Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Using Design Pattern Clues to Improve the Precision of Design Pattern Detection Tools

Francesca Arcelli Fontana[a] Marco Zanoni[a] Stefano Maggioni[a]

a. DISCo — Dipartimento di Informatica, Sistemistica e Comunicazione University of Milano-Bicocca Viale Sarca, 336 — Building U14 20126 — Milan, Italy

Abstract Design pattern detection, or rather the detection of structures that match design patterns, is useful for reverse engineering, program comprehension and for design recovery as well as for re-documenting object-oriented systems. Finding design patterns inside the code gives hints to software engineers about the methodologies adopted and the problems found during its design phases, and helps the engineers to evolve and maintain the system. In this paper, we present the results provided by four different design pattern detection tools on the analysis of JHotDraw 6.0b1, a well-known Java GUI framework. We show that the tools generally provide different results, even while evaluating the same system. From this observation, we introduce an approach based on micro structures detection that aims to discard the false positives from the detected results, hence improving the precision of the analyzed tools results. For this purpose we exploit a set of micro structures called design pattern clues, which give useful hints for the detection of design patterns.

Keywords

Software re-engineering; software maintenance; reverse engineering; design recovery; design pattern detection.

1 Introduction

Reverse engineering and reengineering activities are very important to support software maintenance, comprehension and evolution [CCI90, MJS+00]. One of the objectives of reverse engineering consists in reconstructing the architecture of target software systems [DP09] and detecting their fundamental components to consequently reveal their relationships. The retrieval of these components would make the restructuring and maintenance phases easier, as the system would not be seen as a single monolithic

Francesca Arcelli Fontana, Marco Zanoni, Stefano Maggioni. *Using Design Pattern Clues to Improve the Precision of Design Pattern Detection Tools*. In Journal of Object Technology, vol. 10, 2011, pages 4:1–31. doi:10.5381/jot.2011.10.1.a4

structure, but as a set of smaller interacting components that are usually easier to manage.

In this context, particular relevance is given to *design patterns* [GHJV95, Coo98]. Design patterns are useful both in the design phases, as they are a sort of directive to solve a problem in a given context, and in the reverse engineering phases, as the detection of such elements in a system gives to the software maintainer hints about the issues faced during the system design. The detection of design patterns, or better the detection of structures that match design patterns, gives therefore useful information about the organization of a system, and it can reveal the logical foundations of a given implementation. Knowing the potential uses of design patterns helps during a design recovery phase by outlining possible design problems and decisions. Moreover, design pattern detection is important during the re-documentation phases of a system, in particular when the system documentation is scarce, incomplete or not up-to-date to the current system version.

For what concerns the vocabulary used in the design pattern detection community, Guéhéneuc et al. [Gué07] proposed the use of the term *motifs* to express the solutions advocated by the design patterns. These solutions are implemented in systems as micro architectures, where a micro architecture is composed of classes, methods, fields, and relationships having the structure similar to one or more motifs. The authors distinguish between patterns and motifs because patterns encompass information that is not readily available for their identification. While design patterns describe good solutions to common and recurring problems, design motifs are the solutions which software engineers introduce in their program design [KGH10]. This aspect leads to the implementation of personalized solutions, given by different design motifs, which in the literature are usually called *variants* [BMR+96, KB96, SvG98]. The variants problem concerns the possibility of potentially infinite implementations of the same pattern, hence making its detection a difficult task. Guéhéneuc et al. [Gué07] suggest that strictly speaking, we cannot use the terms "design pattern identification" or "detection", but rather the instantiation and identification of micro-architectures similar to some motifs; thus, they propose to use the term "design motif detection" for the process traditionally called design pattern detection. In the remainder of this paper, we will be consistent with this vocabulary.

Many different approaches and tools for the detection of design motifs exist, and they normally give different results when analyzing the same system. The comparison among these tools is not easy because a standard benchmark platform is not yet available, even if some benchmarks have been proposed [FHFG08, Ess10, AFZC10]. The tools exploit different detection techniques; some of them are based on the identification of micro structures [NNZ00, SS03] inside source code, that can be used as the basic bricks onto which the detected motifs can be built.

By micro structure we mean a kind of program construct or arrangement that has limited scope and size, and that can be represented as a property of a program element (class, method, attribute, etc) or as a relation between a couple of elements. An example of this kind of micro structure is given by Elemental Design Patterns (EDPs) used in the SPQR approach [SS03].

In the context of these micro-structures, we have introduced the concept of design pattern clues [Mag06], which constitutes a new category of micro-structures, with the aim to identify particular hints useful in a design motif detection process.

In this paper we analyze how design pattern clues and some EDPs can be used and exploited as elements that can help software engineers to validate or discard the results

provided by tools for design motif detection. This aspect is interesting because the results of design motif detection are generally characterized by low precision. Therefore, having a means to help the engineers to verify the produced output could increase the precision of the results. In this context, we do not take into consideration the recall values of the single tools, as our objective in this work is to check the retrieved design motifs and to discard the false positives that have been eventually detected by the tool.

The validation and refinement process (which is explained in detail in section 5) is based on the definition of rules for each of the design patterns to be analyzed, where we define which clues or EDPs should or must belong to any design motif we want to validate. The instances obtained through the design motif detection tool are checked according to the defined rules, to validate or discard them according to the clues and EDP that can be identified, giving the possibility to classify them as correct or wrong. To test our approach we have manually validated the instances obtained by the design motif detection tool and compared them with the results of our refinement process.

The principal aims of this paper are:

- to show how design pattern clues and EDPs can be used to refine the results obtained by other tools for design motif detection;
- to analyze the improvement of the precision of these results through the detection of clues and EDPs in the found design motifs.

We describe and analyze in the paper the results obtained with four tools for design motif detection: Design Pattern Detection Tool [TCSH06] (which we will refer to as DPD Tool from now on), PINOT [SO06], Web of Patterns [DE07] and FUJABA [NNZ00]. We will show the results we obtained by analyzing the JHotDraw framework and will observe how our refinement process helps in reducing, for some design patterns, the number of false positives produced by the detection tools.

The paper is organized as follows: in section 2 we describe selected related work on design motif detection; in section 3 we introduce design pattern clues, which can be useful for the refinement of third-party-identified pattern motifs, and we give an example of clues and EDPs in a design motif. In section 4 are discussed the results obtained by the four considered tools on the analysis of JHotDraw 6.0b1, showing the differences among them. In section 5 we describe the principal phases of our refinement process, in section 6 we describe the pattern refinement rules, explaining the steps followed during the refinement process. In section 7, we provide examples of the application of the rules on some detected instances and in section 8 we summarize and discuss the refinement results obtained on the instances identified by the detection tools. Finally, section 9 traces the conclusions of our work and discusses possible future improvements.

2 Related Works

Several tools and approaches for design motif detection have been proposed in the literature, but we emphasize that undertaking a comparison among design motif detection tools is a difficult task. Benchmark proposals for the evaluation of design motif detection tools have been presented [AFZC10, FHFG08, Ess10], but a standard benchmark platform is not yet available. Another initiative where a repository of pattern-like micro-architectures called P-MARt [GA08] has been defined, serving as

a baseline to assess the precision and recall of motif identification tools; in order to support also benchmarking a common exchange format, called DPDX [KBH+10], for design motif detection tools has been proposed.

The work more correlated to our research is described by Kniesel et al. [KB09], who compare different design motif detection tools, and they propose a novel approach based on data fusion, built on the synergy of proven techniques, without requiring any re-implementation of what is already available. They show how a pattern can be a witness for the existence of another pattern. Their approach differs from our approach because they exploit the results of the tools in a data fusion approach to better improve both precision and recall, while we exploit micro structures to validate the results of the tools and to improve only the precision. It would be interesting in the future to analyze if our approach based on micro structures could be used to improve their approach.

We briefly introduce below several approaches or tools developed for design motif detection and we start by describing the four tools that we have considered for our experimentation of the refinement process, while the others are cited as examples of known design motif detection tools.

Pinot [SO06] is a modification of Jikes, IBM's Java compiler, developed to detect various design motifs based on static rule-based analysis. The authors present a reverse engineering oriented reclassification of the GoF design patterns into different categories: patterns provided by the programming language, syntax-based patterns, semantic based patterns and domain-specific patterns.

Tsantalis [TCSH06] proposes a design motif detection methodology, based on similarity scoring between graph vertices. The approach has the ability to also recognize motifs that are modified from their standard representation and exploits the fact that motifs reside in one or more inheritance hierarchies, reducing the size of the graphs to which the algorithm is applied. Evaluation on three open-source projects demonstrated the accuracy and the efficiency of the method described in the paper.

