Università degli Studi di Milano-Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Dottorato di Ricerca in Informatica – XXII Ciclo

Design Pattern Detection and Software Architecture Reconstruction: an Integrated Approach based on Software Micro-structures

Ph.D. Candidate: Stefano Maggioni

Thesis advisor: Prof. Francesca Arcelli Fontana
Thesis tutor: Prof. Francesco Tisato

Academic Year 2008-2009

Table of Contents

Chapter 1

Introduction

In this chapter we give an overview of the research field related to our activities. We provide common definitions of Design Pattern Detection and Software Architecture Reconstruction, and we introduce the concept of software micro-structure. After an overview on common micro-structures, we finally introduce the thesis objectives and aims, which are concerned with the exploitation of micro-structures for Design Pattern Detection and Software Architecture Reconstruction activities.

1.1. The research field

A software engineering area that has assumed more and more importance during the last years in the field of software maintenance is *reverse engineering*. In [Chi90], Chikofsky defines reverse engineering as "the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction". A principal aim of reverse engineering is to allow the reconstruction of the structure of target software systems and to detect their fundamental components and modules, in order to consequently obtain their forming structures. The retrieval of this kind of information would make the restructuring and maintenance phases easier, as the system would not be seen as a single monolithic structure, but as a set of small-sized interacting components that are usually easier to manage with respect to the overall system.
In the context of reverse engineering, two activities are of particular interest. These are namely *Design Pattern Detection* (which, from now on, can also be indicated with the *DPD* acronym) and *Software Architecture Reconstruction* (*SAR*).

1.1.1. Design pattern detection (DPD)

Design patterns (DPs) have been introduced by Gamma, Helm, Johnson and Vlissides (collectively known as the *Gang of Four*) in [GHJV94]. A design pattern is a description of a commonly occurring software design problem, together with a description of a possible solution to that problem. The proposed solution is applicable whenever the problem is faced, independently from the particular system to be designed or to the precise context in which the system is being developed. A design pattern gives therefore general indications about how to solve a well-defined issue, without deepening into the implementation details about how the problem is actually solved. Each pattern is presented according to the following structure:

- *Pattern name and classification*: a descriptive and unique name that helps in identifying and referring to the pattern;
- *Intent*: a description of the goal behind the pattern and the reason for using it;
- *Also known as*: other possible names for the pattern;
- *Motivation (forces)*: a scenario consisting of a problem and a context in which this pattern can be used;
- *Applicability*: the possible situations and cases in which this pattern is usable;
- *Structure*: a graphical representation of the pattern, usually through an UML class diagram;
- *Participants*: a listing of the classes and objects used in the pattern and their roles in the design; in general, each class plays a well defined role inside the pattern, according to the pattern description;
- *Collaboration*: a description of how classes and objects used in the pattern interact with each other;
- *Consequences*: a description of the results, side effects, and tradeoffs caused by using the pattern;
- *Implementation*: a description of an implementation of the pattern, that is the solution part of the pattern;
- *Sample code*: an illustration of how the pattern can be practically used and implemented;
- *Known uses*: examples of real usages of the pattern;
- *Related patterns*: other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

Twenty three design patterns have been defined, subdivided in three categories. *Creational design patterns* deal with object creation mechanisms, trying to create objects in a manner suitable to the context in which the patterns should be applied. *Structural design patterns* describe how classes and objects can be combined to form larger structures. Each structural pattern is then further specified as being based on *classes* or based on *objects*. Structural patterns based on classes describe how inheritance can be used to provide more useful program interfaces. Object patterns describe how objects can be composed into larger structures using object composition, or through the inclusion of objects within other objects. Finally, *behavioural design patterns* identify common communication ways among objects and concentrate on the assignment of responsibilities among them.

Design patterns are useful both during a system's design phases both in forward engineering (as they are well known and optimal solutions to given design issues and can be seen as directives to follow in order to solve a problem in a given context), and in reverse engineering activities (as the identification of design patterns inside a software system can give hints about the issues faced during its design). In the context of reverse engineering, they can also be considered as indicators of good system design quality, as their presence grants the use of structures that are, for their self definition, reusable. Therefore, the activity of DPD, aimed at identifying design patterns inside a subject software system, can give useful information about the design of a system, indicating the logical fundaments of a certain implementation. Moreover, DPD 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. The activity of design pattern detection may also reveal useful for the specification and development of a design advisor. Analyzing a subject system may provide information about the existence of components or modules whose implementation is close to that of some design patterns, hence suggesting for their refactoring in order to comply with the patterns specifications. Moreover, the analysis may reveal parts of the system representing design issues that have not been properly solved, at least not in the optimal way. In this case, the advisor could suggest for the implementation of a design pattern that is adequate to the detected issue, in order to obtain an elegant and effective solution to it.

The detection of design patterns is supported by ad-hoc software tools. The main steps a tool performs in a design pattern detection process are related to information extraction from the target system, archetypes recognition and presentation of results, as reported in Figure 1.1. The information extracted from the analyzed systems depends on the elements searched by the detection algorithm used to identify the design patterns. Usually, the extracted information is represented in a language-independent form, such as abstract syntax trees (ASTs) or abstract syntax graphs (ASGs). Besides the extracted information, the detection algorithm usually receives in input a catalogue of design patterns, in which patterns are described according to the meta-representation used by the detection algorithm. This is often based on matching techniques which try to map representations of design patterns stored in a catalogue to the representation of the information extracted from the analyzed system. Other techniques can exploit rule-based systems, or are related to the identification of patterns sub-components, or else are based on the computation of characteristic metrics. The various kinds of approaches will be discussed in Chapter 2.

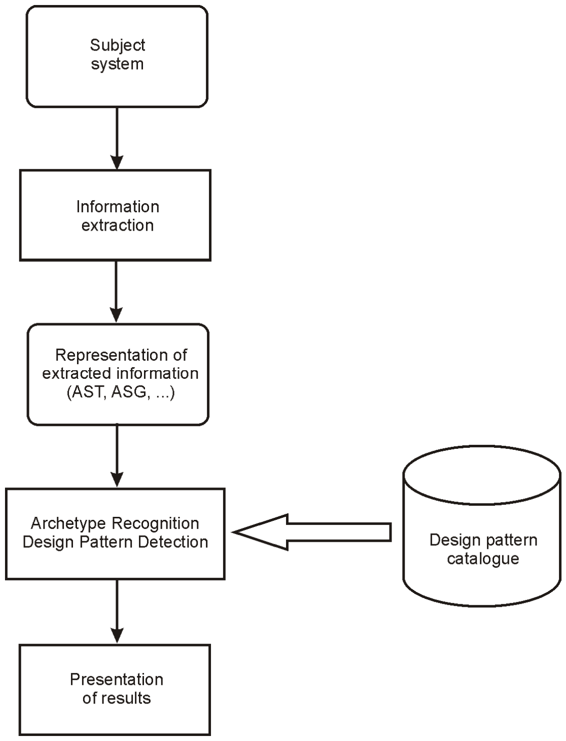

Subject system
Information extraction
Representation of extracted information (AST, ASG, ...)
Archetype Recognition Design Pattern Detection
Design pattern catalogue
Presentation of results

Figure 1.1 *– The main steps of a design pattern detection process*

However, the use of design patterns inside software systems introduces a problem that is troublesome for their detection. In fact, even if the specifications of each pattern are completely generic so that they can be applied in different ways and in different contexts, the actual implementations of each pattern can inevitably differ from one another, due to their intrinsic generality. This problem is known as the *variants problem*, and it is one of the biggest obstacles in a design pattern detection process, as design patterns are to be identified starting from actual implementations, that can be quite different from the proposed general pattern structure. In this thesis we try to face this problem by introducing and adopting *software micro-structures* (that will be briefly presented in Section 1.2) as possible means to distinguish among various realizations of design patterns and to improve the precision of design pattern detection tools.

1.1.2. Software architecture reconstruction (SAR)

Many different definitions of software architecture (SA) actually exist. The Software Engineering Institute (SEI) at Carnegie Mellon University collected something like two hundred and twelve software architecture definitions by software engineering experts [Sei]. Many definitions are also proposed in the literature and in common standards. Here we report some of the most representative ones.

The ANSI/IEEE Standard 1471-2000 (Recommended Practice for Architectural Description of Software-Intensive Systems) [IEEE] states that "architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution".

According to the Rational Unified Process, "an architecture is the set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these structural and behavioral elements into progressively larger subsystems, and the architectural style that guides this organization (these elements and their interfaces, their collaborations, and their composition)" [Kru99].

Bass, Clements and Kazman [BCK03] propose the following clear definition for software architecture: "the software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them".

Out of the considered definition, we think that this latter one describes at best and immediately what software architecture actually is. Bass, Clements and Kazman also point out that software architecture shall also comprise its documentation [BCK03]. Documentation is indeed a crucial issue, because it shall be the most effective mean to understand a complex system and its structures. The lack or poor quality of documentation makes the understanding of a system more difficult and challenging. Nonetheless, one of the core problems with software system is actually the lack of documentation. Having tools to generate documentation (that, thanks to Bass' definition, can be considered in the set of tools for the reconstruction of software architecture) leads to the obtainment of an important source of information for the understanding of the subject systems. For all these reasons, while speaking about software architecture throughout this thesis, we refer to the definition suggested by Bass, Clements and Kazman.

Besides software architecture, numerous definitions for software architecture reconstruction (SAR) have also been proposed.

Van Deursen defines architecture reconstruction as "a reverse engineering activity that aims at recovering those decisions that either have been lost (because have not been documented or the original developers have left) or are unknown (because they originate from the system's evolution)" [DHK+04]. In [DR04] he also states that "software architecture reconstruction is the process of obtaining a documented architecture for an existing system. Although such a reconstruction can make use of any possible resource (such as available documentation, stakeholder interviews, domain knowledge), the most reliable source of information is the system itself, either via its source code or observations on its execution".

Seacord and Plackosh [SP03] state that "architecture reconstruction provides analysis at the highest level of abstraction. In this process, the as-built architecture of an implemented system is obtained from the existing system by analyzing it, using tools to extract information and to build system models at various levels of abstraction. This process produces a representation of the system architecture and generates views of this architecture. Architecture reconstruction is a complex task requiring a variety of activities and skills. Although tool support is usually a requirement for architectural reconstruction, no single tool or set of tools supports all reconstruction activities".

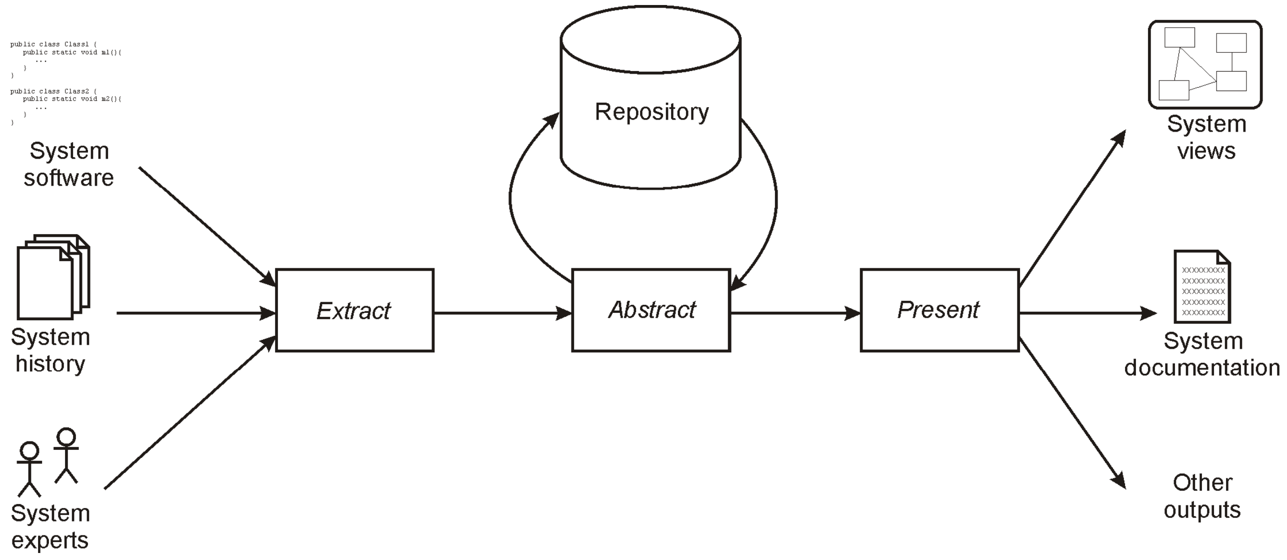
A clear definition of software architecture reconstruction is provided by O'Brien *et al* [BSV02]: "architecture reconstruction is the process by which the architecture of an implemented system is obtained from the existing system. The approaches to architecture reconstruction are aimed to evaluating the conformance of the as-built architecture to the as-documented architecture, reconstructing architecture descriptions for systems that are poorly documented or for which documentation is not available, and analyzing and understanding the architecture of existing systems to enable modification of the architecture to satisfy new requirements and to eliminate existing software deficiencies".

Out of the presented definitions, the definition provided by O'Brien is the one that best fits with our activities. In particular we are interested in the last part of the definition. In fact, we think that one of the principal aims of architecture reconstruction is providing the engineers with sufficient understanding helping them to effectively intervene in the maintenance and evolution of the subject system. Thus, while referring to SAR along the thesis, we make an implicit reference to the just presented definition.

Figure 1.2 represents the steps pursued by a software architecture reconstruction tool during the reconstruction process, known as the *extract-abstract-present* model [TPS96].

First of all, information is extracted from different possible sources. The main source of information is obviously the subject system itself and its documentation. Important sources are also the history of the analyzed system, and the expertise of the system developers and managers. The extracted information can be represented in many different ways; each reconstruction tool adopts its own conventions and representation mechanisms.

These representations cannot be directly exploited by the engineers, but they need to be interpreted and abstracted, in order to be modified and translated in usable artifacts.


```
public class Class1 {
   public static void m1(){
      ...
   }
}

public class Class2 {
   public static void m2(){
      ...
   }
}
```

Figure 1.2 *– A three-step process for software architecture reconstruction*

Therefore, the abstraction process usually generates views, documentation or other kinds of media, in order to obtain an abstracted overview of the analyzed system: the aim is to provide the engineer means to have a global understanding of the system without minding at the implementation details and at the low-level issues. The generated artifacts are finally presented to the user, who can interpret and exploit them in order to deal with the reconstructed software architecture.

