Object-Oriented Design Patterns Recovery


(*) University of Sannio, Faculty of Engineering - Piazza Roma, I-82100 Benevento, Italy
(**) University of Naples "Federico II", DIS - Via Claudio 21, I-80125 Naples, Italy
(***) Sodalia SpA via V. Zambra 1, 38100 Trento, Italy

I. SUMMARY

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

While for forward engineering the benefits of using design patterns are clear, using reverse engineering technologies to discover instances of patterns in a software artifact (e.g., design or code) may help in several key areas, among which are program understanding, design-to-code traceability and quality assessment.

This paper describes a conservative approach and experimental results, based on a multi-stage reduction strategy using OO software metrics and structural properties to extract structural design patterns from OO design or C++ code.

To assess the effectiveness of the pattern recovery approach, a process and a portable tool suite written in Java, remotely accessible by means of any WEB browser, has been developed. The developed system and experimental results on 8 industrial software (design and code) and 200,000 lines of public domain C++ code are presented.

Keywords

OO design pattern recovery; OO redocumentation; software metrics; traceability

II. 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 al. (Gamma et al., 1995): 23 design patterns were collected and documented by the authors who also presented pattern implementation in Smalltalk and C++.

This paper summarizes and extends an approach for OO design pattern recovery we presented elsewhere (Antoniol et al., 1999a), (Antoniol et al., 1999b): a revised design pattern representation and design pattern recovery constraints are presented. Design pattern recovery was considered an instance of the cliché matching problem and the source of variation identified by L. Wills (Wills, 1992) discussed. Moreover, new results on ET++ (Gamma et al., 1995), a design-pattern-based system are presented and discussed.

The approach relies on a conservative multi-stage reduction strategy based on software metrics to extract OO design pattern candidates from software artifacts, namely the design class diagram or the source code.

While for forward engineering the benefit of using design patterns is clear (Schmidt, 1995), from a program understanding and maintenance perspective a pattern provides knowledge about the role of each class within the pattern, the reason for certain relationships among pattern constituents and/or the remaining parts of a system. In other words, the discovery of patterns in a software artifact highlights rationale of the adopted solution, representing a step in the program understanding process.
and improving documentation. Consequently, in maintenance, the identification of design pattern instances provides insight on software artifact structure and reveals places where changes, reuse, or extensions are expected.

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, reusability and ease of extension. 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 also be reflected in the corresponding code: the extraction of 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.

Design patterns is a relatively young field, few works in program understanding and reverse engineering have addressed design pattern recovery. Kramer and Prechelt (Kramer and Prechelt, 1996) have proposed an approach and developed a system, called Pat, that localizes instances of structural design patterns by means of structural information. It relies on 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. Experimental results on industrial and public domain software were provided. However, a direct comparison with our approach is not possible. The CASE tool does not pre-process source code; furthermore, Pat is case insensitive, hence it retrieves design patterns that do not really exist in the code (Kramer, 1998).

Keller (Schauer and Keller, 1998), (Keller et al., 1999) have developed an approach and a prototype tool for recovering and visualizing both generic and ad-hoc design patterns, given the reverse-engineered source code of a system. The tool supports recovery of design patterns using automatic, manual and semi-automatic design clustering techniques. Knowledge about the existence of design patterns is preserved in a central repository. Patterns are visualized directly in the reverse-engineered source code models through three types of diagrams: the pattern-enhanced class diagram, the pattern-analysis diagram and the pattern-collaboration diagram. Quantitative data on the proposed approach are not available. The tool, at the present, privileges manual semi-automatic recovery process and design pattern visualization.

Shull, Melo and Basili (Shull et al., 1996) have developed an inductive method to help discover custom, domain-specific design patterns in existing OO software systems. The method, however, is performed manually, although it could be greatly assisted by tools.

Antoniol and Tonella (Antoniol and Tonella, 1999) proposed an approach for the inference of recurrent design patterns directly from the code or the design. No assumption is made on the availability of any pattern library, and the concept analysis algorithm was adapted for the pattern inference.

Seemann and Wolff (J. Seemann and Wolff, 1998) presented a pattern-based design recovery approach relying on structural information and class-exchanged messages.

Different approaches (Kontogiannis et al., 1996), (Mayrand et al., 1996), not specifically oriented towards the OO paradigm and design patterns, exploited software metrics to automatically detect design concepts and function clones in large software systems written in procedural languages.

In the approach presented here, software metrics play a central role: a design pattern is represented as a tuple of classes and relations among classes. When examining potential pattern instances, OO software metrics (Chidamber and Kemerer, 1994), (Lorenz and Kidd, 1994) are used to avoid combinatorial explosion in checking all possible class combinations while determining pattern constituents' candidate sets. Pattern structure is then considered 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 constrained 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 a certain number of edges between candidate
classes. Shortest path filtering effectively reduces the number of candidate tuples determined by the first application of software metrics. On these reduced sets, exact structural design pattern constraints are applied.

In our approach, software artifacts, code or design, are mapped into an intermediate representation, called Abstract Object Language (AOL) (Petroni, 1997). This ensures independence from the programming language and the adopted CASE tools. Other pattern recovery approaches (Wuyts, 1998) use language dependent representation to express different high-level structural relationships among classes.

AOL is basically focused on class diagram concepts: classes, methods, attributes and their properties, as well as relations among classes, are modeled. Software metrics, as well as structural properties, are extracted from an Abstract Syntax Tree (AST) produced by parsing the AOL software artifact representation.

Since necessary conditions are applied, our approach is conservative; thus if a pattern is in the code it is surely reported in the results. Consequently, sometimes spurious patterns may be reported: to reduce such false positives, design pattern constraints in terms of method delegation have been exploited. Method delegation means that a class implements one operation by simply calling an operation of another class with 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, implemented as a WWW server, can be accessed by users by means of a WWW browser.