Web of Patterns [DE07] uses an approach to the formal definition of design motifs based on the web ontology language (OWL). The authors present their prototype which accesses the motif definitions and detects motifs in Java software. The tool connects to a pattern server, downloads and scans the patterns, translates them into constraints, and resolves these constraints with respect to the program to be analyzed.

FUJABA [NNZ00, NSW+02], exploits a kind of micro structure, called sub-pattern, and fuzzy logic combined with FUJABA Abstract Syntax Graphs (ASGs) to cope with two different types of pattern variations: design variants and implementation variants. The former is addressed using ASGs, by modelling various design variants with different graphs, and implementation variants are handled by defining a set of fuzzy rules together, that determine the degree of belief that a motif is found at a certain location in the program.

SPQR [SS03] exploits another kind of micro structure called Elemental Design Patterns (EDPs) and a system for logical calculus, the ρ-calculus, to detect DPs. The authors claim that the tool can detect several design motifs in C++ systems.

DeMIMA [GA08] is an approach to semi-automatically identify micro-architectures that are similar to design motifs in source code and to ensure the traceability of these micro-architectures between implementation and design. DeMIMA consists of three layers: two layers to recover an abstract model of the source code, including binary class relationships, and a third layer to identify design motifs in the abstract model. Through the use of explanation-based constraint programming, DeMIMA ensures

100% recall on an experimentation on five systems.

SPOOL [KSRP99] stands for Spreading Desirable Properties into the Design of Object- Oriented, Large-Scale Software Systems. The authors outline three possible ways of detection: manual, automatic, and semi-automatic, the first two of which are supported in SPOOL. Automatic recovery is implemented through queries to a previously generated repository.

The Pat system [KP96] transforms C++ source code into PROLOG facts and matches them against pattern definitions given as PROLOG statements. The approach is based on first-order logic and constraint solving techniques. The authors claim that this system can detect many motifs without missing any and with few false positives. Although we cannot verify the truth of these assertions, we can easily imagine the high computational costs of this approach. In addition, only header files are examined, so no behaviour is available for them.

PTIDEJ-Décor tool [MG07] uses constraint solving with explanation. Explanation consists in first detecting instances matching DP definitions exactly and then, by relaxing some constraints, entities that are less and less similar to DPs.

The MAISA tool [PKG+00] uses a library of motifs defined as sets of variables, representing the motif's roles, and unary or binary constraints over them. A solution to the constraint satisfaction problem is a possible instantiation of these variables. To be able to detect instances which do not exactly correspond to the definition, it is possible to relax the definition by removing some constraints, but the number of candidates tends to increase quickly. A similar approach has been used in the Columbus tool [FBTG02], in which motifs are defined by using an XML based language called Design Pattern Markup Language (DPML) and searched for in an Abstract Semantic Graph (ASG) generated by the tool itself.

Several other tools and approaches have been proposed and described in the literature [Tai07, AFC98, Wuy98, Vok06, SvG98, AFRG+06].

3 Micro-structures for Design Motif Detection

Different kinds of micro-structures have been proposed in the literature, with different objectives, like design motif detection, identification of common programming techniques and extraction of architectural relationships. As far as design motif detection is concerned, the approaches based on the recognition of micro-structures inside the code and other input generally exploit source code static analysis.

The relevance of micro-structures in the general detection process can be important. To obtain an effective detection process with good rates of precision and recall, micro-structures should help to identify those aspects that are fundamental for the presence of motifs inside the code.

We have previously compared two types of micro-structures for design motif detection [AFMRT05], focusing on their relevance in the identification of GoF design patterns [GHJV95] and we realized that the detection of one kind of micro structure is not enough to detect design motifs. Other techniques have to be used as for example fuzzy logic, constraint solving or data mining as we saw in section 2 with SPQR exploiting elemental design patterns and ρ-calculus and FUJABA exploiting sub-patterns and fuzzy logic.

Hence we decided to study and propose a new category of micro-structures, named *design pattern clues*, with the aim to identify hints, conditions and concepts useful for design motif detection. The aim to introduce a new kind of micro structure was

to try to complement the information that can be extracted through other micro structures, to obtain information to be used in a design motif detection approach. So we started to analyze design pattern motifs, extracted from examples and real systems, and we tried to understand, for each specific design pattern, what information the other micro-structures (in particular EDPs) were not able to capture. We put together the causes of bad detection for each pattern, and we tried to specify more precisely what all these causes had in common. Then we tried, when possible, to specify how to detect these causes in the code without ambiguity; clues were the output of this process. After having done this work for each pattern, we also made a further analysis of the clues coming from different patterns in order to avoid duplication and let their definition become more stable. It was a bottom-up task, done starting from the real implementations of patterns rather than from their theoretical definition.

Recently we made an extensive comparison [AFMR11] of four micro-structures types, precisely: Design Pattern Clues, Elemental Design Patterns [Smi02], Sub-Patterns [NSW+02] and Micro Patterns [GM05], with the aim to provide in the future a unified catalog of micro-structures. In this paper our focus is to analyze how two kind of micro-structures, in particular clues and EDPs, can be used to refine the results of detection obtained through different tools.

Many differences exist between clues and EDPs. For example, clues are strictly focused on formalizing constructs that are typical in the implementation of design patterns, while EDPs depict basic programming constructs (like object instantiations or method invocations) that are independent from the presence of design patterns inside the system to be analyzed. Clues and EDPs share the same detail level, as in general they can be detected by the analysis of single statements and elements of a class, like method invocations or field declarations.

EDPs capture object-oriented best practices and are independent of any programming language; clues aim to identify basic structures peculiar to each design pattern. In spite of the differences between them, these micro-structures can be used both for the construction and the detection of design motifs.

Examples of both clues and EDPs are given in the next sub-sections.

3.1 Design Pattern Clues

We already introduced design pattern clues for creational patterns [Mag06] and we introduce here the clues for all the other categories [GHJV95]. We have defined design pattern clues by manually analyzing design pattern architectures and sample implementations, identifying basic structures that are peculiar for each single pattern. Clues give us more information related to the single roles that constitute the various patterns. Roles [KB09] are duties that can be fulfilled by program elements (type, methods, ...) relations (inheritance, association, etc) and collaborations in a design pattern. The information about roles is exploited in combination with EDPs, which on the contrary tend to be useful to identify relationships among the pattern roles, leading to the extraction of pattern motifs. An example of a design pattern and of its clues is reported in subsection 3.2.

Currently, we have identified 46 design pattern clues (definitions are available in Appendix A), subdivided into the following nine categories:

Class Information: collects clues that can be identified analyzing a class declaration or that characterize a single class;

Multiple Class Information: collects clues that can be identified by the comparison among two classes (or more) and their contents;

Variable Information: gathers information about particular variables;

Instance Information: contains clues regarding particular instances of a certain class, and one clue representing a controlled instantiation mechanism;

Method Signature Information: collects clues which are identifiable analyzing the signature of a method;

Method Body Information: contains those clues that can be identified by only analyzing the body of any kind of method;

Method Set Information: collects clues whose details can be deducted analyzing the whole set of methods the involved classes declare and implement;

Return Information: includes those clues regarding various possible return modes from a method;

Java Information: collects clues which are strictly bound to the Java language.

All 46 clues can be automatically detected from source code, as they are representations of implementation issues which can be easily understood through an analysis of it. The clue catalogue is reported in Appendix A. Each design pattern clue is automatically recognizable from source code with the use of an ad-hoc tool called *Micro-Structures Detector (MSD)* [AFZ11, Ess09]. Design pattern clues are extracted through AST matching, that is described in detail in section 5.

3.2 Example of Micro-structures in a Design Motif

Considering the structural design pattern category, we propose the basic implementation of the Composite design pattern and we discuss the design pattern clues and EDPs that can be identified in it. The description of design pattern clues can be found in Appendix A, while the description of EDPs are avilable in a separate catalog [Smi02, Ess10]. Next we show a simple Java implementation of the Composite design pattern:

```
public abstract class Component{
  public abstract void operation();
  public void add(Component c){}
  public void remove(Component c){}
}

public class Composite extends Component{
  private List<Component> components = new Vector<Component>();

  public void operation(){
    for (Component c : components)
      c.operation();
  }

  public void add(Component c){
```

```
    components.add(c);
  }

  public void remove(Component c){
    components.remove(c);
  }
}

public class Leaf extends Component{
  public void operation(){ ... }
}
```

3.2.1 Design Pattern Clues

Seven design pattern clues can be found in this basic implementation of the Composite:

Abstract cyclic call: (method signature information category) in the `Composite` class the method `operation()` invokes the `Component.operation()` abstract method within a cycle; therefore the `Composite` class contains an *Abstract cyclic call*;

Component method: (method signature information category) the two methods `Component.add()` and `Component.remove()` are instances of this clue, as they receive as parameter an object belonging to the same class;

Node class: (method set information category) `Composite` extends a class (`Component`) declaring *Component method*s and overrides them;

Leaf class: (method set information category) `Leaf` extends a class (`Component`) declaring *Component method*s without overriding them;

Same interface container: (instance information category) `Composite` contains a list of `Component`s, which are objects that share the same interface with the `Composite` class; so `Composite` has a *Same interface container* clue;

Multiple redirections in family: (method body information category) the *Redirect in Family* EDP is detected inside a cycle (into the `Composite.operation()` method), therefore it is supposed to work on a set of elements. In this case, the `operation()` method is invoked on each `Component` object belonging to the `Component`s list.

