so there was no reason to relax the aggregation constraint of those patterns into an association constraint. The rare use of patterns found in the design is confirmed when examining the code.

A measure of the effectiveness of the delegation filter in the case of the industrial software is more difficult, given the absence of actual design patterns in most systems, which gives in a precision of 0% before using delegation filter and 100% after[1]. Hence these cases are to be considered as special, degenerate cases. Anyway, in the case of `Sys4` an actual improvement of 17% can be observed.

A comparison of the pattern instances retrieved on the design and those retrieved on code, shows that there is no intersection between these two sets, i.e. all the patterns discovered in the design are actually not present in the code. This can be explained basically for three reasons: first, when working on design we do not have the information about delegation available so we tend to find more patterns than those actually present. Second, often code includes collection of classes reused from libraries or COTS that are not modeled in the design so they are found only in code. Third, often design documents are not completely consistent with the code, in that after code modifications they are not properly updated to reflect the changes and the gap between design and code may become relevant.

5 CONCLUSIONS

A multi-stage approach to extract structural design pattern from object oriented artifacts, design or code, has been presented. With respect to related work [6], the approach works on design and code which are both expressed in AOL, exploits software metrics to reduce search space complexity and makes use of method delegation information to further reduce the cardinality of the set of the retrieved pattern candidates and augment retrieval precision.

Experimentation shows that OO software metrics are essential to reduce the problem search space allowing to achieve acceptable computation times. The first step of the proposed approach entirely relies on a family of well known and widely used OO software metrics. Without

[1]By applying the precision formula a 0/0 ratio results, which corresponds to an indeterminate value. In this context, we considered it a 100% precision since no relevant documents have been retrieved when no relevant documents were present in the database

using the metrics-based filter, computational times will rapidly become unacceptable as the system size grows from small to medium.

Exact structural constraints are then exploited to further reduce the candidate set and finally, by means of method delegation information the precision is remarkably enhanced. As demonstrated in the paper, the delegation based filter effectively improves system performance, proving that the existence of an association between two classes most of the times is not enough to ensure that an actual method call is issued between the two classes as the given pattern may require.

Experiments have been performed on public domain code and on industrial code to assess the approach effectiveness. On both, industrial and public domain code, the approach performs quite well in terms of computational complexity and retrieved patterns, showing on public domain code an average precision of 55%, and an increase with respect to the use of structural constraints alone of about 35%. On industrial code, the number of patterns actually present in the code was so small (2 patterns for 150 KLOC) that a comparison resulted difficult. What we observed was that patterns retrieved from design and code had no intersection, i.e. the design information was not consistent with code as regards patterns traceability.

We developed a distributed system based on Java and WWW technology to automate the design pattern recovery process. It is well known that Java assures portability across platforms at the price of being quite inefficient. However, our multi-stage search space reduction approach allows to maintain response time acceptable.

Future work will be devoted to extend recovery process to other pattern families and to integrate polymorphism and points-to analysis in the present framework. However, as it is shown in the paper, with the presented approach the recovered size of candidate set is reasonably small and can be verified by hand.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introductions to Algorithms*. MIT Press, 1990.

[2] Rational Software Corporation. *Unified Modeling Language, Version 1.0*. 1997.

[3] Interactive Development Environments. *Software Through Pictures Manuals*. 1996.

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

[5] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and Ettore Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, March 1996.

[6] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object oriented software. In *Third Working Conference on Reverse Engineering*, pages 208–215, Amsterdam, The Nederlands, March 1996.

[7] J. Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance*, pages 244–253, Monterey, CA, Nov 1996.

[8] A. Petroni. *Rappresentazione di design orientati agli oggetti ed analisi basata su metriche*. Tesi di Diploma, University of Trento, Trento, Italy, March 1997.

[9] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[10] F. Shull, W. L. Melo, and V. R. Basili. An inductive method for discovering design patterns from object-oriented software systems. Technical report, University of Maryland, Computer Science Department, College Park, MD, 20742 USA, Oct 1996.

A AOL Extended BNF Grammar

```
Metacharacters extended BNF notation:

       {}   means zero or more times
       []   means an optional element, so 0 or 1 time
       ""   means a terminal symbol
       |    means the boolean symbol OR
       ()   means a block of elements, useful to group them

AOL_design_description::= list_AOL_declarations
list_AOL_declarations ::= {AOL_decl ";"}
AOL_decl              ::= class
                        | association
                        | generalization
                        | aggregation

/*------------------------- CLASSES ------------------------*/

class            ::= CLASS class_name scope
                       [ATTRIBUTES attribute_list]
                          [OPERATIONS operation_list]
class_name       ::= id
scope            ::= "{" (EXTERNAL | ABSTRACT) "}"
attribute_list   ::= [attribute {"," attribute}]
attribute        ::= visibility [SHARED]
                       attribute_name ":" type
visibility       ::= PUBLIC
                   | PRIVATE
                   | PROTECTED
                   | UNDEF_SCOPE
attribute_name   ::= id
operation_list   ::= [operation {"," operation}]
operation        ::= visibility [SHARED] operation_name
                        "(" operation_arg_list ")" [":" type]
                           [an_annotation]
```

```
operation_name   ::= id
opeation_arg_list::= [argument {"," argument}]
argument         ::= arg_name [":" type]
an_annotation    ::= "{" (ABSTRACT | other_annotations) "}"
other_annotations::= string

/*------------------------- RELATIONS  ----------------------*/

association      ::= RELATION [relation_name] ROLES roles
                       [ATTRIBUTES attribute_list]
                          [IS_A_CLASS assoc_class_id_ref]
relation_name    ::= id
roles            ::= role "," role {"," role}
role             ::= [NAME role_name] CLASS class_id_ref
                       MULT multiplicity
                         [QUALIFIER qualifier_name]
role_name        ::= id
class_id_ref     ::= id_ref
multiplicity     ::= string | ONE | MANY
                       | ONE_OR_MANY  | OPTIONALLY_ONE

qualifier_name   ::= string
assoc_class_id_ref ::= id_ref

aggregation      ::= AGGREGATION [NAME aggregation_name]
                         CONTAINER role PARTS part_roles
aggregation_name ::= id
part_roles       ::= role {"," role}

generalization   ::= GENERALIZATION
                        [DISCRIMINATOR discriminator_name]
                         super_class_id_ref SUBCLASSES
                           sub_classes_ids

discriminator_name ::= string
super_class_id_ref ::= id_ref
sub_classes_ids  ::= sub_class_id {"," sub_class}
sub_class        ::= id

id               ::= IDENTIFIER
id_ref           ::= IDENTIFIER
```