1.2. A brief overview on software micro-structures for DPD and SAR

With the term *software micro-structure* (*micro-structure* for brevity) we indicate any code element that can be automatically and univocally detected from the source code of a software system, and which represents useful basic hints for the understanding of the structures composing a system. According to our research interests, these elements can be exploited in two different ways:

- For design pattern detection purposes, as possible hints for the presence of design patterns inside a system;
- For software architecture reconstruction activities, as they directly or indirectly codify information about structural relationships among classes.

Different categories of micro-structures have been defined in the literature. In this dissertation we consider only the three kinds of micro structures that have been exploited for our activities, namely *elemental design patterns* (*EDPs*), *design pattern clues* (*DP clues*) and *micro patterns*, that will be now briefly introduced. EDPs, DP clues and micro patterns will be considered with more detail in Chapter 3, where we also provide definitions for each single micro-structure.

1.2.1. Elemental design patterns

Smith and Stotts proposed and introduced elemental design patterns in [SS02]. EDPs are means to provide solutions to very common programming problems, which are faced in the everyday programming practice. EDPs address problems of very limited dimensions, generally involving at most three classes. Sixteen EDPs have been defined so far, and they face the creation and referencing of objects, the various possible forms of method invocation, and the inheritance relationships between two classes or interfaces.

1.2.2. Design pattern clues

In [Mag06a] we have discussed about the usefulness of EDPs for design pattern detection. As we think that EDPs don't behave well as far as the detection of pattern roles is concerned, we have defined design pattern clues, as possible hints for the presence of design patterns inside a software system. During our research activities, we have analyzed the structures and the roles of the design patterns, focusing also on their possible Java implementations, in order to deduce which particular code structures and realizations could be peculiar for each pattern. From these studies, a first set of clues for the creational design patterns category emerged [Mag06a, Mag06b]. The set of clues has been further modified and enriched during the Ph.D. course, covering all the pattern categories. The current set of design pattern clues is constituted by 41 elements, collected in eight categories that focus on the various constructs related to a class or interface, like its definition, its attributes, its method signatures and bodies and so on.

1.2.3. Micro patterns

Micro patterns were introduced by Gil and Maman [GM05] in order to capture very common programming techniques. Currently, there are 27 micro patterns subdivided into eight categories mainly considering the state of a class or interface (represented by its attributes) and its behavior (represented by its set of methods).

1.2.4. Usefulness of micro-structures for DPD and SAR

The usefulness of micro-structures for DPD and SAR activities is mainly concerned with the subdivision of such complex problems in simpler tasks, and to make the basic or particular relationships and constraints among classes explicit. The first identification of these bricks is to be considered as a starting point for an incremental process devoted to the identification of more complex structures, like the design patterns or the structural relationships among a large number of system classes and modules. Moreover, the micro-structures are univocally and non-ambiguously detectable from a subject system, with respect for example to design patterns, whose direct identification is made impossible due to their intrinsic generality and the possible infinite variants they could realize each of them.

1.3. Thesis objectives and aims

Four main objectives characterize this dissertation. The first main activity that will be described is concerned with a comparison of the considered micro-structures on the basis of several core aspects. The comparison allowed us to deeply understand the usefulness of these elements both for DPD and SAR activities. Next, we will propose a novel approach to the refinement and validation of the results provided by different design pattern detection tools. The approach is based on the definition and application of micro-structure-based refinement rules that demonstrated to be useful for the improvement of the precision of the results provided by the considered tools. Finally, we will consider the exploitation of micro-structures for SAR activities, in particular as far as the generation of static views, the calculation of software metrics, and the detection of object-oriented and structural antipatterns are concerned. In this context, we will also provide a new interpretation of micro patterns based on similarity scores. The new interpretation allows for the detection of classes that are very close realizations of micro patterns, which cannot be detected by precise matching approaches, but which indeed can give useful indications about possible antipatterns, about the stability of the analyzed systems, as well as about the evolution of micro patterns along different software releases.
Two main motivations justify these objectives. First of all, software micro-structures have generally been considered "as they are" in the literature. The inspection of their usefulness for DPD and SAR purposes has never really been deeply inspected. Moreover, having a common base (i.e. the considered micro-structures) for both DPD and SAR activities allows the design and implementation of a single integrated tool supporting both

disciplines, with a common starting point represented by the micro-structures detected from the analyzed systems.

1.4. Thesis outline

The thesis is structured according to the following chapters.
In Chapter 2 (Related works) we will present an overview to the major approaches to design pattern detection. We will differentiate among static, dynamic and combined approaches, and we will also introduce and describe some of the available and analyzed design pattern detection tools. In this chapter we will also present the main software architecture reconstruction approaches and tools, introducing some of the available and tested tools, and proposing a comparative evaluation of them.
Chapter 3 (Software micro-structures) is focused on the description and definition of the considered micro-structures (EDPs, DP clues and micro patterns). As the micro-structures have been in general informally introduced, and as we verified that some of them could be ambiguously interpreted, in this chapter we suggest a more formal definition of micro-structures aimed at solving these ambiguities and at having a unique common structured repository of these elements. The redefinition started from the introduction of a new set of fundamental elements (that we name *code atoms*), which each single micro-structure can be defined from, dealing to the obtainment of a unified micro-structures catalogue.
Chapter 4 (Micro-structures for design pattern detection) is devoted to the analysis of the usefulness of micro-structures for design pattern detection purposes. The micro-structures have been compared according to six peculiar aspects, namely objectives, detail level, definition technique, detection technique, categorization, and interdependence among elements. We will further analyze and detect micro-structures in common design pattern instances. The study of the obtained results will help us in the definition of sets of micro-structures that can be seen as necessary and/or useful elements for the detection of the considered patterns.
Chapter 5 (Micro-structures for the validation and refinement of design pattern detection tools results) describes the process of validation and refinement of the results provided by third-pary design pattern detection tools. We will present the results provided by four tools on the analysis of a well-known and established Java framework. We will compare the obtained results, showing the strong differences among them, and pointing out the problem related to the identification of many false positives, due to the different algorithms and detection strategies adopted by the tools and to the difficult pattern variants problem. Next, we will define refinement rules based on micro-structures for the patterns detected by the various tools. We will describe the refinement process, and verify

the detected instances according to the defined rules. We will demonstrate how the refinement process succeeds in the elimination of a good percentage of false positives, hence improving the precision of each tool.
Chapter 6 (Micro-structures for software architecture reconstruction) analyzes and compares elemental design patterns and micro patterns as means to support the identification of architectural information about a software system. We will show how the EDPs revealed useful to highlight the structural constraints and relationships among the entities of a system, to calculate common software metrics, and to identify structural antipatterns like breakable, butterfly and hub classes. On the other side, micro patterns revealed useful for the detection of interesting peculiar classes and interfaces, in particular as far as critical classes or object-oriented antipatterns are concerned.
In Chapter 7 (A novel interpretation of micro patterns) we will consider micro patterns under a different point of view according to two common object-oriented metrics, namely the number of attributes (NOA) and of methods (NOM) of a certain class or interface. The new interpretation allows for the detection of classes whose implementation is similar to a correct micro pattern implementation. We will focus on micro patterns devising critical classes and we will analyze different Java systems, both through the original precise matching approach and through the new interpretation. We will compare the obtained results, and underline how the new interpretation allows for the detection of micro pattern instances that are very close to a correct micro pattern implementation and hence should be considered as possible candidates for refactoring or restructuring.
Finally, Chapter 8 will resume our work, discusses the conclusions and the obtained results, and traces possible future scenarios.

Chapter 2

Related works

Abstract

In this chapter we present some of the main works related to the activities of design pattern detection and software architecture reconstruction, giving also an overview of the major types of approaches.

2.1. Related works on design pattern detection

Many different approaches and tools for design pattern detection (DPD) have been presented in the literature. The approaches differ in the kind of analysis pursued on the subject systems, in the algorithms adopted for pattern detection, in the set of patterns they are able to recognize, and in the analysis results, which may differ from one tool to another even while considering the same subject system. Section 2.1.1 considers and discusses different possible approaches and methodologies for DPD, while in Section 2.1.2 we introduce some of the major approaches and tools reported in the literature, giving an organic overview of them.

2.1.1. Categorizations of approaches and methodologies for design pattern detection

The detection of design patterns in software systems requires the extraction of meaningful information from these systems and the recognition of patterns starting from this information. Various classifications for design pattern detection solutions and approaches have been presented in the literature, considering various points of view. One major classification consists in categorizing design pattern detection approaches as *static*, *dynamic*

or *hybrid*. Usually, the input information required by static extraction and analysis approaches is the source code of the subject system itself (as for example in [NNZ00, SS03]). Dynamic approaches may require either the source code or the execution traces of the analyzed system [SSys02]. Hybrid or combined approaches, like those proposed in [AB05, Wen03], usually extract dynamic information from the system, to be further analyzed though static investigations.
Design pattern detection heavily exploits language dependent mechanisms and constructs. At now, each of the available tools for design pattern detection is suited for the analysis of a particular programming language: for example FUJABA [NNZ00] and PTIDEJ [Gue05] have been developed for the analysis of Java systems, Columbus/MAISA [FGMP02] for C++, KT [Bro97] for Smalltalk and DPVK [WT05] for Eiffel. An attempt to define a language independent approach to detect object-oriented best practices and design patterns starting from Smalltalk or Java systems is described in [FM04]. Another example is provided by the SPQR approach [SS03], which defines a language independent way to represent the extracted information from the source code and a language independent algorithm to detect design patterns. However, the information extraction mechanisms in SPQR are specific to C++ software systems.
In [AMRT05] we proposed the categorization of the available approaches based on the information used in the detection process. This work resulted in the definition of three main categories of pattern detection tools:

- The classical solutions considering the entire representation of design patterns (like for example PTIDEJ [Gue05] or CrocoPat [BL03]). In these cases, the detection algorithm tries to map at once the entire pattern on the representation of the source code; usually, this kind of approaches claims for a complete catalogue containing all possible implementations of design patterns;
- Solutions considering a minimal set of key structures the design patterns consist of (e.g. JAdept [APRR09], SPOOL [KSRP99]). The detection algorithm tries to individuate (at once) the core set of structures a pattern is built on; this approach claims for a further analysis of design patterns leading to the identification of their core elements;
- Solutions considering the sub-components the design patterns are built of (e.g. FUJABA [NNZ00], MARPLE [Arc06], SPQR [SS03]). The detection algorithm works incrementally by first individuating the pattern sub-components, then trying to combine these sub-components into patterns. This last category of approaches is particularly interesting and challenging, because it implies a further formalization of design patterns which preserves their flexibility and improves their definition and understating.

We now present the more important approaches to design pattern detection, giving a brief description of their working principles and adopted detection strategies.

2.1.2. Approaches and tools for design pattern detection

Since their introduction in 1994, design pattern became well established practices in forward engineering, as we have outlined in the introduction. Interest for design patterns soon captured the reverse engineering community. The detection of such structures allows the improvement of software systems understanding, the assessment of software quality, the identification of relevant structural and reusable information, and the simplification of systems re-documentation as well. Along the years, many research groups devoted their activities for the proposal of approaches and the development of tools for the detection of design patterns. Tools and approaches may consistently vary from one another, in terms of the adopted detection strategies and algorithms, the detectable patterns, and the quality of the detection results in terms of precision and recall [BR99] of the identified pattern instances.

Pree [Pre94, Pre97] describes meta patterns as minimal means to capture reusable object-oriented design. He introduces seven meta patterns that identify seven possible class/object composition by means of template and hook methods and of the relationships between the caller and the called objects. These meta patterns are then related to sample frameworks in order to illustrate how implemented classes (and their method calls) satisfy the defined meta patterns.

Information about the architecture of classes composing structural patterns is the fundament of the Pat system proposed by Kramer and Prechelt [KP96]. Structural patterns are represented as Prolog rules, while the source code to be analyzed is represented as Prolog facts. The detection process is therefore performed through the application of Prolog queries.

Keller et al. proposed SPOOL [KSRP99] to detect design patterns inside C++ systems, basing on design patterns structural representations. Information needed to detect patterns and extracted from the analyzed systems is codified in UML/CDIF format, while the patterns are depicted as abstract design components and stored in a repository.