3.2.2 Elemental Design Patterns

In the implementation above of the Composite the following EDPs have been detected: one *Abstract Interface* EDP states that the `Component` class declares an abstract method, and consequently is an abstract class; two *Inheritance* EDPs connect the `Composite` and `Leaf` class through an extension relationship; a *Create Object* EDP can be found in `Composite`, where a list of `Component`s is instantiated; finally a *Redirect in Family* EDP is detected in the `Composite.operation()` method. This method invokes a method with the same signature belonging to `Composite`'s superclass.

4 Detection of Design Motifs through Four Design Motif Detection Tools

Several tools for design motif detection exist such as those cited in section 2. Each one is based on a different approach, adopts different strategies to detect motifs, and in general can identify only a subset of the defined motifs. In this paper we focus on the evaluation of four known tools, namely DPD Tool [TCSH06], PINOT [SO06], FUJABA [NNZ00] and Web of Patterns [DE07] and we report the results obtained by these tools in the analysis of the JHotDraw 6.0b1 framework [Bra]. We focus our attention on this system because the development of JHotDraw demonstrates the practical application of design patterns in a software project. For each class of the system, the documentation indicates if it eventually belongs to a certain pattern or set of patterns, and which role it plays within the patterns it takes part in. Thus, we have a precise indicator about what patterns have been implemented, how many instances of a certain pattern can be found in the system, and which classes take part in which patterns.

Table 1 summarizes the results produced by the four considered tools on JHotDraw 6.0b1, in terms of the number of occurrences they are able to detect for each pattern. From the analysis of this table, several points can be observed. First of all, no tool is able to detect or provide techniques for the identification of the whole set of design patterns defined by Gamma et al. [GHJV95].

A second consideration is related to the different results obtained by the tools in the detection of the same pattern. As it can be noticed, there is no pattern for which the tools return the same number of occurrences. Even if this would have been the case, it could have been possible that the detected instances differed from one tool to another in terms of classes realizing each single instance. The difference in the results obtained by the tools is due to the different detection strategies and sometimes to the slightly different pattern definition interpretations that lead to include or exclude certain mofits during the detection. As the different tools identify a considerable number of false positives, hence with low precision rates, we propose in this paper an approach aimed at discarding false positive instances through the help of micro structure based refinement rules. Our approach aims at improving the precision of design motif detection tools, therefore obtaining results that are more close to the actual design pattern motifs implemented in the analyzed systems.

5 Pattern Instances Refinement Process

We now describe the steps of the refinement process that we propose in this paper and the steps of the validation of our process. Figure 1 summarizes both of them. In the figure, the grey rectangles represent the tools that we have developed involved in the refinement process. Rounded rectangles are related to the needed artifacts and representations, while straight rectangles represent the pursued activities and operations.

In the first phase, design motifs are identified from an analyzed system through the different detection tools; then to validate our refinement process, they are manually evaluated to check which of them are correct instances, and which are false positives. The manual evaluation is based both on the system documentation (in the case it traces the existence of patterns within the system), and on personal experience and knowledge about patterns. Manual evaluation has been performed by several software

Table 1 – Results of the design motif detection process obtained by four tools on the analysis of JHotDraw 6.0b1

Pattern category	**Pattern name**	**Design Pattern Detection Tool**	**PINOT**	**FUJABA**[2]	**Web of Patterns**
Creational	Abstract Factory	n/a	n/a	2	14
	Builder	n/a	n/a	n/a	n/a
	Factory Method	2	34	2	n/a
	Singleton	2	0	0	1
	Prototype	3	n/a	n/a	n/a
Behavioural	Chain of responsibility	n/a	n/a	0	n/a
	Command	23[1]	n/a	n/a	n/a
	Iterator	n/a	n/a	10	n/a
	Mediator	n/a	n/a	n/a	n/a
	Memento	n/a	n/a	11	n/a
	Observer	3	n/a	n/a	n/a
	State	29[1]	3	0	n/a
	Strategy	29[1]	51	0	n/a
	Template Method	5	2	31	1
	Visitor	1	1	0	0
Structural	Adapter	23[1]	5	26	1
	Bridge	n/a	n/a	0	n/a
	Composite	1	4	0	1
	Decorator	3	5	0	n/a
	Façade	n/a	n/a	8	n/a
	Flyweight	n/a	n/a	0	n/a
	Proxy	n/a	n/a	n/a	n/a

[1]Design Pattern Detection Tool identifies the Adapter and the Command as being the same pattern. This is due to the fact that the two patterns actually present an identical structure. The 23 results are to be considered comprising both Adapter and Command instances. The same considerations are applicable to the State and Strategy patterns, which the tool recognizes as being the same pattern.

[2]The instances detected by FUJABA are expressed in terms of similarity to the actual correct implementation of the pattern. For each instance, a percentage value is given, which represents the grade of similarity of the instance to the actual pattern. For brevity, we don't report here the similarity values. Anyway, each of the identified instances is at least 80% close to the real pattern.

engineering master students. In the future, the evaluation step could be supported by an automated comparison with a repository of valid instances (work in progress [Ess10]). The results obtained by the detection tools are represented in different forms, depending on the tool. In general, the tools provide graphical or textual representations, where each role is associated with a particular class.

In the second phase, each instance to be refined by the corresponding micro-structure-based rule described in the next section, must be represented in a graph-based form, where each node represents a class (having a role in the pattern) and each edge represents the set of micro-structures relating two classes. A design pattern role is the label, defined in the pattern definition, that we can give to each class belonging to a design pattern instance, to specify its job (the role) within the overall pattern. In this phase of the refinement process, we define the roles for each detected instance: each role identified by the tool is translated to a graph node. The graph structures are defined in appropriate XML templates (one for each kind of pattern). Each element of the template corresponds to a role, and has to be completed with the actual class or classes playing that specific role. The class-role association is currently supported by a manual process, but we are working on the development of the automation of the process, at least for the most common tools for design motif detection [Ess10, AFZC10].

In the third phase, the micro structures are detected through the *Micro Structures Detector* Module (MSD module) and the defined graph nodes obtained in phase two constitute the first input for the *Design Pattern Refiner* (DPR module), a graphical front end devoted to the validation of motifs. For each instance, starting from the graph nodes and from the micro-structures identified by the *Micro-Structures Detector* on the subject system, the *Refiner* generates the actual micro structure based motifs.

Hence, through the *Micro-Structures Detector*, the micro structures are extracted by visitors that traverse an AST representation of the source code, each of them returning instances of the micro structures if the analyzed classes or interfaces actually implement them. The information is acquired statically and is characterized by 100% rate of precision and recall. This value is due to the fact that these kinds of structures are meant to be mechanically recognizable, *i.e.* there is always a 1-to-1 correspondence between a micro structure and a piece, or a set of pieces, of code. In other words, the micro structures are not ambiguous (as on the contrary design patterns may be), and once a micro structure has been specified in terms of the source code details that are used to implement it, the micro structure can be correctly detected. Our actual implementation of the MSD module is based on AST analysis, and exploits the JDT library, which provides all the classes and interfaces that can be used to access a project's ASTs. The micro structures are collected by a set of visitors, invoked sequentially on the ASTs of the classes constituting the project and visit only those nodes that may contain the information they are able to detect (for example, the visitors that look for method call EDPs only analyze nodes that represent a method invocation, *i.e.* instances of the `MethodInvocation` class). The results coming from the visitors, *i.e.* the instances of basic elements that have been found inside the project, are then stored in an XML file.