The paper is structured as follows: we first introduce pattern concepts along with two examples. Then we present our recovery process and the pattern matching approach. Finally, we describe the experimental results obtained on industrial and public domain software.

III. DESIGN PATTERNS

The current use of the term pattern is derived from the writings of the architect Christopher Alexander who has written several books on the topic (Alexander et al., 1977); Alexander's books describe architecture and urban planning. However, the key idea is applicable to many other disciplines including software development. Software patterns became popular with the wide acceptance of the book by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Gamma et al., 1995). Since the early work of E. Gamma (Gamma, 1991), there has been a flourish of interest and activities around patterns and pattern languages (Buschmann et al., 1996), (Coppens and eds., 1995), (Vlissides et al., 1996), (Brown et al., 1998).

Beyond the design phase, patterns have been proposed for almost each phase of the development process (Coppens and eds., 1995): requirement patterns, analysis patterns, architectural patterns, design patterns and (code) idioms represent at different abstraction levels proven solutions to recurring problems within certain contexts (another good source of patterns is the Pattern Home Page at http://hillsidesoftware.com/patterns/patterns.html).

This work is focused on patterns at design level: such patterns correspond to well-known and frequently reused design micro-architectures, excluding other higher or lower levels category of patterns. A design pattern description encompasses its static structure, in terms of classes and objects participating in the pattern and their relationships, but also the intent and the participants’ exchanged messages, i.e., the pattern behavior. 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 to create a shared language for communicating insight and experience about these problems and their solutions.

According to the taxonomy 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.

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 information is explicit in their syntactic representation, for the other two pattern families behavioral information must be recovered, which might involve the analysis of messages exchanged and the code of class methods. This also implies that the recovery of creational and behavioral 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/received from other objects.

For these reasons, we focused our recovery effort on five of the seven structural design patterns proposed in (Gamma et al., 1995): the adapter, the proxy, the composite, the bridge and the decorator. The remaining two structural patterns (the facade and flyweight) were not considered since their structure does not provide enough information to retrieve their instances. In particular, the facade pattern consists of a single class while the flyweight pattern has much of its information embedded in the methods and attributes of its participants. Identifying flyweight instances would require identifying a FlyweightFactory, creating and managing flyweights and distinguishing flyweight classes storing intrinsic
and extrinsic states.

Focusing on structural design patterns does not mean that non-structural design patterns cannot be represented and recognized: it simply means that the extraction process does not exploit behavioral information, thus a much higher number of false positives can occur with respect to the structural design pattern category.

In Figures 1 and 2, the UML (Booch et al., 1999) class diagrams of the Bridge and the Adapter patterns are depicted. The Bridge design pattern package is a common technique used to provide flexible and extensible implementations when an abstraction can have several implementations. For example, when implementing a portable Window abstraction in a user interface toolkit (Gamma et al., 1995), by putting the abstraction and the implementation in separate hierarchies, the abstraction and the platform-specific implementation are decoupled. Clients create an Abstraction which is not bound to any ConcreteImplementor, and it is the responsibility of the Abstraction to get the proper Implementor.

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.

IV. RECOVERY PROCESS

The approach relies on the design pattern recovery process represented in Fig. 3. Both design and source code can be the input of the recovery process. The process consists of the following activities:

1. AOL Representation Extraction: in this phase an AOL representation is recovered from design or code through respectively a CASE2AOL or a Code2AOL extractor;
2. Pattern Recognition: a multi-stage Constraint Evaluator process localizes design pattern instances on decorated AOL ASTs;
3. User Interaction: the recognition process, extracted metrics and pattern results visualization are implemented using HTML pages displayed by a WEB browser.

The pattern recovery process is based on: AOL, software metrics computed on AOL ASTs and a design pattern library. In the following subsection we will highlight the key issues of these elements and the related implications.
A. Design Pattern Library

The pattern recovery process relies on a design pattern library, thus there are similarities with the program understanding and architectural recovery approaches based on cliché matching and plan recognition. The AOL intermediate representation is similar to other representations adopted by the program understanding community (Fiutem et al., 1996), (Kozaczynski et al., 1992), (Ning et al., 1994), (Quilici, 1994), (Tonella et al., 1996) while the library has its counterpart in the cliché (Fiutem et al., 1996), (Tonella et al., 1996) and plan collections (Kozaczynski et al., 1992), (Ning et al., 1994), (Quilici, 1994). Our work substantially differs from the above mentioned problem, programming language and tools as well as the approach are different. We adopt a strategy based on design local properties, represented by means of software metrics, to avoid the design pattern recovery process complexity combinatorial explosion. However, an instance of design pattern may appear in several different forms due to variations: we share with the program understanding and architectural recovery community the problems identified by Wills (Wills, 1992). Wills classifies the main sources of variation as: syntactic variation, implementation variation, delocalization, organization variation, redundancy, unrecognizable code and function sharing.

Syntactic variation mostly regards the syntactic level clichés; they are recognized directly with pattern matching on the AST.

The different cliché forms corresponding to the different language constructs that may be used in specifying a cliché are codified as different implementations (Wills, 1992) of the same concepts. A cliché recognizer embodies the knowledge of all the different forms that a certain cliché can assume.

In the design pattern recovery domain, syntactic variability is mostly related to the different ways in which an aggregation can be implemented i.e., by means of pointers or array data members.

Implementation variation is related to the fact that a given concept may be implemented in different ways: an aggregation may be implemented with a list or a set or any other user defined type. Another example is the depth of the inheritance tree between a superclass and a derived class participating in a pattern (see for example the Bridge) it may or may not be exactly one level.