FUJABA (From UML to Java and Back Again) [NNZ00] defines sub-patterns to categorize the structural recurring elements of design patterns, trying to build a detection algorithm based on the recognition of these elements and on their incremental combination towards design patterns. The formal basis of FUJABA exploits the graph grammar. Sub-patterns and patterns are expressed as graph transformation rules. The information extracted from the source code is represented as an AST, further enriched with annotations during the

design pattern detection process. These annotations aim to indicate the presence of design patterns or sub-patterns in the analyzed AST and are added to an AST though transformation rules. Additional nodes and edges inserted in the AST correspond to the identified sub-patterns. To address the large number of possible implementation variants, the FUJABA rules are enhanced with fuzzy values [Nie02] to describe a degree of uncertainty allowing one rule to match several implementations with a certain degree.
Kim and Boldyreff proposed a metrics-based approach for the detection of design patterns [KB00]. Three categories of metrics are considered: object-oriented, structural and procedural metrics. Each design pattern is characterized by a signature, which is derived by the calculation of the metrics on the design patterns proposed by Gamma. The search algorithm compares the metrics of each class with the patterns signatures.
In the JBOORET tool [MXY01] Hong Mei, Tao Xie and Fuqing Yang adopt a parser-based approach to assist the activity of extracting the higher-level design and source models from system artifacts. A conceptual model is formulated as the knowledge representation. Multi-perspective design and source models are recovered by JBOORET based on the comprehensive program information extracted from source code.
Smith and Stotts [SS03] proposed SPQR (System for Pattern Query and Recongnition), which is based on a first identification of elemental design patterns (EDPs), which are further analyzed in order to deduce the presence of pattern instances inside the analyzed systems. EDPs inherit from design patterns their ability to capture design intents, but are significantly simpler than design patterns. EDPs and their composition rules are formally expressed in terms of rho-calculus [SS03], which represents a subset of sigma-calculus properly extended with new reliance operators. Design patterns and their implementation variants are not statically described, but they are dynamically inferred through the formalized rules.
Heuzeroth et al. [HHHL03] introduce an approach to design pattern detection that combines both static and dynamic analysis techniques, where static analysis techniques based on the exploration of the ASTs of the source code are used first in order to find pattern candidates, which are to be inspected by a dynamic analysis stage. They further propose algorithms for the detection of several structural and behavioural patterns, focusing on the identification of design pattern instances in the Java Swing library.
Birkner [Bir07] presents the Pattern Detection Engine (PDE), which combines static and dynamic analysis techniques for the detection of a large part of the design patterns defined in Gamma [GHJV94]. Static analysis is first used in order to create static definitions of design patterns starting from UML class diagrams. In this phase, design pattern candidates are extracted from the subject system. Dynamic analysis techniques are then applied on the candidate instances in order to obtain dynamic definitions from UML

sequence diagrams. This process allows for the refinement of the candidate instances and for the obtainment of the detection results.
In [BF03] Balanyi and Ferenc introduce an automated approach to detect design patterns in C++ systems through a pattern miner algorithm. The system to be analyzed is represented as an Abstract Semantic Graph (ASG), while the patterns are stored in an XML-based language named DPML. The algorithm tries to match the XML tree obtained from the DPML description to the ASG representing the subject system.
Beyer and Lewrentz developed CrocoPat [BL03, BNL05], a tool which exploits relational expressions to specify the properties of a system. It adopts a three-step approach. In the firs one, the data to be analyzed are extracted from source code. In the second step the patterns to be detected are defined using a pattern specification language making use of the relations stored in an ad-hoc file. In the final step, the tool translates the relations into binary decision diagrams [BL03, BNL05].
In [WT04] Wang and Tzerpos introduce DPVK, a reverse engineering tool to detect pattern instances in Eiffel systems. In order to get better detection results, we analyze many different patterns and examine Eiffel software in terms of both static structure and dynamic behaviour.
Fabry and Mens [FM04] introduce a language-independent meta-level interface that can be used to extract complex information about the structures (and as a consequence, the patterns) that compose an object-oriented system, providing examples of their techniques on medium-sized systems implemented in Java and Smalltalk.
In [Gue05] Guéhéneuc introduces PTIDEJ (Pattern Trace, Identification, Detection and Enhancement in Java), which allows to create a model of a program from its source code and to identify micro-architectures that are similar to a design pattern. These micro-architectures are detected by an explanation-based constraint solver [Jus01], and represent architecures that are comparable with what the author calls *design motifs* (that are the solution parts of design pattern definitions [Gue05]). As the identification of these micro-structures is difficult, due to the huge amount of combination of classes, especially in large software systems, in [GSZ04] Guéhéneuc, Sahraoui and Zaidi propose a study on classes playing roles in design motifs using metrics and a machine learning algorithm to calculate fingerprints (i.e. sets of metrics values that can help in identifying classes playing a given role in a design pattern) in order to identify design motifs roles. They also show how the use of fingerprints can help to significantly reduce the search space of micro-architectures, focusing on the identification of the Composite design pattern [GHJV94] inside the JHotDraw framework [JHD].
In [GA08] Guéhéneuc and Antoniol propose DeMIMA, an approach to semiautomatically identify architectures that are similar to design patterns from source code. DeMIMA is composed by three layers: two layers are devoted to recover an abstract model of the

source code, including binary class relationships, while the third layer is actually aimed to identify pattern instances inside the abstract model.
Tsantalis and Chatzigeorgiou [TC06] have developed Design Pattern Detection Tool, a tool for design pattern detection that is based on structural similarity scoring between program graph representations and their vertices. Both the systems under analysis and the patterns to be recognized are described with set of matrices representing the various aspects of their static structures. A graph similarity algorithm calculates similarity scores between the graph representations of the analyzed systems and of the patterns. The approach claims to recognize also patterns that are slightly different from their basic representation, allowing the recognition of an important amount of pattern variants.
Philippow et al. [PSRN05] propose a technique for design pattern detection that is based on minimal key structures, i.e. the minimal class and object structure that has to be present in order to identify the pattern, and on positive and negative search criteria, the former collecting criteria that will occur with high probability in the implementation of particular patterns, the latter identifying those relationships that are not allowed in the context of a pattern, in order to reduce false positives cases.
In [CDD+05] Costagliola et al. propose an object oriented design pattern recovery approach which makes use of a design pattern library, expressed in terms of visual grammars, and based on a visual language parsing technique. They also present a visual environment which supports the pattern recognition process by automatically retrieving design patterns from imported UML class diagrams.
De Lucia et al. presented an approach for the identification of structural design patterns from object oriented systems [DDGR09]. Tha approach follows a two-step detection process. At first, pattern instances are identified only by considering their structure, exploiting a parsing technique for visual language recognition. In the second step, the candidate instances are validated by a source code analysis phase. They developed a tool named Design Pattern Recovery Environment for the detection of patterns, experimenting it on six public-domain programs and libraries.
Shi and Olsson [SO06] propose PINOT, a design pattern detection tool that has been developed around the modification of the Jikes Java compiler [Jikes], with the objective to achieve better performances and accuracy in the detection of more design patterns with respect to other tools. Shi and Olsson reclassify the GoF patterns in five categories (language provided, structure driven, behaviour driven, domain specific and generic concepts), and defined different detection techniques and algorithms according to each of the newly defined categories.
In [PL06] Niklas Pettersson and Welf Loewe propose a method to improve the performance of pattern detection. It is based on the idea of filtering information from the program representation (graphs), which is unnecessary for detecting a particular pattern.

This makes the remaining program representation graph planar, in many cases, thus allowing for linear pattern detection.
In DP-Miner tool [JDY07], Dong Jing et al. present a novel approach to discovering design patterns by defining the structural characteristics of each design pattern in terms of weight and matrix. Our discovery process includes several analysis phases. Our approach is based on the XMI standard so that it is compatible with other techniques following such standard. We also develop a toolkit to support our approach.
Dietrich and Elgar [DE07] present an approach to the formal definition of design patterns using the OWL web ontology language, and introduce the Web of Patterns prototype for the detection of the so defined patterns in Java systems.
The D-cube tool [SW08] presented by Krzysztof Stencel and Patrycja Wegrzynowicz is able to detect nonstandard implementation variants of design patterns. It is customizable because an analyst can introduce a new pattern retrieval query or modify an existing one and then repeat the detection using the results of earlier source code analysis stored in a relational database.

2.1.3. Comparisons among tools for design pattern detection

Although there are so many approaches to design pattern detection, the variants problem and the differences among the detection techniques often cause differences in the detection results provided by the various tools, even on the analysis of the same target systems. Therefore, it looks interesting to compare the various approaches to design pattern detection, especially focusing on the results they are able to produce. In [AMRT05] we present a comparison among two different tools for design pattern detection, namely FUJABA [NNZ00] and SPQR [SS03], which are based upon the decomposition of design patterns themselves. The comparison considers two main issues: the design pattern definition method adopted by each tool, and the design pattern detection strategy each tool adopts, discussing possible pros and cons for both the analyzed approaches.
In [GMW06] a general framework for the comparison of design recovery tools (hence not only for the comparison of design pattern detection tools) is proposed, illustrating how the framework can be applied to the comparison of two different tools, namely PTIDEJ [Gue05] and LiCoR [Licor].
Furthermore, benchmark proposals to provide evaluations about design pattern detection tools have been recently presented in [ATZ08, FHFG08], even if a standard benchmark platform is not yet available, and a real and effective comparison among them is very difficult.

2.2. Related works on software architecture reconstruction

While there exist many tools for design pattern detection, but which are basically devoted to the very same activity (even if pursued with varying strategies), the situation for software architecture reconstruction (SAR) tools is far more complex. Each tool is heavily different from one another in several aspects. First of all, each tool has different objectives and aims from the others, and is devoted to the extraction of a well defined set of information that make it unique among the other tools. The produced results are consequently different. The results may differ in their representations (some tools may adopt graphical views, while others software metrics, or textual representations), in their meaning (as each tool is focused on particular characteristics of the systems to be analyzed), and in their interpretation (some results may be of easy interpretation by the user, while others may need efforts or further analyses in order to be completely captured). Tools also differ in the formalisms and theoretical bases they adopt, which strictly depend on each single tool objectives.
In this section we present a set of tools and approaches which we consider of particular interest, or which we had the possibility to test and experiment. Different exhaustive taxonomies for SAR tools have been proposed, and frameworks for the comparison of these tools have also been introduced: both taxonomies and comparative frameworks are discussed, and the proposal for a new framework for the evaluation of SAR tools is presented.
Some authors consider some of the tools that we describe in the following as simply tools for software or structural analysis. As already mentioned in the introduction, in this dissertation we consider, under the category of SAR tools and approaches, also tools that revealed useful to support program comprehension, the reconstruction of views about the subject systems and the evaluation of software metrics, which are all activities helping in the software evolution and maintenance phases (which are a core aspect of SAR activities).

2.2.1. Approaches and tools for software architecture reconstruction

We now introduce some of the most common and known tools for software architecture reconstruction. Some of them come from the Academia, others are on the other hand commercial tools.

Alborz

Alborz [SYS06], developed by Kamran Sartipi, is a toolkit for the recovery and evaluation of the architecture of a software system, using techniques for pattern-matching, data mining, clustering, and quality evaluation. The result of the recovery is provided to the user through hypertext pages and graph visualization techniques and tools. The process of architecture reconstruction consists of two phases. In the first phase, Alborz parses the source code of the subject system translating it into a graph; then it splits the graph in cohesive regions through data mining techniques. In the second phase, the user can specify the architectural views he wants to obtain in terms of patterns. The defined patterns are then mapped with graph regions built during the first phase using graph matching and clustering techniques. Finally, the user can iterate the process again, basing on the partially reconstructed architecture and evaluation information provided by the tool. In our experience, the usage of this tool revealed very complex, especially due to the lack of documentation and explanations about how to define patterns and how to extract them. In our opinion, a tool for software architecture reconstruction shall provide the user with a set of self-contained functionalities that can report results of immediate exploitability, and should avoid over complicated tasks to be manually performed by its users.

Armin

Armin (Architecture Reconstruction and Mining) [KBV03] has been developed by the Software Engineering Institute at Carnegie Mellon University, in collaboration with Robert Bosch Corporation. Armin is a modelling and visualization tool, which accepts inputs in the RSF format [Mül93]. In this format, couples of entities of the system (that can be classes, interfaces, methods and so on) are textually expressed, specifying the kind of relationships connecting them. Once the data about a system have been imported, a scripting engine allows for the manipulation of the entities and relationships, and also for the execution of user-defined queries, in order to obtain architectural views that are of particular interests for the users.

Bauhaus

The Bauhaus toolkit [Bau] is an environment composed by a set of tools to extract, analyze, query and visualize information about existing software, supporting software architecture reconstruction from source code analysis. To support these activities, it provides several techniques, like metrics computation, pointer, side-effect and data flow analyses, program slicing, source code navigation and so on.

CodeCrawler

CodeCrawler [Lan03] is a language independent reverse engineering tool combining metrics and software visualization. The system to be analyzed must be translated with an ad-hoc parser into a correspondent FAMIX model [DTD01], which constitutes the actual input of the tool. The core concept which characterizes CodeCrawler is the *polimetric views* [Lan03]: the tool provides the user with a set of views about the analyzed systems that are

generated by the calculation of at most five metrics on the entities to be visualized. Each entity (that can be either a class or interface, or a method, or an attribute, depending on each single view) is depicted as a rectangle, with five associated parameters: width, height, horizontal position, vertical position, and colour. Each parameter can be associated to a metric, and the size of each parameter depends on the value the correspondent metric assumes for that entity. CodeCrawler distinguishes two sets of views: *coarse-grained views* are at a higher level of abstraction, and let the user manage the system in its generality. The system complexity view, for example, represents the analyzed system in its constituting hierarchies. Each entity depicts either a class or an interface, edges connecting them represent the inheritance relationships, and the complexity of each entity is calculated in terms of their number of attributes (associated with its width), the number of methods (related to its height), and the number of lines of code (associated with its colour). On the other hand, *fine-grained views* are more low level, and let the user focus on the details of each single class. As an example, the class blueprint views, shows the attributes and methods for a given class or interface. Attributes and methods are classified according to their nature with particular colours, and relationships among attributes and methods are depicted as edges connecting them.
More recently, an open source software visualization plug-in for Eclipse, named X-Ray [XRay], has been developed. It currently provides the system complexity view and the class and package dependency view for a given Java project. Being an Eclipse plug-in, it can directly analyze a selected project without the need for the creation of an external FAMIX model or other kinds of models, improving therefore the usability and effectiveness of the approach.

CodeLogic

CodeLogic [CL] is a commercial tool for the analysis of systems written in Java. The functionalities offered by the tool are restricted to the generation of UML class diagrams, sequence diagrams and flow charts. The results can be exported in the PNG, Visio .Net and XMI Rational formats. No further functionalities or analyses are provided by the tool, which demonstrates to be quite limited, if compared with other available approaches.

Dali

Dali [KC99], proposed by Kazman et al., is a workbench of tools for software architecture reconstruction. The workbench is based on Rigi [MWT95] for the visualization and handling of the views generated by Dali. Users can define personal query patterns in order to generate various views of the system at various levels of abstraction. System information must be extracted by software analysis tools, or provided by other forms of documentation, and then must be loaded into Dali. The information is stored in a PostgreSQL database and then visualized in Rigi.

Doxygen

Doxygen [Doxy] is a documentation tool that can be used on systems written in a variety of programming languages, such as C++, C, Java, Objective C, Python, IDL, Fortran, VHDL, PHP, C#. Its aim is to provide an organic and exhaustive documentation for those systems whose documentation is scarce, not up-to-date, incomplete, or lacking at all. The generation of documentation is performed through source code analysis, thus maintaining the documentation more consistent with the system. The generation process can be guided by a wizard, and the output can be exported in several formats, like HTML, PostScript, RTF, Latex and other kinds of documents. Doxygen can also generate views about the analyzed system, which are integrated with the documentation. The *type graph* shows the classes and interfaces that constitute a package, and the relationships among them (inheritance, implementation, association). The *caller* and *callee graphs* are generated on methods, and show respectively which methods call the subject method, and which methods are called by the subject methods, adding a sort of dynamic information to the generated documentation. Even if the use of views within a system's documentation is a promising idea, actually the navigability of such views proved to be quite difficult, especially while handling large packages with many classes. In fact, as the generated views are part of the textual documentation, they cannot be modified, extended or reduced.