The Design Pattern Refiner associates roles to nodes in the graph according to the output of the motif detection tool; the refiner then applies the appropriate refinement rule to each instance, to check which of the micro-structures defined by the rule are actually implemented. Based on this application on the micro-structures which are peculiar for the pattern and on the necessary pattern's micro-structures, in the

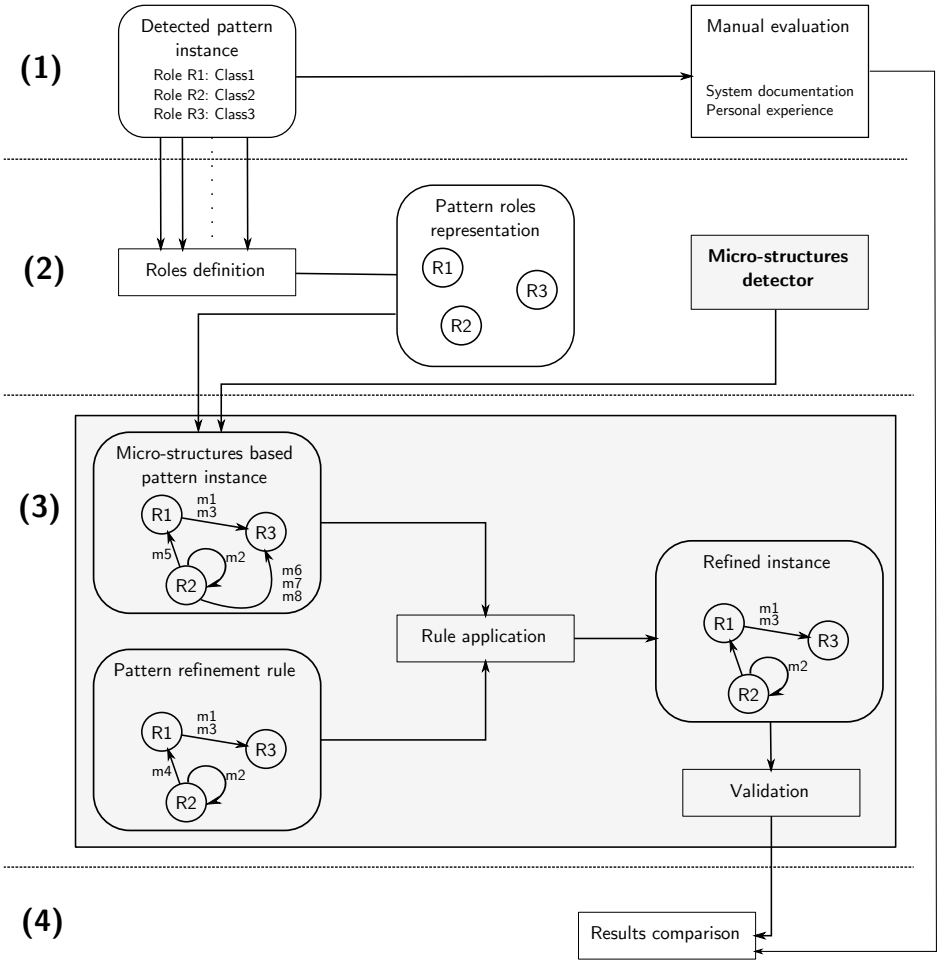

Figure 1 – An overview of the refinement process

validation step each instance is either accepted as a true pattern instance or classified as a false positive and discarded.

In order to validate whether the refinement process provides the same results or not, we have to compare the obtained results with those of the manual evaluation. We plan to automate this phase in a future integration of the refinement process just described with the benchmark platform for design motif detection evaluation we are currently developing [Ess10].

6 Definitions of Refinement Rules

The rules we define in this section aim to increase the precision values of design motif detection tools. Given a subject system S, we indicate with *tp (true positives)* the number of real design pattern instances implemented in S and identified by the motif detection tools; *fp (false positives)* indicates the number of instances which have been detected by the tool on S but which are not correct realizations of the subject pattern,

while fn *(false negatives)* indicates the number of pattern instances implemented in S which cannot be identified by the detection tool. The precision P of a design motif detection tool is computed as $P = tp/(tp + fp)$, and indicates what proportion of the detected instances are correct implementations. The more P is close to 1, the more the tool is precise and the fewer false positives the tool detects. The recall R is defined as $R = tp/(tp + fn)$, and indicates what proportion of the actually implemented pattern instances the tool is able to recover. A widely used combination of these two indicators is the F-measure, calculated as $F = 2 \cdot P \cdot R/(P + R)$; it is equivalent to the harmonic mean of precision and recall, where precision and recall have the same weight. To balance differently P and R in the F-measure another formula is used: $F = (1 + \beta^2) \cdot P \cdot R/(\beta^2 \cdot P + R)$; in particular, when $\beta = 1$ the formula is the same of the first version. Augmenting β results in more weight for recall, lowering β wheigts more precision.

To increase the precision of the results provided by available tools, we propose to analyze the pattern instances identified by them with the use of refinement rules that are based on micro-structures that can be detected in each pattern. Micro-structures are as the same level of abstraction as design patterns, but, due to their nature and definition, each of them can be assigned to a single role inside the pattern it is a hint for. We have analyzed the structures and typical implementations of design patterns, in order to assign to each role the micro-structures (clues and EDP), that characterize them.

The rules are based on two kinds of micro-structures: the EDPs, which are useful to recover and define the structures of the patterns, and the design pattern clues, which are more useful to characterize the single pattern roles. Each refinement rule for a given design pattern is represented as a graph $G = (V, E)$, where V represents the set of classes that constitute the pattern, *i.e.*, the pattern roles, and E represents the set of clues and EDPs that connect the various roles and that are peculiar for the pattern. In this context, each clue or EDP can be seen as a relationship between two roles (therefore it is depicted as an edge between two nodes of the rule graph), or as a relationship between a role and itself (hence depicted as a self-link on the role node).

Refinement rules are not to be considered sufficient conditions for the correctness of pattern instances, but only necessary conditions. The evaluation of the rules will prove to be useful in the refinement process, as ambiguous instances will be discarded or accepted based on the verification of the conditions in the rules.

We now describe the rules for the validation of the patterns that are recognizable by the majority of the considered tools. Necessary clues and EDPs are underlined. The clues and EDPs not underlined are not part of the refinement rules, but they are optional conditions, which are often verified on design motifs, but are not mandatory. We include them, as a support for manual validation.

We will define the rules and discuss the refinement process for the following patterns: the Abstract Factory, Factory Method and Singleton creational patterns, the Adapter, Composite and Decorator structural patterns, and the Template Method and Visitor behavioural patterns, because these patterns are recognizable by the majority of the tools. Even if three out of the four considered tools are supposed to detect instances of the State and Strategy patterns, we will not provide refinement rules for them, as we have not identified any particular micro-structure which could help in their validation. This is due to the strictly behavioral nature of these patterns, which cannot be represented in the form of elements that can be statically detected from source code analysis. In future work we will study if some clues can be defined, exploiting

dynamic analysis, which for example will point to some particular interaction sequence or will reveal concrete objects in polymorphic method calls.

Figures 2, 3 and 4 describe the refinement rules for the considered structural design patterns, namely the Adapter, Composite and Decorator patterns. In order to understand the clues used into the refinement rules, please refer to the Appendix A for the definitions of each clue. For example in the Composite refinement rule (an example of clues and EDPs in a Composite motif is given in subsection 3.2, the Leaf role needs a *Leaf class* clue in order to be acceptable and the Component role needs a *Component Method* clue and could have an *Abstract Interface* EDP, but the latter is not mandatory; the Composite role must present the *Same interface container*, *Multiple redirections in family* and *Node class* clues, but can have also an *Abstract cyclic class* clue. The meaning of the refinement rule is that a motif is valid when the classes covering all its roles present the mandatory micro-structures, otherwise the motif has to be discarded.

All the figures in sections 6 and 7 report the name of the micro-structures on the graphs edges. When a micro-structure's name ends with EDP, it is an Elemental Design Pattern; in all other cases it is a Design Pattern Clue.

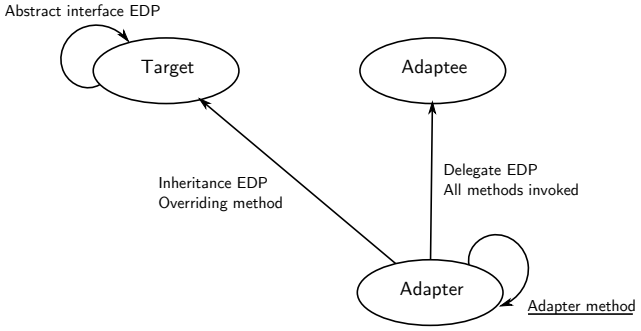

Figure 2 – Adapter refinement rule. Roles = {Target, Adaptee, Adapter}. The *Adapter method* clue requires that the Adapter overrides the methods provided by the Target in order to be able to invoke the methods declared by the Adaptee.

Figures 5, 6 and 7 report and explain the refinement rules for the considered creational patterns. For example in the Abstract Factory rule there are four roles (Abstract Factory, Abstract Product, Concrete Factory, Concrete Product), and we require that the Concrete Factory and the Concrete Product roles must be connected by a *Concrete Product Getter* clue: if an instance having two classes associated respectively to these two roles does not present that connection, it is considered wrong, and therefore discarded.