As for syntactic variation, also in this case the different cliché forms are codified as different implementations (i.e. a set of alternative forms) of the same concepts. In other words, a preliminary work is required to enrich a standard design pattern library with the company/project expected implementation variations.

Delocalization (Letowsky and Soloway, 1986), (Rugenber et al., 1995), (Wills, 1992) and organization variation are quite similar issues. They occur when the components of a given cliché are not contiguous but spread over different files in the source code. This phenomenon could cause some problems to a purely syntactically-based cliché recognition. By considering the entire system design we are guaranteed that if an instance of a design pattern is present it will be surely reported.

The redundancy problem occurs when a part of a cliché appears more than once in the same instance of a cliché: by applying only necessary conditions a superset of the actual design pattern is retrieved.

The unrecognizable code phenomenon is also called partial recognition: the recognition system, because of the incompleteness of its cliché library, is not able to classify every line of the source code or class in a design as belonging to some cliché. What happens actually is that a forest of partial matching is generated, and some parts of the program remain uncovered. For the design pattern recovery domain, partial recognition of a program is expected since usually most of a program is devoted to performing the computation related to its application domain while only a small fraction of the design/code is devoted to implementing design pattern structures.

Function sharing is also called optimization (Wills, 1992) or overlapping implementations (Rich and Wills, 1990). It happens when the same portion of code is shared among two or more clichés. Here the implementations of the clichés overlap. This phenomenon does not represent an actual problem given the software metrics approach, the organization of the cliché library and the recognition process. This simply means that in the recovered family of design patterns, some components will be repeated.
CLASS Target
   OPERATIONS
      PUBLIC operation();
CLASS Adapter
   OPERATIONS
      PUBLIC operation();
CLASS Adaptee
   OPERATIONS
      PUBLIC specOperation();
GENERALIZATION Target
   SUBCLASSES Adapter;
RELATION Refers
   ROLES
   CLASS Adapter   MULT One,
   CLASS Adaptee   MULT One

Fig. 4. AOL Adapter representation

The five design pattern were represented in the pattern library in the (Gamma et al., 1995) canonical forms. However, since in OO languages such as C++ an aggregation may be implemented by means of a pointer to a complex data structure, the Bridge, the Composite and the Decorator were represented as soft patterns. In terms of the Gamma book (Gamma et al., 1995) the aggregation was substituted by an association.

B. AOL Representation and Parsing

AOL has been designed to capture OO concepts 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 (Booch et al., 1999) (UML), a notation that is becoming a de facto standard in OO design. UML is a visual description language with some 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 Adapter design pattern in Figure 2. In the present version 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. AOL design representation language ensures independence from any specific programming or proprietary design representation language.

The AOL specification derived either from the code or from the design is 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. An excerpt of the AOL grammar is given in the appendix.

C. Class Metrics Extraction

Software metrics are usually exploited to characterize artifact properties. In our approach we are mostly interested in object class specific properties, properties that can be used to avoid the complex combinatorial explosion of the design pattern recovery process. 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 Section V.

Software metrics are displayed to the user at the end of the recognition process more as a by-product, since the goal of our approach is to recover design patterns.

Software metrics are the essential means 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. 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:

- number of public, private and protected attributes;
- number of public, private and protected operations;
- number of direct subclasses, number of direct superclasses;
- number of association and aggregation relations in which a class is involved;
- total number of attributes, methods and relations.

The set of the above metrics suffices to perform recognition process for the structural design patterns as described in this paper.

Other metrics, such as depth of inheritance tree or number of derived (directly or not) classes can be easily computed and could be used to recover different kinds of patterns or to augment the precision of the recovery process. For example, facade instances could be identified using class relation fan-in and fan-out metrics, searching for classes with high fan-in and fan-out. However, conservativity of the approach would not be guaranteed, since the fan-in and fan-out values are not constrained by the facade pattern representation: applying thresholds on these values could result in missing some true pattern instances.

V. PATTERN RECOGNITION

Finding an instance of a design pattern involves identifying a set of classes which exhibit the exact pattern properties: relationships, behavior and intent. Structure and delegations among classes must hold, behavior in terms of class responsibility distribution and exchanged messages must be checked.

Furthermore, the pattern intent should be elicited. For example, the key difference between Adapter and Bridge lies in the intent. The Adapter is a wrapper, it resolves an incompatibility between already existing interfaces. The Bridge decouples an abstraction from its possibly numerous implementations. Both Adapter and Bridge may accommodate different implementations; however, the Bridge makes objects work together before they are implemented; the Bridge was intentionally designed, e.g., to accommodate new implementations as the system evolves. The Adapter may be useful to avoid rewriting code because it makes things work together after they have been designed.

Intent redocumentation requires discussion with developers or, whenever this is not possible (e.g., dealing with public domain code) at least manually browsing the code attempting to discover the intent of the developers. This last activity unfortunately may introduce a certain degree of subjectivity: we have to decide whether or not a certain structure was generated by chance or represents the solution to the recurring problems for which the design pattern was invented.

In the authors’ experience, the matter is further complicated whenever a class may participate in multiple design patterns; for instance a Bridge and an Adapter may be partially overlapped. In the absence of reliable documentation, during the manual verification we safely assumed that both design patterns could possibly be present. Moreover, whenever they were judged compatible with the pattern definition they were counted as true design patterns regardless of being overlapped.
A pattern may be represented as a tuple \(<e_1, \ldots, e_k>\) of elements \(e_i\) and a set of properties among them. The representation of properties may depend on the property nature: structural properties may be represented by the class diagram or other equivalent textual notations, behavioral properties may be expressed by interaction diagrams or state diagrams, intention may be described by a plain text. As already discussed, we focused our attention on structural design patterns, thus we are interested in the set of structural relations, \(\mathcal{R}\), which almost entirely characterize structural design patterns. Given a pattern \(p = (<e_1, \ldots, e_k>, \mathcal{R})\) with cardinality \(k\), a brute force approach to identify all possible pattern candidates in a design containing \(n\) classes would require computation of 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 a check of the validity of the \(\mathcal{R}\) relations for each of them. The resulting worst case complexity is therefore \(O(n^k)\).