JDepend

JDepend [JDepend] is a tool for the generation of design quality metrics about the packages composing the analyzed systems. The quality of design is measured according to three main concepts, namely extensibility of the system, reusability and maintainability. Actually, this tool doesn't reconstruct any views, documentation or reports about a system, but can still be used as a complementary mean to SAR activities. The metrics analysis can help the engineers to understand and to manage the criticalities of the system in terms of its structure. This evaluation leads to focus precisely on those parts of the system that reveal more problematic. In this way, the engineers can then use a third-party tool to apply views and evaluations on those modules and components, without the need to analyzed the whole system before reaching the critical issues.

Rational Software Architect

Rational Software Architect (Rational, IBM) [RSA] is an industrial tool that can be exploited both for forward and reverse engineering purposes. One interesting feature of this tool is that it provides consistency mechanisms among the different UML diagrams and the code with respect to round-trip engineering. It also provides support for a sort of impact analysis that is very similar to the What-if analysis provided by SA4J. Another peculiarity is that Rational Software Architect supports dynamic analysis since it is able to reconstruct sequence diagrams, and it also supports design pattern detection for a small set of design patterns.

Shrimp-Creole

Shrimp-Creole (Simple Hierarchical Multi-Perspective) [SM95] is an Eclipse [Eclipse] plug-in which can be used to give a graphical representation of a subject system and explore its architecture at different detail levels. The tool exploits different layouts to depict the involved entities, and can show only the entities and relationships of actual interest through a filtering process. Layouts can be tailored on the user preferences and needs, and a *filmstrip* functionality allow to get different pictures of the analyzed system, that can be analyzed in a second time.

Structural Analysis for Java

Structural Analysis for Java (SA4J) [SA4J] has been developed by IBM, and provides the user with lots of functionalities, views, metrics, analyses and reports. SA4J is based on the concept of *dependencies* and *dependents* of a given entity, either a package or a class or an interface. The number of dependencies of an entity is the number of entities the subject entity depends from. The number of dependents of an entity is the number of entities that depend on the functionalities provided by the subject entity. The adoption of these concepts allowed for the introduction of functionalities that reveal very useful in a software architecture reconstruction and evaluation process. Besides the exploitation of views that are very similar to UML class diagrams, and which let the user understand the actual physical structure of the analyzed systems, SA4J allows the detection of structural antipatterns, like butterflies, breakables, hubs and tangles, the evaluation and the analysis of a good number of metrics about classes and packages and of the stability of the system, a skeleton view to determine which components rely on top of the others and consequently affect the system stability, and a what-if analysis, which dynamically shows the entities that are involved if a certain part of the system changes.

Swagkit

Swagkit [Swag] is a toolkit developed by the Software Architecture Group at the University of Waterloo (SWAG), that can be used to extract, abstract and explore software architectures. Currently, Swagkit supports the extraction from C/C++ code, the abstraction to the architectural level and the presentation in a landscape form.

Swagkit uses the *compiler wrapping* technique to obtain information about source code. It consists of substituting the compiler and linker callings with specific commands, that perform compiling and linking combined with information extraction. The output of the extraction process is formed by the so-called, graphical layout models of the analyzed software system and of its subsystems. A landscape is a graph with multiple named types of nodes and multiple types of edges. At least one type of edges must be named "*contain*": it forms either a tree or a forest connecting all the nodes in the graph and denotes the containment structure of the landscape. The remaining edge types do not have any restrictions (they can cross containment levels, form loops, etc). Both nodes and edges can have additional attributes, specifying their names, types, ways to display them, and other various properties. This kind of extracted models provides the user with a large amount of information that abstracts from the system's implementation and eases its understanding. For example, landscapes help in identifying modules with a huge number of interrelationships, which are symptoms of a high degree of complexity, leading the engineer to concentrate on these modules that are supposed to be the main candidates for system refactoring.

Symphony
Symphony [DHK+04b] is an approach to software architecture reconstruction based on views. It defines a general reconstruction model based on the concepts of viewpoints and views as defined by the IEEE 1471 standard [IEEE]. The Symphony approach consists of two stages. In the first stage, *reconstruction design*, the reconstruction problem is analyzed, viewpoints [IEEE] and corresponding views are defined, and mapping rules from source to target views are formalized. In the second phase, *reconstruction execution*, the subject system is analyzed, the source views are extracted and then mapped to populate the target views through the application of the mapping rules. the two stages may be iterated. The execution may reveal new reconstruction necessities, leading to a refined understanding of the problem and a more detailed reconstruction design. The execution phase follows the well-established extract-abstract-present approach, tailored to the specific needs of architecture reconstruction.

Understand for Java
Understand for Java [U4J] offers different views that may be generated at different detail levels, from the whole system to single classes. Therefore, they are useful both to get a general comprehension of the system and to analyze single components in detail. Anyway, Understand for Java is mainly focused on the computation of metrics for software quality and complexity evaluation.

Many other tools and approaches for software architecture reconstruction and program comprehension actually exist, like ART [FATM99], DiscoTect [YGS+04] and QUADSAR [SRBV06], and those considered for example in [BSV02, PDP+07].

2.2.2. Taxonomies and comparisons of software architecture reconstruction tools

The heterogeneity, the quantity and the variety of SAR tools and approaches currently available, both coming from the academia and from the industry, stimulated the proposal of taxonomies of tools and approaches, and led also to the definition of possible comparative frameworks and benchmarks to evaluate SAR tools.
In [BSV02], Stoermer et al. provide a taxonomy for 13 approaches and tools for software architecture reconstruction. In the same context, they define six practice patterns which presented while applying architecture reconstruction techniques in industrial settings, and which describe recurring situations where problems can be solved by applying well-known strategies. The presented approaches are collected in four defined categories: exclusively manual approaches, manual reconstruction approaches with tool support, strategies based on query languages for reconstruction, and other kinds of approaches, like the use of clustering and data mining techniques, or the use of architecture description languages.
Pollet et al. [PDP+07] suggest a process-oriented taxonomy of 35 tools and approaches, evaluated according to five axes: the planned goals, the kind of processes followed by each approach (being either top-down, bottom-up, or hybrid), the kind of input received by each tool, the techniques adopted in the reconstruction process (distinguishing among quasi-manual, semi-automatic and quasi-automatic), and finally the kind of output produced by the tools.
As far as comparisons among SAR tools are concerned, several researchers have focused their attention in defining characteristics that may make such comparison possible. To cite some examples, Guéhéneuc et al. [GMW06] propose a comparative framework for design recovery tools, based on eight concerns (context, intent, users, input, technique, output, implementation and tool); they apply the framework to compare only two design recovery tools, namely PTIDEJ [Gue05] and LiCoR [Licor]; further validation is required to verify whether the framework enables an objective comparison of tools, and to allow also the evaluation of other SAR tools. Bellay and Gall [BG97] propose a comparison of four reverse engineering tools introducing four functional categories that help in the assessment of the reverse engineering tools, namely analysis, representation, editing/browsing and general capabilities. Gorton and Zhu [GZ05] pointed out the complementariness of reconstruction tools discussing the analysis of a small industrial

system (296 classes) through four tools: Understand for Java [U4J], JDepend [JDepend], SA4J [SA4J], Armin [KBV03] and Enterprise Architect [Ent]. Jha et al. [JMP04] report a comparison experience about four reconstruction tools by considering their extraction, abstraction and visualization capabilities, their supported languages and their completeness with respect to architecture reconstruction purposes. They concentrated their attention on reconstruction tools supporting C or C++ languages.

2.2.3. A novel comparative framework proposal for SAR tools

In [AM09] we proposed a comparative framework for architecture reconstruction tools. In our framework, six aspects concerning the tools are to be considered. First of all, we consider implementation issues like the language the tools have been developed with, the supported operating systems, the eventual third-party software needed to correctly run the tool, the latest available release and the occupied space.
All this information is useful in order to get an immediate understanding of the target the tools refer to. Another important kind of information is related to the input each tool accepts, i.e. the supported programming languages and the type of information source needed by the tool (like source code, byte code, or different models). The principal relevance is given to the output produced by the tool. We inspect if each tool is able to recover views about the analyzed systems, to calculate metrics on them, to generate documentation for them, and to eventually detect design patterns or antipatterns from the subject systems. We next, indicate if the tools are documented, and we give an evaluation of the documentation quality. In our opinion, usability is a core issue. Having a tool that is simple to install, to manage and to use will make the users more productive and active in their tasks. Finally, we report considerations about the carried out experimentations, tracing the possible problems encountered while using each tool.
As an example of the application of the framework, in [AM09] we discussed seven tools for architecture reconstruction, and evaluated them according to the six aspects just described. The results are replicated in Table 2.1, which reports the comparative evaluation of four out of the seven considered tools. From this table, it can be immediately noticed how the tools are heavily different from one another both in terms of their accepted inputs and in terms of the produced artifacts and of their usability. In the paper, we also described which are the peculiarities that characterize each tool with respect to the others, in order to better focus on the target users and reconstruction activities each tool refers to. Actually, providing an effective comparison is difficult, due to the different nature of each system and the functionalities it provides, and also because the evaluation of some of the systems' characteristics is mainly subjective.

		CodeCrawler	Doxygen	SA4J	MARPLE SAR
IMPLEMENTATION	*Language*	Smalltalk	C++	Java	Java
	Supported Platforms	Windows, Linux, MacOS	Windows, Linux, MacOS	Platform independent	Platform independent
	Third-party required software	FAMIX model generator	GraphViz	None	Micro-structures detector
	Latest release (version/date)	4.5 March 2004	1.5.8 27 December 2008	1.0 March 2004	1.0 July 2008
	Occupied space	24 Mb	17 Mb	47 Mb	1 Mb
INPUT	*Supported programming languages*	Language independent	Java, C, C++, Python, Objective-C, IDL, Fortran, VHDL, PHP, C#	Java	Java
	Information source type	FAMIX model	Source code	Source code, byte code	Software micro-structures
OUTPUT	*Architectural views*	Polymetric views	UML-styled type diagrams	Package explorer, Skeleton	Class compact, Class extended, Package diagrams
	Behavioural views	Polymetric views	Call graphs	What-if	None
	Metrics	Yes	No	Yes	Yes
	Documentation generation	No	Yes	No	No
	Design pattern detection	No	No	No	Yes
	Antipattern detection	No, but supported by the manual analysis of the generated views	No	Yes (butterflies, breakable, hubs, tangles)	Yes (butterflies, breakable, hubs, tangles)
	Other defects or micro-structures detection	No, but supported by the manual analysis of the generated views	No	No	Micro patterns [GM05] devising non-object-oriented programming practices
DOCUM.	*Kind of documentation*	None	HTML	PDF manual	None
	Documentation quality	n/a	Good	Very good	n/a
USABILITY	*Installation simplicity*	Not straightforward	Not straightforward	Very simple	Simple
	Use simplicity	Not simple	Quite simple	Simple	Simple
	Results self explanation	Not self explaining	Self explaining (even if views are often too large)	Self explaining (even if views can be too overwhelmed)	Self explaining (even if views can be too overwhelmed)
EXPERIMENTATIONS	*Case studies*	JasperReports Industrial case studies JHotDraw Plazma Sakai	Adempiere JasperReports Plazma Sakai	Adempiere JasperReports JHotDraw Plazma Sakai	Industrial case studies JHotDraw Plazma
	Traced problems	Generation and load of the FAMIX model not always possible. Graphical bugs using CodeCrawler under Windows	None	None	Seldom out of memory exception in the case of very large systems

***Table 2.1** – A comparison among four software architecture reconstruction tools*

Currently, no silver bullet tool exists, but the tools can be considered somehow complementary to each other. In our work we also tried to identify some of the most common problems we have to face for re-engineering and reverse engineering activities, like for example design evaluation, integration, systems refactoring and restructuring, migration towards SOA and so on, and we suggested the more useful tools and views to face these problems, according to our experiences on the analysis of systems of different sizes.
The choice of a reconstruction tool may also be guided by the size of the analyzed systems. While analyzing small or medium sized systems, we would probably need views that may expose some details of the architecture and of the nature of the various system components. On the contrary, these "detailed" views would add too much information to the comprehension of large systems, for which we are probably interested in obtaining the highest abstraction level as possible, in order to have a general understanding of the system, and eventually decide on which components to focus with more detailed views.
Our comparison proposal differs from those introduced in Section 2.2.2 for different points of view. First of all, we focused on users which should be guided to the choice of a particular tool according to their main needs, in terms of systems to be analyzed, data to be extracted and ease of use and exploitation. Therefore, in this comparative framework we resembled the aspects we think that are of crucial importance while having a first contact with a SAR tool of interest. Finally, we also tried to be as more objective as possible: the considerations we carried out on subjective or non-quantifiable aspects are derived by the general opinion of a pool of people who actually experimented the considered tools.

Chapter 3

Software micro-structures

Abstract

Software micro-structures are the core concept of this thesis. In this chapter we introduce three categories of micro-structures, namely Elemental Design Patterns (EDPs), design pattern clues (DP clues) and micro patterns, that we consider as the basic bricks both for design pattern detection activities (as they can be interpreted as hints for the presence of design patterns inside the analyzed software systems) and for software architecture reconstruction tasks (as they can be used to identify structural relationships among the classes composing a system). As they have never been considered as similar elements in the literature, and as their definition is not formal and thus may lead up to ambiguities, in this chapter we propose a redefinition of the analyzed micro-structures which tries to solve the possible ambiguities and to give a new interpretation of them basing on common concepts.