Figures 8 and 9 introduces the refinement rules for the considered behavioral patterns, namely the Template Method and the Visitor.

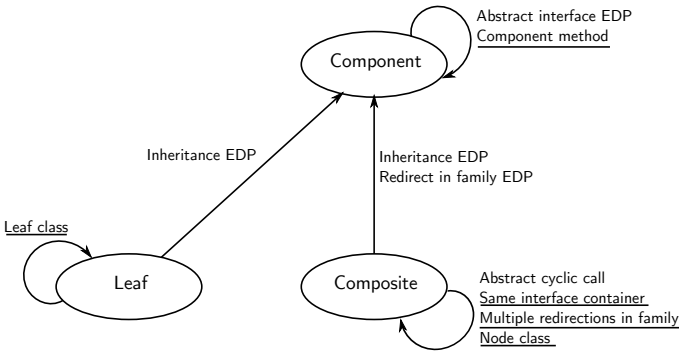

Figure 3 – Composite refinement rule. Roles = {Composite, Component, Leaf}. With *Same interface container* the Composite must have a collection of Component elements, and *Node class* ensures it overrides the component methods defined by the Component. Finally, with *Multiple redirections in family* the component methods are invoked on all the components belonging to the collection. *Component method* requires the Component defines component methods.

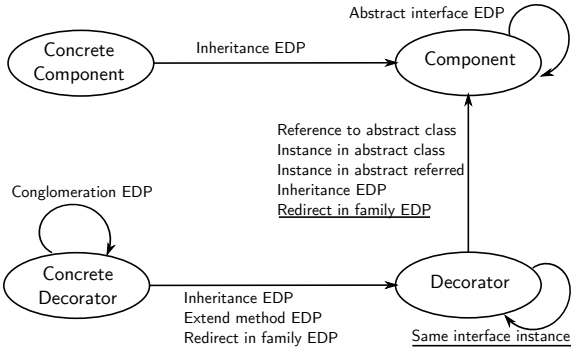

Figure 4 – Decorator refinement rule. Roles = {Component, Decorator, Concrete decorator, Concrete component}. *Same interface instance* clue requires the Decorator maintains a single reference to the Component. *Extend method* EDP ensures the concrete decorators enrich the methods defined by the Decorator.

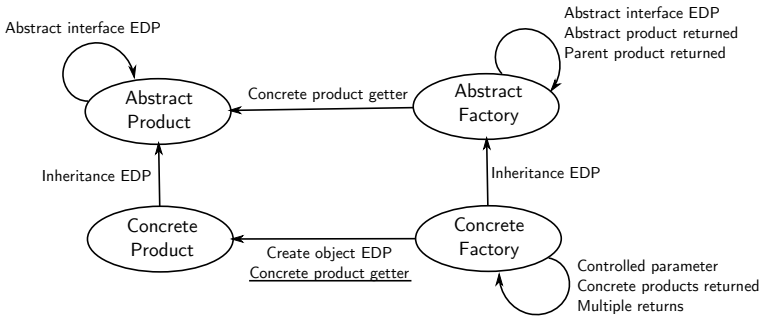

Figure 5 – Abstract Factory refinement rule. Roles = {Abstract Factory, Abstract Product, Concrete Factory, Concrete Product}. The *Concrete product getter* requires that the Concrete factory implements at least one method which returns an instance of the Concrete product.

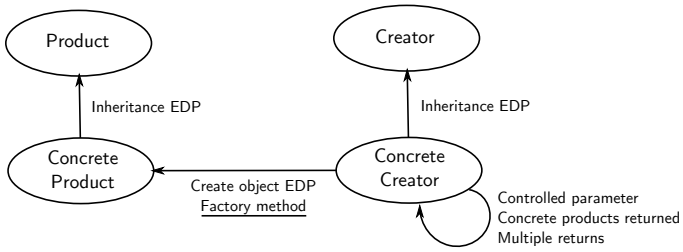

Figure 6 – Factory Method refinement rule. Roles = {Creator, Concrete Creator, Product, Concrete Product}. The *Factory Method clue* assures the existence of a method which creates instances of the Concrete product within the Concrete creator.

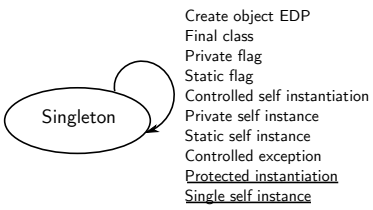

Figure 7 – Singleton refinement rule. Roles = {Singleton}. *Protected instantiation* avoids the creation of instances from external classes. *Single self instance* requires the presence of only one instance for the Singleton class.

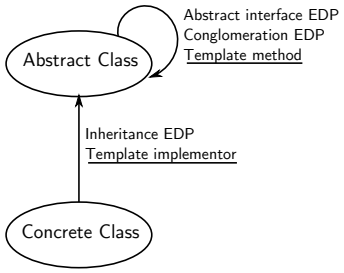

Figure 8 – Template Method refinement rule. Roles = {Abstract class, Concrete class}. The *Template Method* clue requires that in the Abstract class a concrete method invokes abstract methods inside its body. The *Template implementor* clue instead requires that the Concrete class gives an implementation to the abstract methods invoked by the Template Method defined in the Abstract class.

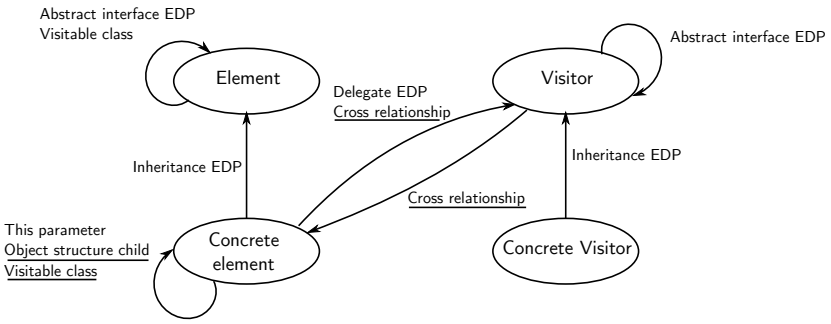

Figure 9 – Visitor refinement rule. Roles = {Visitor, Concrete Visitor, Element, Concrete Element}. The *Object structure child* clue requires the Concrete elements to belong to a well defined object structure, like a tree. With the *Visitable class* clue Concrete elements also provide methods to accept visitor classes in order to be inspected.

7 Application of the Rules to the Detected Instances

We now provide some examples of the results obtained with the refinement of the instances detected by the four considered tools. For each instance, we indicate the corresponding design pattern, the graph representing the instance after the application of the refinement rule, and the consequent considerations about the validity of the analyzed instance. Figures 10, 11, 12 and 13 report examples of the application of the refinement process on creational design pattern instances. The description of the rule application is reported on the caption of each figure.

Figure 10 – Factory Method detected instance. Roles assignment = {`DrawingView`: Creator}. The application of the Factory Method rule to this instance does not validate it, as it lacks the remaining pattern roles and the fundamental micro structures defined by the rule.

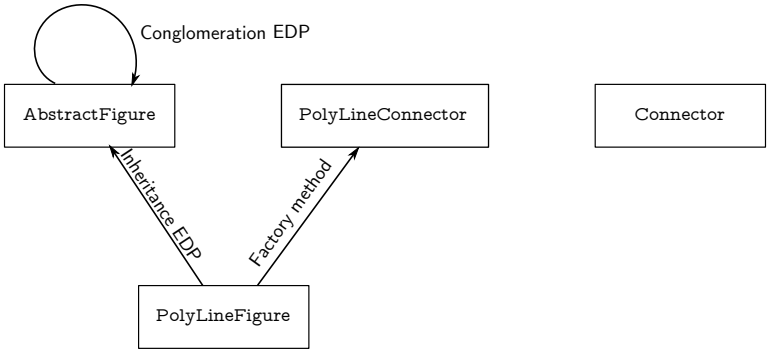

Figure 11 – Factory Method detected instance. Roles assignment = {`AbstractFigure`: Creator, `PolyLineFigure`: Concrete creator, `PolyLineConnector`: Concrete product}. This instance is validated by the rule, because the structural relationships among the roles exist and the necessary *Factory Method* clue is implemented.

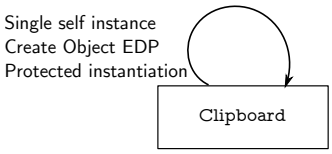

Figure 12 – Singleton detected instance. Roles assignment = {`Clipboard`: Singleton}. This instance is validated by the Singleton rule as it presents all the three required micro structures: *Single self instance*, *Create Object*, *Protected instantiation*. In fact the Clipboard class maintains a single instance of itself and prevents the creation of Clipboard instances from other classes.