To reduce the complexity, the multi-stage recognition process shown in Fig. 5 has been adopted. Each class in a pattern candidate must exhibit a number of aggregations, associations and inheritances consistent with the given pattern prototype. Hence a first search space reduction can be accomplished by means of Class Level Metrics Constraints. The output of the first block in Fig. 5 is a set of candidate classes for each pattern constituent: classes which exhibit a compatible number of relations with respect to the design pattern searched.

If the inheritance, the aggregation and the association are not distinguished, a design pattern may be thought of as a graph: nodes represent classes and edges represent relations. By disregarding the specific type of traversed edges, each couple of pattern constituents may be characterized by the minimum number of in-between relations: the second stage of Fig. 5 constructs reduced candidate classes filtering out from candidate classes sets those couple elements, belonging to distinct candidate classes, not satisfying the Shortest Path Constraints.

In the last structural filtering stage, reduced candidate classes elements are required to verify the exact correspondence with the pattern prototype relations. This step, highlighted in Fig. 5 as Structural Constraints, requires the explicit construction of all possible tuples; however, the tuple elements are extracted from reduced candidate classes and only a subset of the \(O(n^k)\) combinations is actually examined.

Although in practice reduction factors of several orders of magnitude can be observed by exploiting the process described above, from a theoretical point of view we cannot devise the reduction factor for a generic design. In principle, a design could exist such that no reduction is accomplished by the metrics-based stages. A trivial, but meaningless example is a design in which each class is related in some way with all the other classes. Thus, the effectiveness of the metrics-based stage reduction depends on the topology of a design and it is not a-priori predictable. Section VII will present actual values for the reduction factor on industrial and public-domain programs.

A. Class Level Metrics Constraints

Each element \(e_i\) belonging to a given design pattern \(p\) is characterized by a tuple of metrics which allows to extract, with linear complexity, a candidate set for each pattern element. In other words, let \(p = (<e_1, \ldots, e_k>, \mathcal{R})\) be 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 the array \(m_{ij}(k = 1, \ldots, l)\) of software metrics chosen to describe \(e_i\). For any given pattern \(p\), \(e_i\) candidate class is the subset of classes \(x\), belonging to the design \(D\), that exhibit software metric values \(m_{ij}(x)\) compatible with \(m_{ij}(e_i)\):

\[
C_p(i) \overset{\text{def}}{=} \{x | x \in D \land m_{ij}(x) \bowtie m_{ij}(e_i) \} \forall i \in M_p \land j = 1, \ldots, l
\]

These represent necessary conditions: we used the \(\bowtie\) to state that \(x\) metrics values, \(m_{ij}(x)\), are enforced by the pattern structure. To ensure admissibility, \(\bowtie\) was implemented as \(\geq\) and no upper-
bound values were imposed on relations: we allow each class to be also related with any number of other classes in the design (beyond those participating in the pattern). Clearly, the same design element $x$ could be in more than one $C_p(i)$. However, actual pattern instances are surely within the tuples extracted from the set collection $C_p(1),\ldots,C_p(k)$.

As an example, suppose we want to identify instances of the Adapter pattern in the design of Fig. 6. The given design contains 11 classes, so in principle all the dispositions without repetition of 11 classes 3-by-3 should be checked, i.e. 990 class triplets.

![Diagram](image)

**Fig. 6.** Sample design for Adapter design pattern recovery. Classes are tagged with T, $A'$ or $A''$ if their metrics values are compatible with respectively the Target, Adapter or Adaptee pattern elements.

For simplicity’s sake we restrict the set of metrics used to four of the above metrics, namely the number of direct subclasses, direct superclasses, association and aggregation relations, since they are the most important to identify Adapter instances. The Adapter structure (see Fig. 2) imposes the following constraints: the Target must have at least one subclass, the Adapter must have at least one superclass and one association and the Adaptee must participate in at least one association, independently of the specific classes they are related to in these relations.

Let the Adapter pattern be defined as:

$$
p = (<\text{Target}, \text{Adapter}, \text{Adaptee}>, \\
\{\text{subclass}(\text{Adapter}, \text{Target}), \\
\text{superclass}(\text{Target}, \text{Adapter}), \\
\text{association}(\text{Adapter}, \text{Adaptee})\})
$$

Assuming that the metrics in each array $m_{i,j}$ represent respectively the number of subclasses, superclasses, association and aggregation relations, a triple of classes $<c_1, c_2, c_3>$ is an instance of the Adapter pattern only if:

- $\#\text{subclasses}(c_1) \geq 1$
- $\#\text{superclasses}(c_2) \geq 1$
- $\#\text{associations}(c_2) \geq 1$
- $\#\text{associations}(c_3) \geq 1$

Given the Fig. 6 design, the sets $C_{\text{Adapter}}(\text{Target}), C_{\text{Adapter}}(\text{Adapter}), C_{\text{Adapter}}(\text{Adaptee})$ are:
\[ C_{\text{Adapter}}(\text{Target}) = \{A, C, F\} \]
\[ C_{\text{Adapter}}(\text{Adapter}) = \{B, C\} \]
\[ C_{\text{Adapter}}(\text{Adaptee}) = \{B, C, E, H\} \]

The first stage reduces the example to 24 triples, obtaining a reduction factor of 41.25.