3.1. Software micro-structures

Many different kinds of micro-structures currently exist. They have different aims, different definitions and different detail levels. In this thesis we concentrate on three different categories of micro-structures, that are Elemental Design Patterns (EDPs) [SS03], design pattern clues (DP clues) [Mag06b, AMR09a, AMR09b] and micro patterns [GM05]. They have been defined to support different tasks: EDPs represent programming techniques that are in the everyday practice of each programmer, codifying them in the form of simple design patterns; DP clues are hints for the presence of design patterns inside a software system, and have been introduced by analyzing typical DP structures and implementations; micro patterns devise particular classes with peculiar conditions on their attributes and methods. Other micro-structures have been defined, like the sub-patterns exploited by the FUJABA approach [NNZ00] and also considered in [AMR09a], but have not been taken into account in our thesis work.

3.1.1. Elemental design patterns

Elemental design patterns (EDPs) were proposed by Smith and Stotts [SS02]. They provide solutions to very common programming problems (we can state that these problems occur in the everyday practice of each programmer). They share the same aim with the design patterns, but they are applied to more restricted and specific issues. In fact, if design patterns propose solutions to problems which can involve a certain number of classes, EDPs address problems of much more limited dimensions, which generally do not involve more than three classes. There are 16 EDPs subdivided into three categories:

- *Object Elements*: contains three EDPs related to the creation and the referencing of objects as well as to the presence of abstract methods inside an abstract class, or interface methods inside an interface;
- *Method Invocation*: collects twelve EDPs which represent the various forms of possible method calls;
- *Type Relation*: contains a single EDP representing the inheritance relationship between two classes, as well as the implementation of an interface.

EDPs are defined with the same description structure used in [GHJV94] for the presentation of design patterns. For a complete description of each EDP refer to [Smi02]. EDPs can be detected inside Java systems through the *Micro-structures detector* module, which is part of the MARPLE (Metrics and Architecture REcontruction PLugin for Eclipse) project [Arc06]. MARPLE has been developed for design pattern detection and software architecture reconstruction purposes, and it is based on the detection of micro-structures. The Micro-structures detector module is briefly described in Section 3.3. However, the definition of EDPs originated by considerations made on C++ source code. Smith and Stotts developed SPQR [SS03], an approach to design pattern detection based on the identification of EDPs inside the subject systems.

3.1.2. Design pattern clues

We have introduced design pattern clues (DP clues) [Mag06a, MATZ09] as possible hints about the presence of design patterns inside the code, by manually analyzing design pattern architectures and sample implementations identifying basic structures which are peculiar for each single pattern. Currently, we have defined 41 design pattern clues subdivided in the following eight categories:

- *Class Declaration Information*: collects clues which are identifiable at the class declaration level;
- *Multiple Classes Information*: collects clues that can be identified by the comparison of at least two classes and their contents;

- *Instance Information*: contains clues regarding particular instances of a certain class, and one clue representing the controlled instantiations;
- *Method Signature Information*: collects clues which are identifiable analysing the signature of a method;
- *Method Body Information*: contains those clues that can be identified by only analyzing the body of any kind of methods;
- *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.

A complete description of design pattern clues can be found in [MATZ09].
Clues are detected inside a Java system through the Micro-structures detector module of MARPLE.

3.1.3. Micro patterns

Micro patterns were introduced by Gil and Maman [GM05] in order to capture some very common programming techniques. Micro patterns can be thought of as *class-level traceable patterns*, i.e. structures similar to design patterns which can be mechanically recognized and which stand at a class abstraction level. A micro pattern is traceable if it can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components. Currently, there are 27 micro patterns subdivided into eight categories. Authors do not assert that the set of the identified micro patterns is complete or exhaustive.
The eight micro pattern categories are:

- *Degenerate State and Behaviour*: this category includes micro patterns describing interfaces and classes whose state and behaviour are degenerated. In most cases this means that the interface or class does not define any variable or method;
- *Degenerate Behaviour*: these micro patterns are related to classes with no methods or with very simple ones;
- *Degenerate State*: this category is related to classes which have no state (i.e. variables), or their state is shared with other classes or they are immutable;
- *Controlled Creation*: the micro patterns belonging to this category describe special protocols for creating objects;
- *Wrappers*: this category collects micro patterns dealing with classes which have a single central instance field and methods working on it, so that the main functionalities are delegated to this field;
- *Data Managers*: these micro patterns are related to classes whose main purpose is to manage the data stored in a set of instance variables;

- *Base Classes*: the micro patterns belonging to this category describe different ways in which a base class makes preparations for its sublcasses;
- *Inheritors*: the micro patterns in this category correspond to three ways in which a class can use the definitions of its superclass, i.e. abstract method implementation, method overriding and interface enrichment.

For a complete discussion about micro patterns refer to [GM05]. Micro patterns are recognized through a prototype developed by Gil and Maman based on byte code analysis [GM05], and with MARPLE, adopting source code analysis.

3.2. Towards a unique micro-structures catalogue

Even if these micro-structures are so different in nature and aims, they can all be exploited (with different degrees of usefulness) both for design pattern detection and software architecture reconstruction activities. There are two main disadvantages concerned with the definitions of micro-structures:

- often, their definition is not formal, and may result ambiguous;
- they have never been considered as similar elements, even if they can all be automatically detected from a static source code analysis process;

To solve these issues, we propose to introduce a unique catalogue of micro-structures that resembles EDPs, clues and micro patterns redefining them in terms of concepts that are common to all the categories of micro-structures we consider, and that will be exploited in their definitions. We call these common concepts *code atoms* (or *atoms* for brevity). Code atoms are simple code elements (more than the micro-structures) that will be used to provide a new and more formal definition of micro-structures. In the new definition of micro-structures, we will use these atoms, and eventually any micro-structure the element to be defined depends on. The new definition will provide an unambiguous meaning to each micro-structure, and will generate a unique catalogue of micro-structures based only on common concepts.
The elements and concepts that will be used in the definition of the atoms and of the micro-structures will now be defined. These concepts are strictly related with the object-oriented paradigm. As we focused in particular on Java systems, we will consider this language as our target.
Any object-oriented system is based on the key concept of type. We will use T to denote a type. A type can itself be either a class (denoted by C) or an interface (denoted by I). If we

deal with a set of types, classes or interfaces, each of them will be specified by an index: *Ti* will be therefore the *i*-th type out of a set of n types *T1, …, Tn.* The same considerations reflect on classes and interfaces.
Given a type *T*, we can obtain information about it through the following statements:

- *name(T)*: it represents the qualified name of the type, i.e. the name of the class or interface denoting it, preceded by its package name;
- *attributes(T)*: it represents the set of attributes that have been defined by *T*;
- *methods(T)*: it represents the set of methods that have been defined by *T*;
- *inst(T)*: it represents a generic instance of *T*, that can have been created either within *T* (therefore it can be handled as an attribute of *T*) or within another type.

Given an attribute *a* ∈ *attributes(T)*, the following statements are defined:

- *name(a)*: it represents the name of attribute *a*;
- *typeOf(a)*: it represent the type of *a*, which can be either a simple type, a type *T*, or a list of n attributes *list(a1, …, an)*;

Given a method *m* ∈ *methods(T)*, the following statements are defined:

- *name(m)*: it represents the name of method *m*;
- *constructor(m)*: it represents the fact that method *m* is a constructor;
- *returnType(m)*: it represents the return type of the method *m*, that can be either a type *T*, a simple type, or void;
- *params(m)*: it represents the set of formal parameters received in input by method *m*;
- *body(m)*: it represents the body of method *m*, i.e. all the statements and operations defined by the method. The body could also be empty: this aspect is represented by the "is empty" clause. The body can itself contain instances of atoms or micro-structures, or other kinds of statements: this containment aspect is specified by the "contains" relationship;
- *returnedValue(m)*: it represents a single returned value of the method;
- *returnStatements(m)*: it represents the set of return statements or return points specified by the method implementation;
- *typeOf(m)*: it represents the type that defined method *m*. As *m* ∈ *methods(T)*, therefore *typeOf(m) = T.*

Given two methods *m1* and *m2*, *m1 = m2* will indicate that the two methods have the same signature.

Within the body of a method we can find two special elements, that we call *containers*: they are *controlStatement*, which represents all the control structures that are available in the reference programming language (e.g. in Java, if and switch blocks), and *loop*, which represents all kind of loops available in the reference programming language (e.g. for, while, do-while, enhanced for). Both *controlStatement* and *loop* may operate on a set of parameters:

- *param(controlStatement), param(loop)*: it represents the set of attributes handled by the control statement or loop structure.

Another kind of statement that needs to be considered in order to correctly define the sets of atoms and micro-structures is the method invocation between two methods:

- *methodInvocation(m1, m2)*: it represents the invocation of method *m2* occurring within the body of method *m1*;

Given a method invocation, the following properties can be obtained:

- *source(methodInvocation)*: the actual object invoking *m2*, that is an instance of *typeOf(m1)*;
- *target(methodInvocation)*: the actual object on which *m2* is invoked, that is an instance of *typeOf(m2)*;
- *params(methodInvocation)*: it represents the set of actual parameters passed to the method invocation.

On both types, attributes and methods we can use the logical operators $\wedge$, $\vee$, $\neg$, $\forall$, $\exists$, $\exists!$, according to their usual meaning. Moreover, we use the cardinality operator | | to obtain the number of elements composing a specific set (e.g. |*methods(T)*| will return the number of methods defined in *T*). Finally the operator "is" will be used to declare that a type, an attribute or a method must satisfy a particular modifier (e.g. "*a* is private" means that the attribute *a* must be defined private).
Now that we have introduced the notions and concepts that will guide us in the definition and specification of micro-structures, each of them can be defined (according to its definition) on a type, on an attribute, or on a method:

- *micro_structure_name(T)*: the micro-structure is defined on type *T;*
- *micro_structure_name(a)*: the micro-structure is defined on attribute *a;*
- *micro_structure_name(m)*: the micro-structure is defined on method *m.*

However, the largest part of micro-structures represents information relating two entities; in this case, both the source of the micro-structure (i.e. the entity that actually represents it) and its destination (i.e. the entity the micro-structure depends on) must be specified (for example, *micro_structure_name(T1, T2)* represents a micro-structure that is implemented in *T1*, but whose existence is strictly related to *T2*).
With these definitions and concepts, we now provide the precise definition of atoms and micro-structures (collected in tables from Table 3.1 to Table 3.13).

3.2.1. Code atoms definitions

Atom category	Atom name	Atom definition	Explanation
Type atoms	Final class	*Final class(C)* iff *C* is final	The class is declared final.
	Abstract class	*Abstract class(C)* iff *C* is abstract	The class is declared abstract.
	Interface	*Interface(T)* iff *T* is interface	The considered type is an interface.
Method atoms	Controlled parameter	*Controlled parameter(m, ai), i* = *1, …, n*, iff *ai* ∈ *param(m)* ∧ *body(m)* contains *controlStatement*: *ai* ∈ *param(controlStatement)*	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.
	Inheritance this parameter	*Inheritance this parameter(methodInvocation)* iff ∃ *p* ∈ *params(methodInvocation)*: *p* = this	A method receives the same caller object as a parameter.
	Private constructor	*Private constructor(m)* iff *constructor(m)* ∧ *m* is private	A constructor is declared private.
	Protected constructor	*Protected constructor(m)* iff *constructor(m)* ∧ *m* is protected	A constructor is declared protected.
	Interface method	*Interface method(m)* iff ∃ *T*: *m* ∈ *methods(T)* ∧ *T* is interface	The considered method belongs to an interface.
	Abstract method	*Abstract method(m)* iff *m* is abstract	The considered method belongs to an abstract class.
	Getter	*Getter(m, a)* iff *body(m)* = return *a*	The method is a getter method, consisting of a single statement returning a value.
	Setter	*Setter(m, a)* iff *body(m)* = *assignment(a, value)*	The method is a setter method, consisting of a single statement setting a value.

Table 3.1 *– Type and Method atoms definitions*

Atom category	Atom name	Atom definition	Explanation
Attribute Atoms	Same class object	*Same class object(T, o)* iff ∃ *T*: *o* ∈ *attributes(T)* ∧ *typeOf(o)* = *T*	The considered object is an instance of the same class in which it is declared.
	Different class object	*Same class object(T, o)* iff ∃ *T*: *o* ∈ *attributes(T)* ∧ *typeOf(o)* ≠ *T*	The considered object belongs to a different class from the declaring one.
	Same hierarchy object	*Same hierarchy object(T1, o)* iff ∃ *T1, T2, T3*: *o* ∈ *attributes(T1)* ∧ *typeOf(o)* = *T2* ∧ *ancestor(T1, T3)* ∧ *ancestor(T2, T3)* ∧ *name(T3)* ≠ java.lang.Object	The considered object is an instance of a class T1 belonging to the same hierarchy of another considered class T2.
	Different hierarchy object	*Different hierarchy object(T1, o)* iff ∃ *T1, T2*: *o* ∈ *attributes(T1)* ∧ *typeOf(o)* = *T2* ∧ ¬∃ *T3*: *ancestor(T1, T3)* ∧ *ancestor(T2, T3)* ∧ *name(T3)* ≠ java.lang.Object	The considered object is an instance of a class T1 belonging to a different hierarchy from that of another considered class T2.
	Private flag	*Private flag(a)* iff *a* is private	The attribute is private.
	Static flag	*Static flag(a)* iff *a* is static	The attribute is static.
	Private object	*Private object(o)* iff *o* is private	The considered object is private.
	Static object	*Static object(o)* iff *o* is static	The considered object is static.
	Single object	*Single object(o)* iff *o* ∈ *attributes(T)* ∧ *typeOf(o)* = *To* ∧ ∃! *o* ∈ *attributes(T)*: *typeOf(o)* = *To*	The considered object is the only instance of a certain class declared in another class.
Return Type Atoms	Same object returned	*Same object returned(m)* iff *m* ∈ *methods(T)* ∧ *returnType(m)* = *T*	The method returns a reference to the same type in which it is declared.
	Different object returned	*Different object returned(m)* iff *m* ∈ *methods(T)* ∧ *returnType(m)* ≠ *T*	The method returns a reference to a different type from that in which it is declared.
	Simple type returned	*Simple type returned(m)* iff *m* ∈ *methods(T)* ∧ *returnType(m)* = boolean, char, int, double, long	The method returns a simple type.
	Void returned	*Void returned(m)* iff *m* ∈ *methods(T)* ∧ *returnType(m)* = void	The method returns void.
Returned Element Atom	Same hierarchy object returned	*Same hierarchy object returned(m)* iff ∃ *T1, T2, T3*: *m* ∈ *methods(T1)* ∧ *typeOf(returnedValue(m))* = *T2* ∧ *ancestor(T1, T3)* ∧ *ancestor(T2, T3)* ∧ *name(T3)* ≠ java.lang.Object	A method returns an object belonging to the same hierarchy of its declaring class.
	Different hierarchy object returned	*Same hierarchy object returned(m)* iff ∃ *T1, T2*: *m* ∈ *methods(T1)* ∧ *typeOf(returnedValue(m))* = *T2* ∧ ¬∃ *T3*: *ancestor(T1, T3)* ∧ *ancestor(T2, T3)* ∧ *name(T3)* ≠ java.lang.Object	A method returns an object belonging to a different hierarchy from that of its declaring class.