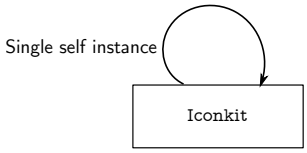

Figure 13 – Singleton detected instance. Roles assignment = {`Iconkit`: Singleton}. The instance cannot be validated by the Singleton rule because it lacks the presence of two micro structures: no protected instantiation mechanism exists.

Figures 14 and 15 describe examples of the application of the refinement process on instances of structural design patterns.

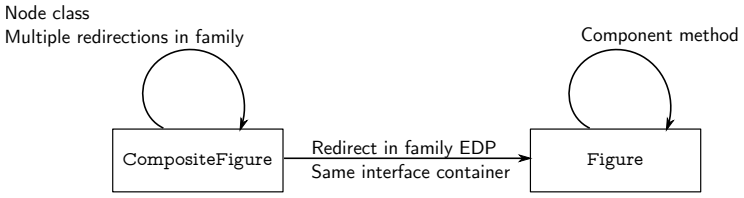

Figure 14 – Composite detected instance. Roles assignment = {`Figure`: Component, `CompositeFigure`: Composite}. The `Figure` class presents the *Component method* clue, and CompositeFigure presents both the *Multiple redirections in family* and *Same interface container* clues, so the refinement rule accepts this instance.

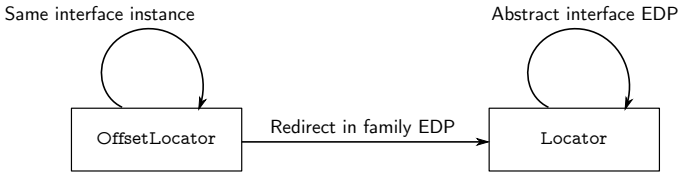

Figure 15 – Decorator detected instance. Roles assignment = {`Locator`: Component, `OffsetLocator`: Decorator}. The instance is validated by the rule, because all the constraints defined are satisfied. This is a Decorator variant without concrete classes.

Figures 16 and 17 introduce examples of the application of the refinement process on instances of behavioral design patterns.

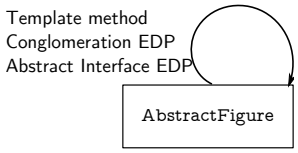

Figure 16 – Template Method detected instance. Roles assignment = {`AbstractFigure`: Abstract class}. AbstractFigure presents all the elements that characterize the Abstract Class role, in particular the *Template Method* clue. However no Concrete class is present in this instance, so the instance cannot be validated. With a further analysis we identified the `PolyLineFigure` class as a correct Concrete class, implementing the Template implementor clue; if we add the `PolyLineFigure` class the instance is correct, and is validated by the rule.

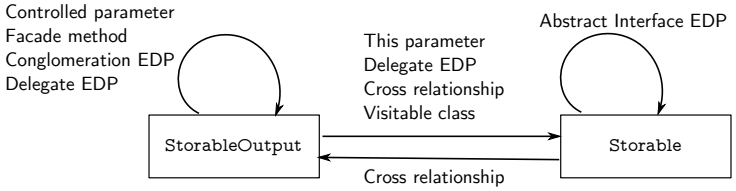

Figure 17 – Visitor detected instance. Roles assignment = {`Storable`: Visitor, `StorableOutput`: Concrete element}. This instance is not correct and is not validated by the rule. For example, the `StorableOutput` does not present the necessary *Object structure child* clue: the concrete elements of the pattern must belong to a hierarchy of objects, whose ancestor is the abstract element. As the *Object structure child* is not present, the abstract element (the root of the object structure) is not present too.

8 Refinement Results Evaluation

The results of the refinement process applied to the instances detected by the four analyzed tools are reported in Tables 2 to 5. For each of the considered patterns, the number of identified instances is reported. The number of *correct instances* column indicates how many of them are correct implementations, according to the manual evaluation process. The number of *validated instances* is then reported, *i.e.*, the number of instances that have been confirmed as correct implementations by the refinement rule. The two precision values (the first one referring to the instances detected by each single tool, the second one referring to the refined instances considering the actual correct detected instances) are then reported. If no instances for a certain pattern have been detected by the tool, the *precision before refinement* value (which considers the number of correct instances with respect to the detected instances) cannot be computed; hence "not applicable" (n/a) value is indicated. Similarly, if no instances for a certain pattern have been validated by the refinement process, the *precision after refinement* (which considers the number of correct instances with respect to the validated instances) cannot be computed, and a "not applicable" (n/a) value is reported. Table 2 describes the refinement results on the instances detected by the Design Pattern Detection Tool.

Good results have been achieved in the refinement of the Factory Method, the Singleton and the Visitor instances, where the corresponding rules succeeded in

Table 2 – Results of the refinement process on the instances detected by Design Pattern Detection Tool

Design Pattern Detection Tool						
Pattern		*Instances*			*Precision*	
Category	*Name*	*Detected*	*Correct*	*Validated*	*Before refinement*	*After refinement*
Creational	Factory Method	2	1	1	50%	100%
	Singleton	2	1	1	50%	100%
Behavioural	Command	23	11	23	48%	48%
	Template Method	5	5	5	100%	100%
	Visitor	1	0	0	0%	n/a
Structural	Adapter	23	11	23	48%	48%
	Composite	1	1	1	100%	100%
	Decorator	3	3	3	100%	100%

discarding all the detected false positives. As far as the Template Method, the Composite and the Decorator patterns are concerned, the detected instances are all correct, and the refinement succeeded in validating them. Some problems are related to the Adapter/Command instances: all of them are accepted as true positives by the refinement rule, even if only 11 of them actually are. We believe that the detection and consequent validation of instances of these patterns is difficult due to their generality. The only kind of information that characterizes them (i.e., overriding a superclass or interface method, then calling a method belonging to another class through a Delegate EDP [Smi02]) is captured by our rule and probably they are already captured by the detection techniques of the tools, so the precision remains the same. Table 3 reports the results obtained for the PINOT tool.

During the refinement of the instances reported by PINOT, the Factory Method and Decorator instances have been correctly refined, and the process succeeded in discriminating all the true positives from the false ones. Visitor and Composite instances have also been correctly discarded, as they revealed to be only false positives. Finally, Template Method and Adapter instances (which are constituted only by true positives) have all been correctly accepted by the corresponding rules.

Table 4 reports the results obtained for the FUJABA tool. The Factory Method instances have been correctly refined, and the Abstract Factory ones have all been discarded being false positives. Template Method instances have all correctly been accepted, while for the Adapter pattern we can make the same considerations as for the Design Pattern Detection tool: the pattern is too generic to be correctly refined by the rule.

Finally, Table 5 indicates the results obtained for the Web of Patterns tool.

In the case of Web of Pattern, the rule did not succeed in accepting the correct Abstract Factory instances, hence the precision rate decreased to 0%; therefore we are working on improving the Abstract Factory refinement rule. The Template Method

Table 3 – Results of the refinement process on the instances detected by PINOT

PINOT						
Pattern		*Instances*			*Precision*	
Category	*Name*	*Detected*	*Correct*	*Validated*	*Before refinement*	*After refinement*
Creational	Factory Method	34	17	17	31%	100%
	Singleton	0	0	0	n/a	n/a
Behavioural	Template Method	2	2	2	100%	100%
	Visitor	1	0	0	0%	n/a
Structural	Adapter	5	5	5	100%	100%
	Composite	4	0	0	0%	n/a
	Decorator	5	2	2	40%	100%

Table 4 – Results of the refinement process on the instances detected by FUJABA

FUJABA						
Pattern		*Instances*			*Precision*	
Category	*Name*	*Detected*	*Correct*	*Validated*	*Before refinement*	*After refinement*
Creational	Abstract Factory	2	0	0	0%	n/a
	Factory Method	2	1	1	50%	100%
	Singleton	0	0	0	n/a	n/a
Behavioural	Template Method	31	31	31	100%	100%
	Visitor	0	0	0	n/a	n/a
Structural	Adapter	26	5	26	19%	19%
	Composite	0	0	0	n/a	n/a
	Decorator	0	0	0	n/a	n/a

Table 5 – Results of the refinement process on the instances detected by Web of Patterns

Web of Patterns						
Pattern		*Instances*			*Precision*	
Category	*Name*	*Detected*	*Correct*	*Validated*	*Before refinement*	*After refinement*
Creational	Abstract Factory	14	3	0	21%	0%
	Singleton	1	1	1	100%	100%
Behavioural	Template Method	1	1	1	100%	100%
	Visitor	0	0	0	n/a	n/a
Structural	Adapter	1	0	0	0%	n/a
	Composite	1	0	0	0%	n/a

instance has been correctly accepted, and the Adapter and Composite instances correctly discarded as false positives.