B. Shortest Path Constraints

To further reduce the search space, let us choose the \(C_p(i)\) set with minimum number of elements; \(e_j\) candidates (elements of \(C_p(j)\) and \(i \neq j\)), are related to \(e_i\) candidates (elements of \(C_i\)) by the minimum number of \(e_i - e_j\) in-between relations. For each \(y\) in \(C_p(i)\), a restriction \(R_{p,i,j}(y)\) over the set \(C_p(j)\) is computed as:

\[ R_{p,i,j}(y) \overset{\text{def}}{=} \{x | x \in C_p(j) \land \text{ShPath}(y, x) = \text{ShPath}(e_i, e_j)\} \]

where \(\text{ShPath}(y, x)\) is the shortest path between two classes \(y\) and \(x\) in a design, measured as the number of relations traversed to reach \(x\) from \(y\), independently of the nature of the relations. Each \(y\) in \(C_p(i)\) induces pattern candidate tuples constructed over the reduced candidate sets \(R_{p,i,j}(y)\):

\[ < r_{i,1}, r_{i,2}, \ldots, y, \ldots, r_{i,k} > \quad r_{i,j} \in R_{p,i,j}(y), \quad j = 1, 2, \ldots, k \quad j \neq i \]

Notice that, even if theoretically speaking \(R_{p,i,j}(y)\) computation could require computation of the all pair shortest paths, with cubic complexity (Cormen et al., 1990); in practice, since design patterns are micro-architectures, typical values for \(\text{ShPath}(e_i, e_j)\) are limited and usually below 4. As a consequence the complexity observed in practical cases is almost linear with the size of \(C_p(i)\).

Returning to our Adapter pattern recovery example, the \(C_{\text{Adapter}}(i)\) set would be the Adapter set i.e., \(C_{\text{Adapter}}(\text{Adapter})\), which contains two classes, B and C. By choosing \(C_{\text{Adapter}}(\text{Adapter})\), given the Adapter pattern structure, Targets and Adaptees belong to the neighborhood of each class in the \(C_{\text{Adapter}}(\text{Adapter})\) (Fig. 7)) reachable in one step:

\[ R_{\text{Adapter, Adapter, Target}}(B) = \{A, C\} \]
\[ R_{\text{Adapter, Adapter, Adaptee}}(B) = \{C\} \]
\[ R_{\text{Adapter, Adapter, Target}}(C) = \{A\} \]
\[ R_{\text{Adapter, Adapter, Adaptee}}(C) = \{B, E\} \]

Thus the set of all possible triples also satisfying these constraints is:

\[ R = \{< A, B, C >, < A, C, B >, < A, C, E >\} \]

Notice that, for the design patterns considered in this paper, a class is not allowed to play more than one role, i.e., \(< C, C, B >\). The reduction factor depends on the choice of the \(C_p(i)\) set. Different classes in a pattern imply different neighborhoods for computing the \(R_{p,i,j}(y)\), according to the topology of the pattern and of the design analyzed. The heuristic of choosing the \(C_p(i)\) set with the minimum number of elements does not assure the highest reduction factor, since such choice could lead to consideration of larger neighborhoods. A different selection criterion could be to choose the \(C_p(i)\) set which corresponds to the pattern element whose distances from the other pattern elements are minimal. However, also this criterion does not guarantee the optimality.
C. Structural Constraints

The exact design pattern structural constraints are applied to tuples computed in the previous step: \(< r_{i,1}, r_{i,2}, \ldots, y, \ldots, r_{i,k} >\); the set \(S\) of candidate tuples is constructed as:

\[
S \overset{\text{def}}{=} \{ z | z = < r_{i,1}, r_{i,2}, \ldots, y, \ldots, r_{i,k} > \land \\
\forall \mathcal{T} \in \mathcal{R} : \mathcal{T}(e_q, e_s, \ldots e_t) \Rightarrow \mathcal{T}(r_{i,q}, r_{i,s}, \ldots, r_{i,t}) \}
\]

where \(q, s\) and \(t\) are generic indices of a subset of pattern elements that participates in a given relation \(\mathcal{T} \in \mathcal{R}\).

With respect to the Adapter pattern recovery example, imposing the exact structural relation constraints in terms of inheritance, aggregation and association does not remove any candidate. However, from Fig. 2, another structural constraint must hold: Target and Adapter classes must have at least one operation, with the same name, which may be abstract in the Target and is implemented by the Adapter. Thus, with reference to Fig. 8, class B cannot play the Adapter role since it does not have common operations with Target. The resulting set \(S\) of tuples satisfying also the structural constraints for the Adapter pattern search is thus:

\[
S = \{ < A, C, B >, < A, C, E > \}
\]

Only structural properties were applied: the above constraint requires the Adapter candidate to override a Target method; that is, there must exist at least a Target method implemented by the Adapter. The semantic of the overridden method is not considered, and we can not guarantee that the overridden methods implement a wrapper.
D. Delegation Constraints

When detailed information on methods call is available, the set $S$ is taken as input of the last stage in which the delegation constraints are checked and the final set tuples produced. Delegation constraints can be represented as proposed in (Antoniol and Tonella, 1999). For example, let us assume that a design pattern requires that $e_i$ invokes method $M$ of $e_j$. This can be represented by the relation $(i, j)_{\text{call}(M)}$, where $c(M)$ is a label for the call to method $M$. If the invocation is associated with a specific method from a class – rather than the class as a whole – the label can be modified into $\text{call}(M_1, M_2)$ to indicate that method $M_1$ from the first class calls method $M_2$ from the second class. For example, the Adapter constraint can be expressed as: $(\text{Adapter}, \text{Adaptee})_{\text{call(Adapter} \rightarrow \text{op()}, \text{Adaptee} \rightarrow \text{op())}}$.