***Table 3.2** – Attribute, Return Type and Returned Elements atoms definitions*

Atom category	Atom name	Atom definition	Explanation
Instantiation and Assignment atoms	Object creation	*Object creation(T, o)* iff *o* is defined and created by *T*	An instance of a certain class is created in type T.
	(Attribute) assignment	*Assignment(a, value)* iff it exists an assignment *a* = *value*. *Value* can be either a simple value or the return value of a method invocation.	The considered statement is an assignment of an attribute.
	(Object) assignment	*Assignment(o, value)* iff it exists an assignment *a* = *value*. *Value* can be either an object creation, a reference to another object, or the returned object from a method invocation.	The considered statement is an assignment of an object.
Class Relationship atoms	Interface inherited	*Interface inherited(T, I)* if type *T* implements or extends interface *I*	The considered type implements or extends and interface.
	Class inherited	*Class inherited(C1, C2)* if class *C1* extends class *C2*	The considered class extends another class.
	Abstract method invoked	*Abstract method invoked(m1, m2)* iff ∃ *methodInvocation(m1, m2)* ∧ *m2* is abstract	A method invokes an abstract method.
	Ancestor	*Ancestor(T1, T2)* iff *T2* is a parent for *T1*, either direct or indirect, ∧ *typeOf(T2)* ≠java.lang.Object	Type T2 is an ancestor of type T1, i.e. it comes before in the hierarchy of T1, and it is not the java.lang.Object class.

Table 3.3 *– Instantiation and Assignment and Class Relationship atoms definitions*

3.2.2. Elemental design patterns definitions

EDP category	EDP name	Definition	Explanation [Smi02]
Object Elements	Abstract interface	*AbstractInterface(m)* iff *interface(typeOf(m))* ∨ (*abstract class(typeOf(m))* ∧ *abstract method(m)*)	It provides a common interface for operating on an object type family, but delaying definition of the actual operations to a later time.
	Retrieve	*Retrieve(o)* iff *o* ∈ *attributes(C)* ∧ ∃ *assignment(o, value)*: *value* = *returnedValue(m)* ∨ *value* = *o2* ∈ *attributes(C2)* ∧ *typeOf(o)* = *typeOf(o2)*	To use an object from another non-local source in the local scope, thereby creating a relationship and tie between the local object and the remote one.
Type Relation	Inheritance	*Inheritance(T1, T2)* iff *interface inherited(T1, T2)* ∨ *class inherited(T1, T2)*	To use all of another classes' interface, and all or some of its implementation.

Table 3.4 *– Object Elements and Type Relation EDPs definitions*

EDP category	EDP name	Definition	Explanation [Smi02]
Method Call	Recursion	*Recursion(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) = signature(m2)*	To accomplish a larger task by performing many smaller similar tasks, using the same object state.
	Conglomeration	*Conglomeration(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) ≠ signature(m2)*	To bring together, or conglomerate, diverse operations and behaviours to complete a more complex task within a single object.
	Extend method	*Extend method(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) = signature(m2)*	Add to, not replace, behaviour in a method of a superclass while reusing existing code.
	Revert method	*Revert method(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) = target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) ≠ signature(m2)*	Bypass the current class' implementation of a method, and instead use the superclass' implementation, reverting to an 'earlier' method body.
	Redirect	*Redirect(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) ≠ typeOf(m2)* ∧ *signature(m1) = signature(m2)*	To request that another object perform a tightly related subtask to the task at hand, perhaps performing the basic work.
	Delegate	*Delegate(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) ≠ typeOf(m2)* ∧ *signature(m1) ≠ signature(m2)*	To parcel out, or delegate, a portion of the current work to another method in another object.

Table 3.5 *– The first six Method Call EDPs definitions*

EDP category	EDP name	Definition	Explanation [Smi02]
Method Call	Redirected recursion	*Redirected recursion(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) = signature(m2)*	To perform a recursive method, but one that requires interacting with multiple objects of the same type.
	Delegated conglomeration	*Delegated conglomeration(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *typeOf(m1) = typeOf(m2)* ∧ *signature(m1) ≠ signature(m2)*	A Conglomeration pattern is appropriate, but we need to work with a distinct instance of our object type, resulting in a need for the Delegate pattern to be used.
	Redirect in family	*Redirect in family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) = signature(m2)*	Redirect some portion of a method's implementation to a possible cluster of classes, of which the current class is a member.
	Delegate in family	*Delegate in family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ *Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) ≠ signature(m2)*	Related classes are often defined as such to perform tasks collectively. In such cases, multiple objects of related types can interact in generalized ways to delegate tasks to one another.
	Redirect in limited family	*Redirect in limited family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ ∃ *T*: *ancestor(typeOf(m1), T)* ∧ *ancestor(typeOf(m2), T)* ∧ *¬Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) = signature(m2)*	When Redirect in family is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.
	Delegate in limited family	*Delegate in limited family(m1, m2)* iff ∃ *method invocation(m1, m2)*: *Source(method invocation(m1, m2)) ≠ target(method invocation(m1, m2)* ∧ ∃ *T*: *ancestor(typeOf(m1), T)* ∧ *ancestor(typeOf(m2), T)* ∧ *¬Ancestor(typeOf(m1), typeOf(m2))* ∧ *signature(m1) ≠ signature(m2)*	When Delegate in family is too generalized, and it is necessary to pre-select a sub-tree of the class hierarchy for polymorphism.

Table 3.6 *– The second six Method Call EDPs definitions*

3.2.3. Design pattern clues definitions

DP clues category	DP clue name	Definition	Explanation [MATZ09]
Class Declaration Information	Interface and class inherited	*Interface and class inherited(C1, I, C2)* iff *interface inherited(C1, I)* ∧ *class inherited(C1, C2)*	A class implements an interface and extends a class, providing therefore a mechanism to simulate multiple inheritance languages not supporting it.
	Multiple interfaces inherited	*Multiple interfaces inherited(C, I1, …, In)* iff *interface inherited(C, Ii)*, ∀ *i* = 1, …, *n*	A class implements *n* interfaces, with *n* > 1.
	Object structure child	*Object structure child(C, T)* iff (*interface(T)* ∨ *abstract class(T)*) ∧ *ancestor(C, T)*	The class belongs to an object structure, i.e. it has at least an ancestor which is either an interface or an abstract class.
	Template implementor	*Template implementor(C1, C2)* iff *template method(C2)* ∧ *class inherited(C1, C2)*	A class extends another class implementing a Template method.
Multiple Classes Information	Façade method	*Façade method(m, m1, …, mn)* iff *body(m)* contains (*object creation(typeOf(m), oi), typeOf(oi) = typeOf(mi)*, ∀ *i* = 1, …, *n*) ∨ (*method invocation(m, mi)* ∧ ∀ *i* = 1, …, *n typeOf(m)* ≠ *typeOf(mi)*)	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.
	Proxy class	*Proxy class(C1, C2, T)* iff (((*interface(T)* ∧ *interface inherited(C1, T)* ∧ *interface inherited(C2, T)*) ∨ (*abstract class(T)* ∧ *class inherited(C1, T)* ∧ *class inherited(C2, T)*)) ∧ ∃ *o* = *inst(C1)* ∈ *attributes(C1)*: *typeOf(o)* = *C2*	A class implements an interface or extends an (abstract) class, referring to a class that implements the same interface or extends the same (abstract) class.

Table 3.7 *– Class Declaration and Multiple Classes Information design pattern clues definitions*

DP clues category	DP clue name	Definition	Explanation [MATZ09]
Instance Information	Controlled self instantiation	*Controlled self instantiation(C, o)* iff ∃ *m* ∈ *methods(C)*, ∃ *controlStatement* ∈ *body(m)*: *object creation(C, o)* ∈ *controlStatement* ∧ *same class object(C, o)* ∧ *o* ∈ *attributes(C)*	The instantiation of an object occurs inside an if (or a switch) block, therefore under a condition.
	Private self instance	*Private self instance(C, o)* iff *private object(o)* ∧ *same class object(C, o)* ∧ *o* ∈ *attributes(C)*	A class has a private instance of itself. Access to this instance can occur only from within the same class.
	Static self instance	*Static self instance(C, o)* iff *static object(o)* ∧ *same class object(C, o)* ∧ *o* ∈ *attributes(C)*	A class has a static instance of itself. Therefore this instance is unique inside the system.
	Single self instance	*Single self instance(C, o)* iff ∃! *o* ∈ *attributes(C)*: *same class object(C, o)*	A class maintains a unique instance of itself, no matter it is static or not.
	Instance in abstract class	*Instance in abstract class(C, o)* iff *abstract class(C)* ∧ *o* ∈ *attributes(C)* ∧ *different class object(C, o)*	An abstract class has a reference to another class.
	Reference to abstract class	*Reference to abstract class(C, o)* iff *o* ∈ *attributes(C)* ∧ *abstract class(typeOf(o))*	A class attribute is a reference to an abstract class.
	Same interface instance	*Same interface instance(C, o)* iff *o* ∈ *attributes(C)* ∧ *different class object(C, o)* ∧ *same hierarchy object(C, o)*	A class contains a reference to an object whose type is compatible with the same interface of the declaring class.
	Same interface container	*Same interface container(C, list(o1, …, on))* iff *list(o1, …, on)* ∈ *attributes(C)* ∧ ∀ *oi* ∈ *list(o1, …, on)*, *different class object(C, oi)* ∧ *same hierarchy object(C, oi)*	A class contains a set or a list of elements that are compatible with the same interface of the declaring class.

Table 3.8 *– Instance Information design pattern clues definitions*

DP clues category	DP clue name	Definition	Explanation [MATZ09]
Method Signature Information	Factory parameter	*Factory parameter(p)* iff ∃ *m* ∈ *methods(C)*: *p* ∈ *param(m)* ∧ *concrete product getter(typeOf(p))*	A method of a certain class receives as an input parameter an object that belongs to a class defining some Concrete product getter methods.
	Protected instantiation	*Protected instantiation(C)* iff ∀ constructor *constr* ∈ *methods(C)*, *private constructor(constr)*	All the constructors within a given class are declared private.
	Adapter method	*Adapter method(m, C, T1, T2)* iff (*m* ∈ *methods(C)* ∧ *interface method(m, T1)* ∧ *ancestor(C, T2)* ∧ ∃ *m2* ∈ *methods(T2)*: *delegate(m, m2)*) ∨ (*m* ∈ *methods(C)* ∧ *overriding method(m, C, T1)* ∧ *body(m)* contains *methodInvocation(m, m')*: *typeOf(m')* = *T2* ∧ *different hierarchy object(typeOf(m), target(methodInvocation(m, m')))*)	Two types of Adapter method exist. It can be a method which is an implementation of an interface method calling a method belonging to the parent class. Or it can be an overridden method from the parent which calls a method belonging to a class that does not share common parents with the adapter method declaring class.
	Interface method implemented	*Interface method implemented(m, C, I)* iff *interface inherited(C, I)* ∧ *interface method(m, I)* ∧ *m* ∈ *methods(C)*	A class implements a method declared inside an interface.
	Overriding method	*Overriding method(m1, m2)* iff ∃ *C1, C2*: *ancestor(C1, C2)* ∧ *m1* ∈ *methods(C1)* ∧ *m2* ∈ *methods(C2)* ∧ *m1* = *m2*	A class overrides, i.e. redefines a method belonging to its superclass.
	Component method	*Component method(m, C)* iff ∃ *p* ∈ *param(m)*: *same class object(C, p)*	A class declares a method that takes an object of the same class as its single parameter.
	Cross relationship	*Cross relationship(C1, C2)* iff ∃ *m1* ∈ *methods(C1)*: ∃ *p2* ∈ *param(m1)*: *typeOf(p2)* = *C2* ∧ ∃ *m2* ∈ *methods(C2)*: ∃ *p1* ∈ *param(m2)*: *typeOf(p1)* = *C1*	Given two classes C1 and C2, C1 declares a method which accepts a reference to C2 as one of its parameters, vice versa C2 declares a method which accepts a reference to C1 as one of its parameters.
	Abstract cyclic call	*Abstract cyclic call(m1, m2)* ∃ *loop* ∈ *m1*: *loop* contains *abstract method invoked(m1, m2)*	A method invokes an abstract method within a cycle.
	Factory method	*Factory method(m, C1, T)* iff *m* ∈ *methods(C1)* ∧ *body(m)* contains *object creation(C1, o)*: *typeOf(o)* = *T* ∧ (∃ *mt* ∈ *methods(T)*: *overriding method(m, mt)* ∨ (*interface(T)* ∧ *interface method(m, T)*))	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.