9 Conclusions and Future Work

In this paper we have presented a new approach to the refinement and validation of the results provided by the experimentation of common design motif detection tools. The approach is based on the application of rules defined in terms of the roles constituting each pattern, and of the micro-structures that characterize them.

As different tools generally provide different results even while analyzing the same target systems (and the results are generally affected by a considerable number of false positives), this approach is intended to discard the identified false positives, hence improving the precision of each single tool. From our experiment, out of the considered design patterns, it emerged that the refinement rules behave well for the Factory Method, the Singleton, the Template Method, the Visitor, the Composite and the Decorator patterns. For these patterns, false positives have been correctly eliminated, and real instances have been confirmed. The Adapter pattern revealed itself to be problematic, as the hints for its detection are too much general due to the actual pattern definition and purpose. For this pattern, the false positives have not been recognized by the rule, therefore they have been accepted as real pattern instances.

The refinement approach is not intended to improve the recall of each single tool, as it is devoted uniquely to the analysis of already detected instances, and it does not allow for the detection of further pattern instances in the subject systems. In the case our refinement process fails to refine a true positive occurrence, discarding it and therefore changing it into a false negative, the recall decreases.

From the results we obtained we can conclude that the refinement process can be useful to improve the precision of the results of the detection tools. We aim to better refine the clue definitions and the corresponding rules in order to improve the refinement process. To the best of our knowledge, the only other approach that tries

to combine information coming from different sources is described by Kniesel et al [KB09]. We would therefore like to investigate a possible integration of our approach with their data fusion approach.

In the future, we plan to extend our experiments to the analysis of more systems, as well as to the analysis of repositories of design pattern instances. The refined instances can be used to enrich the design pattern repository used for benchmark proposals [AFZC10, Ess10]. Moreover, we would like to analyze formal specifications of design patterns and object oriented design [Dey10, EK03] to understand the link between the concepts we defined, the way we model them, and the available formal representations, in order to improve our pattern representations and possibly enrich existing representations with our structures. Another interesting area would be the research and definition of clues relying on dynamic analysis, and therefore able to catch particular situations, which are particulary relevant in the detection of behavioral design patterns.

A A Catalogue of Design Pattern Clues

Table 6 reports the complete catalogue of the design pattern clues. Each clue is identified by its name, its meaning, the design pattern it belongs to and the correspondent design pattern category (C for creational, B for behavioural, or S for structural design patterns), and eventually the other clues it depends on. In fact, the existence of some clues is subordinated to the presence of some others. For example, asserting that the Template implementor clue depends on the Template Method clue means that the existence of the Template Method clue is a necessary condition for the detection of the Template implementor.

Table 6 – A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
Class Information	Final class	The class is declared final.	Singleton	C	
	Interface and class inherited	The class implements an interface and extends a class, providing therefore the only mechanism to simulate multiple inheritance in the Java language.	Adapter (based on classes)	S	
	Multiple interfaces inherited	The class implements n interfaces, with $n > 1$.	Adapter (based on classes)	S	
	Object structure child	The class is a *Visitable class* and it has at least an ancestor which is either an interface or an abstract class.	Visitor	B	Cross relationship, Visitable class
	Template implementor	A class extends another class implementing a *Template Method*	Template Method	B	Template Method

Table 6 – A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
Multiple Classes Information	Façade method	The body of a method consists uniquely of method calls to classes which are not related with it, *i.e.* which are not a superclass, an implemented interface or the class itself. A facade method could also contain some object creations, but no other statements besides object creations or method calls.	Façade	S	
	Proxy class	A class implements an interface or extends an (abstract) class, and owns a reference to a class that implements the same interface or extends the same (abstract) class.	Proxy	S	
Variables Information	Private flag	The class maintains a control flag that is declared private. A flag belongs to a simple type, typically boolean; numerical fields are considered flags when their value is compared to form a boolean expression in a control statement.	Singleton	C	
	Static flag	The class maintains a control flag that is declared static. A flag belongs to a simple type, typically boolean; numerical fields are considered flags when their value is compared to form a boolean expression in a control statement.	Singleton	C	
Instance Information	Controlled self instantiation	The instantiation of an object of the same class occurs inside an if (or a switch) block, therefore under a condition.	Singleton	C	
	Private self instance	The class owns a private instance of the same class. Access to this instance can occur only from within the same class.	Singleton	C	
	Static self instance	The class has a static instance of the same class. Therefore this instance is unique inside the system.	Singleton	C	
	Single self instance	The class maintains a unique instance of the same class, no matter if it is static or not.	Singleton	C	
	Instance in abstract class	An abstract class maintains a reference to a different class.	Bridge	S	

Table 6 – A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
	Same interface container	A class contains some kind of collection of objects that are compatible with an ancestor of the declaring class.	Composite, Interpreter	S	
	Same interface instance	A class contains a reference to an object whose type is compatible with an ancestor of the declaring class.	Decorator	S	
Method Signature Information	Controlled parameter	A method of a certain class receives as input a parameter used inside it to make some controls (*i.e.* the parameter is used in the condition of some `if` or `switch` block). If a method controls more than one of its input parameters, each one of these parameters will be an instance of this clue.	Abstract Factory, Builder, Factory Method	C	
	Factory parameter	A method of a certain class receives as an input parameter an object that belongs to a class defining some *Concrete product getter* methods.	Abstract Factory, Builder, Factory Method	C	Concrete product getter
	Protected instantiation	All the constructors within a given class are declared private.	Singleton	C	
	This parameter	A method receives the caller object as a parameter.	Observer, Visitor	B	
	Adapter method	Two types of Adapter method exist. It can be a method which is an implementation of an interface method and that calls a method belonging to the parent class; or it can be an overridden method from the parent class which calls a method belonging to a class that does not share common ancestors with the adapter method declaring class.	Adapter	S	Interface method (only in the first case)
	Interface method	A class implements a method declared inside an interface.	Adapter (based on classes)	S	
	Overriding method	A class overrides a method belonging to its superclass.	Adapter (based on objects)	S	
	Component method	A class declares a method that takes an object of the same class as its single parameter.	Composite	S	

Table 6 – A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
	Cross relationship	Given two classes $C1$ and $C2$, $C1$ declares a method which accepts a reference to $C2$ as one of its parameters, viceversa $C2$ declares a method which accepts a reference to $C1$ as one of its parameters.	Visitor	B	
	Abstract cyclic call	A method invokes an abstract method within a cycle.	Iterator, Observer	B	
	Factory Method	A method contains a class instance creation statement and overrides a method belonging to the superclass or to one of the superinterfaces of the subject class.	Factory Method	C	
Method Body Information	Instance in abstract referred	A method of a class implementing *Instance in abstract class* invokes a method on the declared instance.	Bridge	S	Instance in abstract class
	Multiple redirections in family	A method contains a *Redirect in Family* [Smi02] method invocation that is contained within a cycle.	Composite	S	
	Proxy method invoked	A proxy class invokes a method on the referred subject using a Redirect in limited family [Smi02] method call EDP.	Proxy	S	Proxy class
	Visitable class	A method has a *Cross relationship* clue and passes the owner object (`this`) to the target method of the *Cross relationship*.	Visitor	B	This parameter, Cross relationship
	Template Method	A method calls at least an abstract method within its body.	Template Method	B	
Method Set Information	All methods invoked	A class invokes all of the public methods declared in a target class.	Adapter	S	
	Leaf class	A class extends another class without implementing or redefining the methods that are concerned with the handling of classes that are compatible with the same interface (therefore tagged with a *Component method* clue), or giving an empty implementation for such methods.	Composite	S	Component method
	Node class	A class extends another class implementing or redefining the methods that are concerned with the handling of classes that are compatible with the same interface (therefore tagged with a *Component method* clue).	Composite	S	Component method

Table 6 – A catalogue of design pattern clues

Category	Clue name	Meaning	Belongs to	DP Category	Depends on
Return Information	Concrete product getter	A class declares one or more methods that return objects of a type different from itself.	Abstract Factory, Prototype	C	
	Concrete products returned	A method returns objects that belong to subclasses of the declared return type.	Abstract Factory, Factory Method, Builder	C	
	Empty concrete product getter	A class declares one or more methods that return objects belonging to some other classes, but the implementation of these methods is empty, *i.e.* it consists only of a default return statement (as, for example, `return null`).	Builder	C	
	Empty method	A class declares one or more methods that return simple types, but their implementation is empty, *i.e.* it is only formed by a default return statement (for example, `return false` for the `boolean` data type).	Builder	C	
	Multiple returns	A method provides several possible return points.	Abstract Factory, Factory Method, Builder	C	
	Void return	A class defines a method that instantiates an object without returning it.	Builder	C	
	Cross hierarchy return	A method returns an object of a class belonging to a different hierarchy.	Iterator	B	
Java Information	Clone returned	A method returns a clone of a certain instance.	Prototype	C	
	Cloneable implemented	A class implements the `java.lang.Cloneable` interface.	Prototype	C	
	Prototyping constructor	A method defines a constructor which receives objects that can be cloned, as instances of classes implementing the `java.lang.Cloneable` interface.	Prototype	C	Cloneable implemented
	Controlled exception	A method of a class can throw an exception inside a control block.	Singleton	C	

References

[AFC98] Giulio Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98)*, pages 153–160, June 1998. doi:10.1109/WPC.1998.693342.