The Code2AOL Extractor is able to safely identify supersets of methods invoked by a given method; verifying the presence of an actual method delegation allows the elimination of discovered patterns that satisfy only 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.

If we assume that method calls are those represented in Fig. 8, then only one triple, $<A, \text{C, E}>$ may be an Adapter instance, since C calls the op method exported by E, while B does not call any methods of C, and vice-versa.

\begin{center}
\begin{tikzpicture}
  \node (A) at (0,0) {A \hspace{1cm}};
  \node (B) at (0,-1) {B};
  \node (C) at (1,-1) {C \makebox[0pt][r]{\quad \text{op()}}};
  \node (D) at (2,-1) {D};
  \node (E) at (3,0) {E \makebox[0pt][r]{\quad \text{op()}}};

  \draw[->] (A) -- (E);
  \draw[->] (B) -- (C);
  \draw[->] (C) -- (E);
  \draw[->] (C) -- (D);

\end{tikzpicture}
\end{center}

Fig. 8. Design example with method call information for delegation-based stage.

The final set of Adapter pattern candidates is thus:

$$P = \{<A, C, E>\}$$

Having applied only necessary structural and delegation conditions, it is not assured that the semantics of $<A, C, E>$ is actually consistent with an Adapter: designer intention, classes behavior could differ from the Adapter design pattern; nevertheless, the instance found is possibly the only Adapter instance in the given design.

VI. System Implementation

The process for extracting candidate design patterns shown in Figure 3 has been completely automated. A CASE2AOL Extractor module has been implemented for the StP/OMT CASE tool (StP Manuals) to obtain an AOL specification of internal object models from its repository. In this case the
information extracted is completely trustworthy, 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 and works 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<street>`), 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 a template, an object array, or an instance of an object is declared as data member. 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, it does not take into account the *syntactic variation* and *implementation variation* identified by Wills (Wills, 1992); structural design patterns containing aggregations have been represented in two forms: the canonical and a more flexible one (referred to as *soft version*) in which aggregations are substituted by associations, since an aggregation is actually a special form of association.

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.

**A. 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.

Due to time and space constraints, as well as confidentiality and security reasons, code is not sent over the network. The **Code2AOL** program can be downloaded to extract, on the client machine, AOL representations.

As a response to the connection, a multi-frame HTML page is returned and a CGI bin command script on the server machine is activated. The CGI bin script is responsible for the successive interaction phases; in the upper frame of Figure 9, it implements two functionalities: a pattern selection menu, where the user chooses a design pattern from the design pattern library, and a file selection box where, by browsing the local (client side) file system, he/she selects the AOL artifact to be processed.

In the lower part of the browser of Figure 9, once the recognition process is completed the four frames are automatically reloaded. The upper-left frame contains the scrollable “List of the classes” found in the AOL file. Each list entry, i.e., class name, is an anchor linked to the corresponding metrics table (lower-right frame). Clicking on the class name forces the lower-right frame to center around the class correspondent metrics table.

The scrollable list of recovered design patterns is contained in the upper-right frame. Pattern constituents class names are linked, as “List of the classes”, to the metrics table frame.

Finally, the lower-left frame is devoted to the scrollable list of recovered design pattern instances. The names of this list are linked to first class of the relative design pattern in the upper-right frame; thus the user easily navigates between design patterns, classes and metrics.

**VII. EXPERIMENTAL RESULTS**

The extraction process described in the previous sections has been applied on public-domain and industrial software. 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, to the authors’ knowledge, no public-domain OO system has both code and design available. As a consequence, the recovery process was applied both to public-domain and industrial code but only on industrial design.

**LEDA** (version 3.4) and **TBB** (version 2.7.2) are two well-known C++ libraries of foundation classes. The first, Library of Efficient Data types and Algorithms, was developed and is 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 optimization problems, developed at the Massachusetts Institute of Technology. **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.
To assess the use of design patterns in an industrial environment and to verify consistency between code and design, a second experiment was conducted on design and code of industrial software for telecommunications. Eight components of a complete system were analyzed. All components were documented with OMT design and were developed in the same environment by the same teams using the same language (C++).

As shown in Table I and Table II, software analyzed in both experiments have sizes and other OO characteristics spread fairly evenly across a broad range.

### Table I

**Public-domain code characteristics.**

<table>
<thead>
<tr>
<th></th>
<th>galib</th>
<th>LEDA</th>
<th>lib++</th>
<th>mec</th>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>20507</td>
<td>115882</td>
<td>40119</td>
<td>21006</td>
<td>3078</td>
</tr>
<tr>
<td>Classes</td>
<td>55</td>
<td>208</td>
<td>167</td>
<td>32</td>
<td>29</td>
</tr>
<tr>
<td>Attributes</td>
<td>206</td>
<td>426</td>
<td>308</td>
<td>94</td>
<td>17</td>
</tr>
<tr>
<td>Operations</td>
<td>916</td>
<td>4610</td>
<td>2863</td>
<td>345</td>
<td>288</td>
</tr>
<tr>
<td>Aggregations</td>
<td>10</td>
<td>166</td>
<td>20</td>
<td>33</td>
<td>2</td>
</tr>
<tr>
<td>Associations</td>
<td>97</td>
<td>334</td>
<td>96</td>
<td>138</td>
<td>23</td>
</tr>
<tr>
<td>Inheritances</td>
<td>36</td>
<td>85</td>
<td>95</td>
<td>6</td>
<td>25</td>
</tr>
</tbody>
</table>

### A. Results and Discussion

The proposed pattern extraction approach can be assessed in the framework of information retrieval systems (Frakes and Baeza-Yates, 1992) including: memory requirements, execution efficiency and retrieval effectiveness. For 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 were 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 a few minutes, which is compatible, in the authors opinion, with the pattern recovery conceivable applications.