Table 3.9 *– Method Signature Information design pattern clues definitions*

DP clues category	DP clue name	Definition	Explanation [MATZ09]
Method Body Information	Instance in abstract referred	*Instance in abstract referred(C, o)* iff *instance in abstract class(C, o)* ∧ ∃ *m* ∈ *methods(C)*, ∃ *mo* ∈ *methods(typeOf(o))*: *method invocation(m, mo)*	A method of a class implementing Instance in abstract class invokes a method on the declared instance.
	Multiple redirections in family	*Multiple redirections in family(m1, m2)* iff *redirect in family(m1, m2)* ∧ *body(m1)* contains *loop*: *redirect in family(m1, m2)*∈ *loop*	A method contains a Redirect in family method invocation EDP that is contained within a cycle.
	Proxy method invoked	*Proxy method invoked(m1, m2)* iff *Proxy class(C1, C2, T), m1* ∈ *methods(C1)* ∧ *m2* ∈ *methods(C2)* ∧ *redirect in limited family(m1, m2)*	A proxy class invokes a method on the referred subject by a Rediriect in limited family method call EDP.
	Template method	*Template method(C1)* iff ∃ *T*, ∃ *m1* ∈ *methods(C1)* ∧ *m2*∈ *methods(T)*: *abstract method invoked(m1, m2)*	A method calls at least an abstract method within its body.
Method Set Information	All methods invoked	*All methods invoked(C1, C2)* iff ∀ *m2* ∈ *methods(C2)* ∃ *methodInvocation(m, m2)*: *m* ∈ *C1*	A class invokes all of the public methods declared in a target class.
	Leaf class	*Leaf class(C1, C2)* iff *ancestor(C1, C2)* ∧ ∀ *mi* ∈ *methods(C2)*: *component method(mi, C2)* ¬∃ *m′* ∈ *methods(C1)*: *overriding method(m′, mi)*	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, or giving an empty implementation for such methods.
	Node class	*Node class(C1, C2)* iff *ancestor(C1, C2)* ∧ ∃ *m* ∈ *methods(C2)*: *component method(m, C2)* ∃ *m′* ∈ *methods(C1)*: *overriding method(m′, m)*	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.

Table 3.10 *– Method Body and Method Set Information design pattern clues definitions*

DP clues category	DP clue name	Definition	Explanation [MATZ09]
Return Information	Concrete product getter	*Concrete product getter(C)* iff ∃ *m* ∈ *methods(C)*: *different object returned(m)*	A class declares one or more methods that return objects belonging to some other classes.
	Concrete product returned	*Concrete product returned(m)* iff *same hierarchy object returned(m)* ∧ (*different object returned(m)* ∨ *same object returned(m)*)	A method returns objects that belong to subclasses extending the class that represents the declared method return type.
	Abstract product returned	*Abstract product returned(m)* iff *abstract class(typeOf(returnedValue(m)))*	A method returns a reference to an abstract class.
	Parent product returned	*Parent product returned(m)* iff *m* ∈ *methods(T1)* ∧ ∃ *T2*: (*class inherited(T1, T2)* ∨ *interface inherited(T1, T2)*) ∧ *typeOf(returnedValue(m)) = T2*	A method returns a reference to the parent class of its declaring class.
	Empty concrete product getter	*Empty concrete product getter(m)* iff *different object returned(m)* ∧ *body(m)* is empty	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).
	Empty method	*Empty method(m)* iff *simple type returned(m)* ∧ *body(m)* is empty	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).
	Multiple returns	*Multiple returns(m)* iff |*returnStatements(m)*| > 1	A method provides several possible return points.
	Void return	*Void return(m)* iff *void returned(m)* ∧ *body(m)* contains *object creation(typeOf(m), o), typeOf(o) ≠ typeOf(m)*	A class defines a method that instantiates an object without returning it.
	Cross hierarchy return	*Cross hierarchy returned(m)* iff *different hierarchy object returned(m)*	A method returns an object of a class belonging to a different hierarchy.
Java Information	Cloneable implemented	*Cloneable implemented(T)* iff *interface inherited(T, I)* ∧ *name(I)* = `java.lang.Cloneable`	A class implements the `java.lang.Cloneable` interface.
	Prototyping constructor	*Prototyping constructor(m, C)* iff *constructor(m, C)* ∃ *p* ∈ *param(m)*: *cloneable implemented(typeOf(p))*	A method defines a constructor which receives objects that can be cloned, that is that belong to classes implementing the `java.lang.Cloneable` interface.

Table 3.11 *– Return and Java Information design pattern clues definitions*

3.2.4. Micro patterns definitions

Micro pattern category	Micro pattern name	Definition	Explanation [GM05]
Degenerate State and Behaviour	Designator	*Designator(I)* iff $\|$*attributes(I)*$\| = 0 \wedge \|$*methods(I)*$\| = 0$	An interface with absolutely no members.
	Taxonomy	*Taxonomy(I)* iff $\|$ *attributes (I)*$\| = 0 \wedge \|$*methods(I)*$\| = 0 \wedge \exists$ *I1*: *interface inherited(I, I1)*	An empty interface extending another interface.
	Joiner	*Joiner(I)* iff $\|$*attributes (I)*$\| = 0 \wedge \|$*methods(I)*$\| = 0 \wedge \exists$ *I1, I2*: *interface inherited(I, I1)* $\wedge$ *interface inherited(I, I2)*	An empty interface joining two or more superinterfaces.
	Pool	*Pool(C)* iff $\|$*methods(C)*$\| = 0 \wedge \forall$ *a* $\in$ *attributes (C) a* is static	A class which declares only static final fields, but no methods.
Degenerate Behaviour	Function pointer	*Function pointer(C)* iff $\|$ *attributes(C)*$\| = 0 \wedge$ $\|$*methods(C)*$\| = 1 \wedge \exists$ *m* $\in$ *methods(C)*: *m* is public	A class with a single public instance method, but with no fields.
	Function object	*Function pointer(C)* iff $\|$ *attributes(C)*$\| \geq 1 \wedge$ $\|$*methods(C)*$\| = 1 \wedge \exists$ *m* $\in$ *methods(C)*: *m* is public	A class with a single public instance method, and at least one instance field.
	Cobol like	*Cobol like(C)* iff $\forall$ *a* $\in$ *attributes(C) a* is static $\wedge$ $\|$*methods(C)*$\| = 1 \wedge \exists$ *m* $\in$ *methods(C)*: *m* is static	A class with a single static method, but no instance members.
Degenerate State	Stateless	*Stateless(C)* iff $\forall$ *a* $\in$ *attributes(C) a* is static $\wedge$ *a* is final	A class with no fields, other than static final ones.
	Common state	*Stateless(C)* iff $\forall$ *a* $\in$ *attributes(C) a* is static	A class in which all fields are static.
	Immutable	*Immutable(C)* iff $\forall$ *a* $\in$ *attributes(C)* $\exists!$ *assignment(a, value)* $\wedge \exists$ *constructor(m)* $\in$ *methods(C)*: *assignment(a, value)* $\in$ *body(constructor(m))*	A class with several instance fields, which are assigned exactly once, during instance construction.
Controlled Creation	Restricted creation	*Restricted creation(C)* iff ($\forall$ *constructor(m)* $\in$ *methods(C), constructor(m)* is private $\vee$ *constructor(m)* is protected) $\wedge$ $\exists$ *a* $\in$ *attributes(C)*: *a* is static $\wedge$ *typeOf(a) = C*	A class with no public onstructors, and at least one static field of the same type as the class.
	Sampler	*Sampler(C)* iff $\exists$ *constructor(m)* $\in$ *methods(C)*: *constructor(m)* is public $\wedge \exists$ *a* $\in$ *attributes(C)*: *a* is static $\wedge$ *typeOf(a) = C*	A class with one or more public constructors, and at least one static field of the same type as the class.

Table 3.12 *– The first set of micro patterns definitions*

Micro pattern category	Micro pattern name	Definition	Explanation [GM05]
Wrappers	Box	*Box(C)* iff $\|attributes(C)\| = 1 \wedge \exists\, m \in methods(C): \exists\, assignment(a, value) \in body(m), a \in attributes(C)$	A class which has exactly one, mutable, instance field.
	Compound box	*Compound box(C)* iff $\exists\, a \in attributes(C): typeOf(a) = C \vee typeOf(a) = T \neq C \vee typeOf(a) = list(a1, \ldots, an)$	A class with exactly one non primitive instance field.
	Canopy	*Canopy(C)* iff $\|attributes(C)\| = 1 \wedge a \in attributes(C) \wedge \forall\, assignment(a, value), assignment(a, value) \in constructor(m) \in methods(C)$	A class with exactly one instance field that it assigned exactly once, during instance construction.
Data Managers	Record	*Record(C)* iff $\|methods(C)\| = 0 \wedge \forall\, a \in attributes(C)$ *a* is public	A class in which all fields are public, no declared methods.
	Data manager	*Data manager(C)* iff $\forall\, m \in methods(C), setter(m, a) \vee getter(m, a)$	A class where all methods are either setters or getters.
	Sink	*Sink(C)* iff $\forall\, m1 \in methods(C)\ \neg\exists\, method\ invocation(m1, m2): typeOf(m2) \neq C$	A class whose methods do not propagate calls to any other class.
Base Classes	Outline	*Outline(C)* iff $\exists\, m1 \in methods(C): \exists\, method\ invocation(m1, m2)$: *m2* is abstract	A class where at least a method invokes an abstract method belonging to the same class.
	Trait	*Trait(C)* iff $abstract\ class(C) \wedge \|attributes(C)\| = 0 \wedge \exists\, m \in methods(C): abstract\ method(m)$	An abstract class which has no state.
	State machine	*State machine(I)* iff $\forall\, m \in methods(I)\ \|params(m)\| = 0$	An interface whose methods accept no parameters.
	Pure type	*Pure type(C)* iff $\forall\, m \in methods(C)\ abstract\ method(m) \wedge \neg\exists\, m \in methods(C)$: *m* is static $\wedge\ \|attributes(C)\| = 0$	A class with only abstract methods, and no static members, and no fields.
	Augmented type	*Augmented type(C)* iff $\forall\, m \in methods(C)\ abstract\ method(m) \wedge \neg\exists\, m \in methods(C)$: *m* is static $\wedge\ \|attributes(C)\| \geq 3 \wedge \forall\, a \in attributes(C)$ *a* is static $\wedge$ *a* is final	A class with only abstract methods and three or more static final fields of the same type.
	Pseudo class	*Pseudo class(C)* iff $abstract\ class(C) \wedge \forall\, m \in methods(C), abstract\ method(m) \wedge \forall\, a \in params(C)$ *a* is static	A class which can be rewritten as an interface: no concrete methods, only static fields.
Inheritors	Implementor	*Implementor(C)* iff $\forall\, m1 \in methods(C)\ \exists\, m2: overriding\ method(m1, m2) \wedge (abstract\ method(m2) \vee interface\ method(m2))$	A concrete class, where all the methods override inherited abstract methods.
	Overrider	*Overrider(C)* iff $\forall\, m1 \in methods(C)\ \exists\, m2: overriding\ method(m1, m2) \wedge \neg(abstract\ method(m2) \vee interface\ method(m2))$	A class in which all methods override inherited non-abstract methods.
	Extender	*Extender(C1)* iff $\exists\, C2: class\ inherited(C, C2) \wedge \neg\exists\, m1 \in methods(C1), m2 \in methods(C2): overriding\ method(m1, m2)$	A class which extends the inherited protocol, without overriding any methods.

Table 3.13 *– The second set of micro patterns definitions*

3.3. The Micro-structures detector

We have developed a plug-in for the Eclipse platform [Eclipse], called *Micro-structures detector*, which is part of the MARPLE project [ATZM08] and is devoted to the identification of EDPs, clues and micro patterns inside subject systems. Figure 3.1 depicts the plug-in architecture.

As it can be noticed, the Micro-structures detector is laid on the functionalities provided by the Eclipse framework. In order to be analyzed, the source code of the subject system needs to be translated into an Abstract Syntax Tree (AST) representation. The tree data structure is inspected by visitors [GHJV94], which have the aim to detect realizations of micro-patterns inside the system. For each single micro-structure, a visitor has been implemented. The AST representation and the basic classes to implement the visitor functionalities are provided by the Eclipse Java DOM/AST library, which contains those classes that model the source code of a Java program as a structured document.

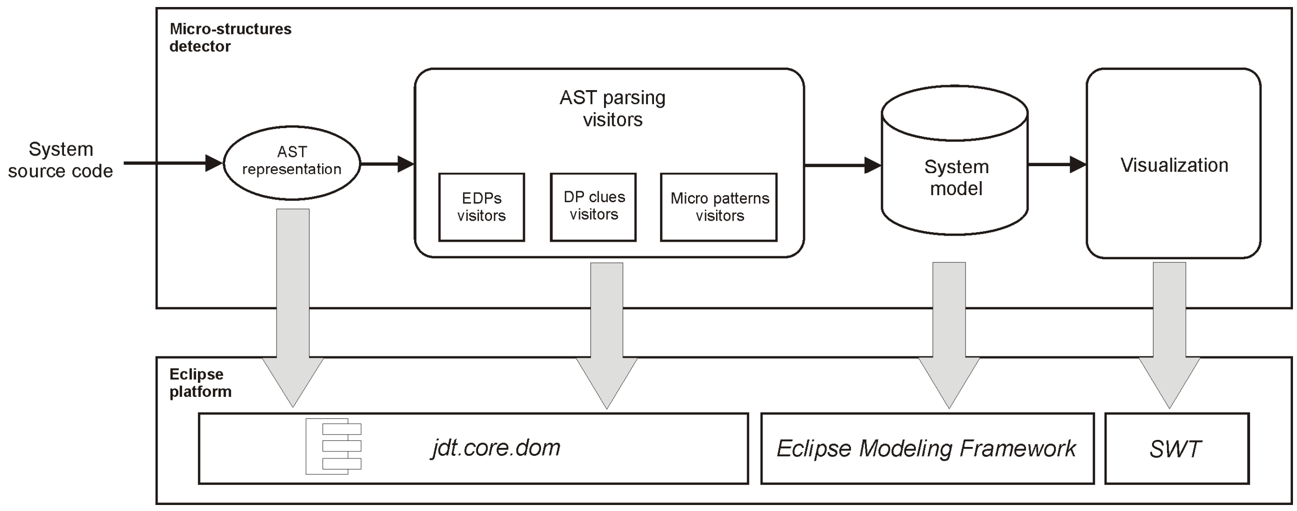


Figure 3.2 *– The architecture of the Micro-structures detector*

The micro-structures instances detected by the visitors are then stored in a model [ATZ+09], built on top of the Eclipse Modeling Framework (EMF), which organically represents all the system classes and interfaces, reporting for each of them the micro-structures that have been detected within it. The model is generated in order to allow an easy recovery of the stored information that is to be used for the DPD and SAR activities.

Finally, the micro-structures instances can be shown to the user. The visualization module is laid on the Eclipse Standard Widget Toolkit (SWT), and basically provides a comfortable tree view in which all the micro-structures instances are collected according to their categories.