[AFMR11] Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Design patterns: a survey on their micro-structures. *To appear in Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[AFMRT05] Francesca Arcelli Fontana, Stefano Masiero, Claudia Raibulet, and Francesco Tisato. A comparison of reverse engineering tools based on design pattern decomposition. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 262–269, Washington, DC, USA, 2005. IEEE Computer Society. doi:10.1109/ASWEC.2005.5.

[AFRG+06] Francesca Arcelli Fontana, Claudia Raibulet, Yann-Gaël Guéhéneuc, Giulio Antoniol, and Jason McC Smith. Design pattern detection for reverse engineering. In *Proc. 13th Working Conf. Reverse Engineering WCRE '06*, 2006. doi:10.1109/WCRE.2006.23.

[AFZ11] Francesca Arcelli Fontana and Marco Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, April 2011. doi:10.1016/j.ins.2010.12.002.

[AFZC10] Francesca Arcelli Fontana, Marco Zanoni, and Andrea Caracciolo. A benchmark platform for design pattern detection. In *Proceedings of The Second International Conferences on Pervasive Patterns and Applications PATTERNS 2010*, page 42 to 47, Lisbon, Portugal, November 2010. IARIA, Think Mind. Available from: http://www.thinkmind.org/index.php?view=article&articleid=patterns_2010_2_30_70046.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: a system of patterns*, volume 1. John Wiley and Sons, 1996.

[Bra] John Brant. Jhotdraw. Web site. http://www.jhotdraw.org/.

[CCI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, January 1990. doi:10.1109/52.43044.

[Coo98] James W. Cooper. *The Design Patterns Java Companion*. Addison-Wesley, 1998. Available from: http://www.patterndepot.com/put/8/DesignJava.PDF.

[DE07] Jens Dietrich and Chris Elgar. Towards a web of patterns. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):108–116, 2007. Software Engineering and the Semantic Web. doi:10.1016/j.websem.2006.11.007.

[Dey10] Shouvik Dey. Formal specification of structural and behavioral aspects of design patterns. *Journal of Object Technology*, 9(6):99–126, November 2010. doi:10.5381/jot.2010.9.6.a5.

[DP09] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July–August 2009. doi:10.1109/TSE.2009.19.

[EK03] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 149–159, Washington, DC, USA, 2003. IEEE Computer Society. Available from: http://portal.acm.org/citation.cfm?id=776816.776835.

[Ess09] Essere lab. Micro structures detector. Web Site, 2009. http://essere.disco.unimib.it/reverse/files/Micro_structures_detector.pdf.

[Ess10] Essere lab. Dp-benchmark. Web Site, 2010. http://essere.disco.unimib.it:8080/DPBWeb/.

[FBTG02] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus — reverse engineering tool and schema for C++. In *Proceedings. International Conference on Software Maintenance*, pages 172–181. IEEE Computer Society, 2002. doi:10.1109/ICSM.2002.1167764.

[FHFG08] Lajos Jeno Fülöp, Péter Hegedus, Rudolf Ferenc, and Tibor Gyimóthy. Towards a benchmark for evaluating reverse engineering tools. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 335–336, October 2008. doi:10.1109/WCRE.2008.18.

[GA08] Yann-Gaël Guéhéneuc and Giulio Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008. doi:10.1109/TSE.2008.48.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GM05] Joseph (Yossi) Gil and Itay Maman. Micro patterns in java code. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 97–116, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094819.

[Gué07] Yann-Gaël Guéhéneuc. PMARt: pattern-like micro architecture repository. In Michael Weiss, Aliaksandr Birukou, and Paolo Giorgini, editors, *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, July 2007.

[KB96] Jung Jae Kim and Kevin Michael Benner. *Pattern Languages of Program Design 2*, chapter Implementation of patterns for the observer pattern, pages 75–86. Addison-Wesley, Reading, MA, 1996.

[KB09] Günter Kniesel and Alexander Binun. Standing on the shoulders of giants — a data fusion approach to design pattern detection. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 208–217, May 2009. doi:10.1109/ICPC.2009.5090044.

[KBH+10] Günter Kniesel, Alexander Binun, Péter Hegedüs, Lajos Jeno Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhenéuc, and Nikolaos Tsantalis. DPDX — towards a common result exchange format for design pattern detection tools. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 232–235, March 2010. doi:10.1109/CSMR.2010.40.

[KGH10] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Identification of design motifs with pattern matching algorithms. *Information and Software Technology*, 52(2):152–168, 2010. cited By (since 1996) 1. doi:10.1016/j.infsof.2009.08.006.

[KP96] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 208–215, November 1996. doi:10.1109/WCRE.1996.558905.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 226–235, New York, NY, USA, 1999. ACM. doi:10.1145/302405.302622.

[Mag06] Stefano Maggioni. Design patterns clues for creational design patterns. In *Proceedings of the First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE 2006), co-located event with WCRE 2006*, October 2006.

[MG07] Naouel Moha and Yann-Gaël Guéhéneuc. PTIDEJ and DECOR: identification of design patterns and design defects. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 868–869, New York, NY, USA, 2007. ACM. doi:10.1145/1297846.1297930.

[MJS+00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM. doi:10.1145/336512.336526.

[NNZ00] Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 742–745, New York, NY, USA, 2000. ACM. doi:10.1145/337180.337620.

[NSW+02] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 338–348, New York, NY, USA, 2002. ACM. doi:10.1145/581339.581382.

[PKG+00] Jukka Paakki, Anssi Karhinen, Juha Gustafsson, Lilli Nenonen, and A. Inkeri Verkamo. Software metrics by architectural pattern mining. In *Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress)*, pages 325–332, Beijing, China, 2000. Available from: http://www.cs.helsinki.fi/group/maisa/ifip2000.pdf.

[Smi02] Jason McC Smith. An elemental design pattern catalog. Technical Report 02-040, Dept. of Computer Science, Univ. of North Carolina - Chapel Hill, December 2002. Available from: ftp://ftp.cs.unc.edu/pub/publications/techreports/02-040.pdf.

[SO06] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society. doi:10.1109/ASE.2006.57.

[SS03] Jason McC Smith and David Stotts. Spqr: flexible automated design pattern extraction from source code. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 215–224, October 2003. doi:10.1109/ASE.2003.1240309.

[SvG98] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '98/FSE-6, pages 10–16, New York, NY, USA, 1998. ACM. doi:10.1145/288195.288207.

[Tai07] Toufik Taibi. *Design Pattern Formalization Techniques*. IGI Publishing, Hershey, PA, USA, 2007.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006. doi:10.1109/TSE.2006.112.

[Vok06] Marek Vokác. An efficient tool for recovering design patterns from C++ code. *Journal of Object Technology*, 5(1):139–157, January 2006. doi:10.5381/jot.2006.5.1.a6.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Technology of Object-Oriented Languages, 1998. TOOLS 26. Proceedings*, pages 112–124, August 1998. doi:10.1109/TOOLS.1998.711007.

About the authors

Francesca Arcelli Fontana is an associate professor of computer science at the Department of Computer Science of the University of Milano Bicocca, where she works on software evolution and reverse engineering. Contact her at arcelli@disco.unimib.it, or visit http://essere.disco.unimib.it.

Marco Zanoni is a PhD Student at the Department of Computer Science of the University of Milano Bicocca; he took his master degree in 2008. He is currently working on techniques for design pattern detection on object oriented systems. Contact him at marco.zanoni@disco.unimib.it, or visit http://essere.disco.unimib.it.

Stefano Maggioni took his master degree in 2006 and his PhD degree in 2010 at the University of Milano Bicocca, working on the definition of design pattern clues. Contact him at maggioni@disco.unimib.it, or visit http://essere.disco.unimib.it.

Acknowledgments We would like to kindly thank the reviewers for their very useful comments and suggestions for improving the paper.