### Table II

**Industrial code characteristics.**

<table>
<thead>
<tr>
<th></th>
<th>Sys1</th>
<th>Sys2</th>
<th>Sys3</th>
<th>Sys4</th>
<th>Sys5</th>
<th>Sys6</th>
<th>Sys7</th>
<th>Sys8</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>47057</td>
<td>19863</td>
<td>10424</td>
<td>54306</td>
<td>15267</td>
<td>5807</td>
<td>22980</td>
<td>31011</td>
</tr>
<tr>
<td>Classes</td>
<td>53</td>
<td>16</td>
<td>17</td>
<td>193</td>
<td>25</td>
<td>5</td>
<td>21</td>
<td>44</td>
</tr>
<tr>
<td>Attributes</td>
<td>50</td>
<td>19</td>
<td>14</td>
<td>241</td>
<td>15</td>
<td>21</td>
<td>162</td>
<td>313</td>
</tr>
<tr>
<td>Operations</td>
<td>708</td>
<td>347</td>
<td>201</td>
<td>973</td>
<td>158</td>
<td>79</td>
<td>328</td>
<td>609</td>
</tr>
<tr>
<td>Aggregations</td>
<td>70</td>
<td>10</td>
<td>4</td>
<td>33</td>
<td>1</td>
<td>2</td>
<td>30</td>
<td>33</td>
</tr>
<tr>
<td>Associations</td>
<td>14</td>
<td>1</td>
<td>10</td>
<td>80</td>
<td>4</td>
<td>5</td>
<td>13</td>
<td>23</td>
</tr>
<tr>
<td>Inheritances</td>
<td>23</td>
<td>9</td>
<td>13</td>
<td>78</td>
<td>20</td>
<td>4</td>
<td>7</td>
<td>23</td>
</tr>
</tbody>
</table>

The most commonly used measures of retrieval effectiveness are *recall* (Frakes and Baeza-Yates, 1992) and *precision* (Frakes and Baeza-Yates, 1992). *Recall* is the ratio of relevant documents 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.

Although the number of documents in the “database” (i.e. design patterns present in a given system) is not known a priori, since the approach being conservative, we need only to assess the number of relevant documents retrieved for a given query (i.e., number of true design patterns recognized). Precision is then computed as the ratio of true design patterns over the number of retrieved pattern candidates.
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. In the present work, each time a doubt on a pattern instantiation arose, we considered the book (Gamma et al., 1995) as the reference in deciding whether or not that instantiation was actually representing a design pattern. This process took us about 3 hours, which is reasonable being that the total size of the analyzed code was about 400 KLOC.

### TABLE III

**Reduction of candidates through the stage filters for the Adapter design pattern.**

<table>
<thead>
<tr>
<th>System</th>
<th>Initial</th>
<th>Metrics Constraints</th>
<th>Shortest Path and Structural Constraints</th>
<th>Delegation Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>gaib</td>
<td>139419</td>
<td>1018</td>
<td>25</td>
<td>5</td>
</tr>
<tr>
<td>LEDA</td>
<td>6434670</td>
<td>2215</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>libg+</td>
<td>4100760</td>
<td>483</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>mec</td>
<td>21360</td>
<td>312</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>socket</td>
<td>21924</td>
<td>269</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Avg Ratio</td>
<td>2388</td>
<td>857</td>
<td>4.4</td>
<td></td>
</tr>
</tbody>
</table>

In a conservative approach, as presented in this paper, all pattern instances present in a software artifact are retrieved: hence perfect recall is obtained. However, recall and precision of the recovery process depends on the exactness of the AOL design input. When the AOL representation is extracted from design it faithfully describes the corresponding design information, while, when extracted from code, associations could be recognized instead of aggregations. By introducing *soft patterns*, we are guaranteed that we do not miss pattern instances due to aggregation and association misclassifications. Of course, more pattern instances are extracted: the approach guarantees a 100% recall at a price of lower precision values. Renouncing the admissibility could enhance precision but would also cause generation of false negatives. The desired trade-off between precision and recall depends mostly on the user objectives: for some tasks a perfect recall might be preferable because the user does not want to miss any of the actual instances, while for other tasks a higher precision would be preferred since the cost of discriminating relevant items from false positives could be high.

### TABLE IV

**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.**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>gaib</th>
<th>LEDA</th>
<th>libg+</th>
<th>mec</th>
<th>socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>ND</td>
<td>D</td>
<td>T</td>
<td>ND</td>
<td>D</td>
</tr>
<tr>
<td>Bridge</td>
<td>23</td>
<td>4</td>
<td>14</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Facade</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Composite</td>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Decorator</td>
<td>56</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tic (sec)</td>
<td>33</td>
<td>62</td>
<td>111</td>
<td>241</td>
<td>12</td>
</tr>
<tr>
<td>Precision (%)</td>
<td>11.1</td>
<td>1100</td>
<td>269</td>
<td>46.9</td>
<td>60.0</td>
</tr>
</tbody>
</table>

Precision and precision ratios between different stages allow to assess both the overall system and single stage effectiveness. Table III shows the number of pattern candidates after each of the stage filters for the analyzed public-domain system, with respect to the adapter design pattern. Notice that, since no actual design patterns were identified in socket, the socket package was removed from precision evaluation and is not reported in Table III and successive tables.

Similar results were obtained for the other patterns considered in this paper. The last row gives 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, class-level metrics plus shortest path, reduces the input by three to four orders of magnitude, showing the effectiveness of the use of metrics to prune the search space. The structural filter reduces by one to two orders of magnitude, depending on the specific pattern, while the delegation filter reduces the input by two to three times.

### Table V

Results of pattern instances recovery on industrial design.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Sys3</th>
<th>Sys4</th>
<th>Sys5</th>
<th>Sys6</th>
<th>Sys8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adapter</td>
<td>0</td>
<td>6</td>
<td>1</td>
<td>21</td>
<td>1</td>
</tr>
<tr>
<td>Bridge</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BridgeSoft</td>
<td>12</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Time (sec)</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Table IV 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 patterns manually classified as true patterns. It can be observed that the most frequently found pattern is the *adapter*. In our approach the aggregation constraint was relaxed in order to find *composite* and *decorator* patterns. In other words, the *composite* and *decorator* patterns we found contained are *compositeSoft* and *decoratorSoft*.

The last Table IV row gives the precision values of the pattern recovery process, results are showed with and without 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 also imposing the delegation constraint, we obtain a 60% average increase in precision, at the expense of a limited increase in execution times.

As regards the industrial system, 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 V the retrieved patterns correspond to the output of the structural constraint evaluation stage. Design patterns were not retrieved in several components, neither in the design nor in the code: patterns seem seldom used. Table V does not include the component for which no pattern instance has been found.

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; that is, all the patterns discovered in the design are not actually present in the code. This can be partially explained 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, code often includes a collection of classes reused from libraries or COTS that are not modeled in the design. Third, design documents are often not completely consistent with the code, in that after code modifications, they are not properly updated to reflect the changes, hence the gap between design and code may become relevant.

### Table VI

Structural design pattern recovery on ET++.
B. ET++

It can be argued that very few instances of design patterns were retrieved. This happens because the analyzed systems were not conceived taking into consideration design patterns. To verify this conjecture we analyzed and extracted design patterns from ET++: the software system that inspired the Gamma, Helm, Johnson and Vlissides book (Gamma et al., 1995). ET++ is an OO framework containing basic data structure, support for input/output and user interface building blocks; it evolved from the initial implementation to the currently available 3.0 release which can be downloaded from ftp.ubilab.ubs.ch. ET++ release 3.0 contains about 100 KLOC of C++ code, with 704 classes; due to software evolution many changes have been made since early ET++ release described in (Gamma et al., 1995): design pattern syntactic variation, implementation variation as well as other software changes took place. For example, the stream class hierarchy changed: StreamDecorator, ASCII7Stream, CompressingStream no longer exist; moreover the Stream class is not part of any of the derived classes; so the decorator is no longer there.

As Table VI clearly shows, ET++ contains a very high number of pattern candidates and a relevant work was required to manually verify them. ET++ contains instances of multiple hits: 12 Bridges were overlapped with Adapters and a Proxy was overlapped with a Decorator. The overlapping phenomenon and the patterns were not documented in (Gamma et al., 1995); out of 12, 7 were classified as implementing a Bridge overlapped to an Adapter, and both the Decorator and the Proxy were judged as true design patterns. The classification was to some extent subjective: three expert OO developers were asked to classify and reach a consensus on the pattern candidates. Given the low number of multiple hits, the precision result is only slightly affected.

VIII. Conclusions

An approach and a multi-stage process to extract structural design patterns from object oriented artifacts, design or code, has been presented. With respect to related work (Kramer and Prechelt, 1996), (Schauer and Keller, 1998), our 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. Unlike (Schauer and Keller, 1998) we do not address specific techniques and tools for pattern visualization: pattern instances are simply represented as class tuples in a textual format.

Experimentation shows that OO software metrics are extremely helpful to reduce the problem search space allowing to achieve acceptable computation times. The first step of the proposed process relies entirely on a family of well-known and widely used OO software metrics. Without using the metrics-based filter, computational times 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 the method delegation filter, 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 time, 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 effectiveness of the approach. 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 81% and an increase using delegation of about 60% with respect to the use of structural constraints alone. In general, we found very few patterns, even if the system were not designed explicitly using design patterns. On average, on public-domain applications, we found 20 patterns every 100KLOC while the patterns actually present were 12 every 100KLOC. On industrial code, the number of patterns actually present in the code was so small (2 patterns for 130 KLOC), a comparison was difficult. What we observed was that patterns retrieved from design and code
had no intersection, that is, the design information was not consistent with code as regards pattern traceability. On the contrary, ET++ contains about 280 design patterns per 100 KLOC; such a high discrepancy does not have any explanation except for the totally different application (a framework), programmer culture and skill.

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 us to maintain acceptable response time. In fact, response time on the public-domain test suite was on average about 5s/KLOC; the analysis time for the largest analyzed system is of the order of a few minutes.

Results produced by the tool could be useful in program comprehension and maintenance in several ways. Localizing design pattern instances in a design provides explicit knowledge about each class and object participant, their interactions and their underlying intent. Finding adapter pattern instances; (for example, signal classes (the Adaptee) which are used in multiple contexts requiring different interfaces) at the same time proxy patterns indicates the existence of large objects or remote objects which are accessed through a surrogate object. Bridge patterns decouple abstractions from their implementations so they show program points in which change or reuse is expected. The knowledge of such information facilitates the task of a maintenance programmer, who is analyzing a program's design, to accomplish adaptive or perfective maintenance, or is seeking reusable parts of a program.

In this paper we focused on the recovery of design patterns and we did not try to evaluate how useful a design recovery tool would be for actual program understanding and maintenance tasks. Such an evaluation would require empirical in-field studies, which are left as future work. Future work will also be devoted to extending the recovery process to other pattern families and to integrating polymorphism and points-to analysis in the present framework. However, as is shown in the paper, with the presented approach the size of recovered candidate sets is reasonably small and can be verified by hand.

APPENDIX

I. AOL Extended BNF Grammar

Conventions: The metacharacters follow the extended BNF notation

{ } means zero or more times
[] means an optional element, 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 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_abbreviation] \
operation_name ::= id
operation_arg_list ::= [argument "." argument]
REFERENCES