3.4. Concluding remarks

In this chapter we have proposed a new definition of the considered micro-structures. The redefinition is based on the use of concepts that are common to each category of micro-structures, named code atoms. Through the use of atoms we succeeded in grouping and giving a common interpretation to structures that are to be considered in some sense similar. In fact, they can all be detected by the source code of a subject system by analyzing the characteristics of the classes, attributes and methods composing it, and they can all be defined through the atoms introduced in section 3.2.1. The principal aim of this redefinition is giving a more formal definition to those micro-structures that may result ambiguous. Just as an example, the Data manager micro pattern is now clearer, as the concepts of getter and setter methods are now treated as code atoms and have been more strictly specified. This more strict definition allows for a more precise identification of these structures inside a subject system.

Chapter 4

Micro-structures for design pattern detection

Abstract

This chapter is devoted to the comparison of the considered micro-structures and to the analysis of their usefulness for design pattern detection activities. The comparison will consider six aspects: the objectives of each category of micro-structures, how they have been defined, their detail level, the source for their detection from the analyzed systems, their categorization, and their eventual interdependence. The micro-structures will be detected from sample design pattern implementations, and their relevance for both pattern structure and pattern role detection will be investigated. Finally, for each design pattern a set of possible hints for the identification of its characteristics will be suggested.

4.1. Micro-structures for design pattern detection

In the introduction we have discussed about the relevance of design pattern detection for reverse engineering activities, that is strongly apparent in the scientific literature, as we have outlined in the related works. Many research groups have proposed several approaches and tools for design pattern detection with different results. A promising approach for design pattern recognition consists in the search for different types of elements, that we generally call micro-structures, which resembled can give strong indications for the presence of design patterns.
In Chapter 3 we have introduced EDPs, DP clues and micro patterns, and provided a definition for each single micro-structure. We now aim to explain which are the similarities and differences among these sub-components, how these sub-components can be described and exploited especially in the reverse engineering process and also as object-oriented best practices. We also want to explore which are the components design patterns can be decomposed in. This is fundamental to better understand what design patterns effectively are and how can they be described in order to be efficiently exploited both in forward and reverse engineering contexts.

4.2. An analysis of micro-structures based on six aspects

The main goal of this chapter is to describe the similarities and differences among the categories of micro-structures we take into account. To achieve this goal we have identified six aspects that we consider relevant for the comparison. Hence, before performing the comparison we present the micro-structures introduced in Chapter 3 according to these aspects.
The six aspects are:

- *Objectives*: why a particular type of micro-structure has been introduced?
- *Definition technique*: how is each category identified and consequently defined?
- *Detail level*: which is the particular code structure which each single micro-structure relates to?
- *Source for the detection*: where the micro-structures are identified from? Are they detected from the source code of the subject system, or from its byte code, or from other forms of code representations?
- *Type of subdivision*: how are the elements of a particular category grouped?
- *Self dependence*: do the micro-structures belong to a certain category independently from each other? Or else is there any element in a category whose presence is strictly related to the presence of some other elements?

The rest of this chapter describes the three categories of micro-structures through these six aspects. This presentation is fundamental for the purpose of the comparison because it enables us to focus on those features which are relevant for their definition and detection, and which are not always clearly presented while focusing on a single type of micro-structures. Discussing about each of these aspects makes the comparison among micro-structures straightforward.

4.2.1. Elemental design patterns

Objectives

EDPs aim to propose solutions to programming issues that are faced in the everyday practice, and that are also exploited in the design pattern detection process, as described in the SPQR approach [SS03].

Definition technique

EDPs were identified by analyzing the design patterns proposed in [GHJV94]. Primarily, eight key concepts were identified for the entire set of design patterns. From these first common concepts, a search for further elements and a better specification of the existing ones were started leading to the definition of the current 16 EDPs.

Detail level

EDPs represent details regarding instances creation, extensions of classes and implementations of interfaces, and method invocations. They do not provide any information about a whole class or about a set of classes.

Source for the detection

In the SPQR approach, EDPs are identified by an inspection of a POML (*Pattern Object Markup Language*) file [SS03], an XML data format representation of the analyzed source code, thus their detection is language-independent as far as translation tools from programming languages to the POML format are implemented. With the Micro-structures detector, EDPs are identified by analyzing AST representations of the source code.

Type of subdivision

EDPs are divided into three main disjoint categories: Object Elements, Method Call, Type Relation.

Self dependence

EDPs are completely independent from each other. The identification of each of them does not rely on the presence of other EDPs inside the code.

4.2.2. Design pattern clues

Objectives

Design pattern clues have been proposed and defined only in the perspective of design pattern detection and have been introduced to capture further information not expressed by EDPs and more related to the identification of design pattern roles characteristics.

Definition technique

The definition of clues derived by the analysis of design pattern architectures. Their specification and definition was improved by the analysis of several possible variants of design patterns (for example, those proposed in [Coo98, GHJV94]). These variants were implemented in Java and were manually analyzed searching for particular programming details which may provide indications on the presence of a design pattern instance, even according to the preliminary hints defined only on the design pattern structures.

Detail level

Design pattern clues provide information related to single code structures, but at different detail levels (e.g. classes, methods, variables). Each clue belongs to a single class, method or any other code structure, thus given a clue instance a corresponding code structure can be associated to it. However, the identification of some of the clues can depend on characteristics which are found in couples of classes, but this remark does not preclude the belonging of a clue instance to a single structure.

Source for the detection
All the clues can be automatically detected from the source code, as they are representations of implementation issues which can be easily understood through an analysis of the source code. Clues are identified by parsing AST representations of the analyzed Java project. A set of visitors in the Micro-structures detector, one for each clue, is responsible for the traversal of the trees to catch instances of these elements.

Type of subdivision
Design pattern clues are subdivided into eight categories which focus on low level information related to implementation details such as class definition, object instantiation, variables features, methods signature, return information and Java specific features.

Self dependence
Some clues rely on the presence of other clues previously identified inside source code. For example (see Chapter 3) the Factory parameter clue can occur only when a method input parameter is an instance of a factory class, meaning that the factory class must declare methods which are instances of the Concrete product getter clue. Therefore, we can assert that the factory parameter clue can exist only in dependence of the presence of concrete product getter instances in another class.

4.2.3. Micro patterns

Objectives
The objective of micro patterns is the identification of common programming techniques in general.

Definition technique
Micro patterns were identified with a manual inspection of code with further refinements during their definition.

Detail level
Micro patterns are defined at a class level. Each of them depicts a characteristic which can be identified inside a class. In particular, each of them establishes constraints about the characteristics of methods and/or attributes of a class.

Source for the detection
Micro patterns are the only category of micro-structures that are identified starting from Java byte code analysis. Actually it seems sensible to assert that the technique can also be based on the inspection of ASTs, as the definition and the characteristics of micro patterns can let them be discovered by a static analysis of such structures. The Micro-structures detector can identify the whole set of micro patterns using this kind of analysis.

Type of subdivision

The 27 micro patterns have been subdivided into eight categories. These categories focus on degenerated classes (i.e., classes with no members or empty interfaces), containment aspects (i.e., classes with all public fields or exactly one non primitive instance field), and inheritance related features (i.e., classes where all methods override inherited abstract methods or classes in which all methods override inherited non-abstract methods). These categories, differently from all other micro-structures, are overlapped, thus some elements may belong to two categories.

Self dependence

No dependencies among micro patterns have been identified. The fact that some micro patterns can contemporarily belong to more than one category does not imply that some of them exist only if others are identified.

4.3. A comparison among micro-structures

The comparison among the micro-structures is performed based on the description we have just presented. A summary of the aspects we have considered for the four micro-structures is shown in Table 4.1. The rows correspond to the micro-structures, while the columns to the six aspects considered for their analysis. This table provides a comparison among micro-structures.

As far as the *objective* is concerned, EDPs have more objectives, closer to the objectives of design patterns and aim to represent both common design and programming techniques.

Regarding the *definition technique* aspect, clues and micro patterns have been defined starting from source code, but considering different detail levels. Each micro pattern depicts a characteristic of a single class [GM05], and some of them can be decomposed in complementary clues.

We noticed that EDPs and clues provide different types of information which can be successfully used together for design pattern detection. The joint exploitation of these two categories enables us to understand, for example, which are the relationships pointed out by EDPs among the classes implementing some clues. We can generally assert that if clues help us in understanding which roles a certain class can play inside a design pattern, EDPs let us understand which kind of relationships these roles may have with the other potential roles identified by clues. In this way, EDPs may identify the structures of design patterns, rather than their single roles, which are supported by design pattern clues detection.

As far as the *detail level* is concerned, clues represent generally fine-grained detail levels regarding instance creations, class extensions, and method invocations, and also provide information on single classes and on set of classes. Micro patterns are all defined only at class level, each of them describing a peculiar characteristic of a single class.

Considering *the type of subdivision* aspect, only micro patterns can belong to more than one category, but we cannot anyway assert that the presence of some of them strictly depends on the existence of some other micro pattern.
Finally, as far as *self dependence* is concerned, clues collect dependent elements: the existence of certain clues is possible only if some other clues have been previously identified. EDPs and micro patterns are not self-dependent, i.e. they can be detected independently from one another.

Category	**Objectives**			**Definition technique**		**Detail level**					**Source for the detection**			**Type of subdivision**		**Self dependence**	
	DP recogntion	Similar to DP	Programming practices description	From source code	Identification of concepts common to various DPs	Set of classes	Single class	Method	Instantiation	Inheritance	Abstract Syntax Tree	Byte code	XML data format representation	Disjoint categories	Overlapping categories	Independent elements	Dependent elements
DP clues	✓			✓		✓	✓	✓	✓	✓	✓			✓			✓
EDPs	✓	✓	✓		✓			✓	✓	✓	✓		✓	✓		✓	
Micro patterns			✓	✓			✓					✓			✓	✓	

***Table 4.2** – Micro-structures revisited according to six core aspects*

4.4. The role of micro-structures in the detection of design patterns

At now we have considered micro-structures as they are. Micro-structures are strictly related to source code and are useful and exploitable in a design pattern detection process. Their exploitation and usefulness is to be validated on three steps:

- Micro-structures detection in sample design pattern implementations;
- Micro-structures relevance evaluation for design pattern detection;
- Micro-structures detection in design pattern implementation variants.

We discuss these three steps on six sample patterns, two for each category. For space reasons, we cannot provide examples for the detection of all the micro-structures for all the design patterns.

4.4.1. Micro-structures detection in sample design pattern implementations

In order to emphasize the role micro-structures have for design pattern detection, in this section we provide six examples of different design patterns implementations (two patterns for each of the three categories: creational, behavioural and structural), and we describe the micro-structures that can be identified within them. The detailed descriptions of all the micro structures can be found in Chapter 3 as well as in the references provided in the same chapter.

1) Micro-structures detection in the Singleton design pattern

We first focus our attention on the Singleton design pattern: even if its structure is very simple, we can make interesting considerations on it because its presence is not trivial to identify.

In [GHJV94] the following implementation of the Singleton creational design pattern is proposed:

```
public class Singleton {
  private static Singleton instance;

  private Singleton(){}

  public static Singleton instance(){
        if (instance == null)
             instance = new Singleton();
        return instance;
   }
}
```

The `Singleton()` constructor is private, so that it cannot be accessed by external classes. In this way, external classes may access the Singleton class only through the static method `instance()`. This method returns an instance of the Singleton class, represented by the static object instance, which is granted to be unique as it is created if and only if it is null. Otherwise, the method returns the already existing instance.

Design pattern clues

In this implementation of the Singleton design pattern six different design pattern clues are detected:

- *Protected instantiation*: the constructors of the class `Singleton` are all declared private (see the `Singleton()` constructor). This means that the class cannot be instantiated by external classes, avoiding the creation of un-controlled instances;
- *Private self instance*: the class maintains a private instance of itself (see the `instance` object declaration). This instance cannot be directly accessed by external classes;
- *Static self instance*: the class maintains a static instance of itself (see the `instance` object declaration). This instance is therefore unique in the system;

- *Single self instance*: the class maintains only one instance of itself (see the `instance` object declaration); this hint is useful while dealing with self instances that are not declared static, and therefore may be more than one inside the system;
- *Controlled self instantiation*: the creation of an object of a certain class is under control of an if or switch statement (see the if statement in the `instance()` method);
- *Concrete product getter*: a method returns a concrete object of the identical type declared by the method return type (see the `return instance` statement of the `instance()` method).

As we can observe, even by the analysis of the above few lines of code useful information can be derived. The presence of one of these elements or their combination is a strong indicator of the presence of Singleton design pattern, and will be further discussed in the next subsection.

Elemental design patterns

In this implementation of the Singleton only one EDP can be found: *Create object*, represented by the "`instance = new Singleton()`" instruction. This EDP does not capture the basic characteristic of the Singleton design pattern, i.e. the uniqueness of its instance during execution. Therefore we cannot detect this design pattern relying only on the information provided by the presence of the Create object EDP.

Micro patterns

Analyzing this particular implementation, five micro patterns can be detected:

- *Function object*: the class has a single public instance method, and at least one instance field (see the `instance()` method and the `instance` field);
- *Common state*: all the fields belonging to the class are declared `static` (see the `instance` field);
- *Restricted creation*: the class does not have any public constructor, and have at least one static field of the same type as the class (see the `Singleton()` constructor and the `instance` field);
- *Data manager*: the methods declared by the class are either setters or getters (see the `instance()` method);
- *Sink*: the methods of the class do not propagate calls to any other class (see the `instance()` method).

The most peculiar micro pattern for the detection of the Singleton design pattern is Restricted creation. This micro pattern can be seen as the combination of the Static self instance and of the Controlled self instantiation clues. For this reason, we can assert that clues are more detailed than micro patterns: analyzing the micro patterns catalogue [GM05], it is possible to see that each of these elements is related to characteristics belonging to a single class. On the other hand, clues collect characteristics that are positioned at different detail levels (for example, Static self instance is related to a single instance, Controlled self instantiation regards a method implementation).

2) Micro-structures detection in the Abstract factory design pattern

We now propose another example of micro-structures detection inside another broadly used design pattern, that surely has a more complex structure with respect to the Singleton design pattern previously analyzed: the Abstract factory.

Starting from the class diagram provided by [GHJV94], we consider the following basic implementation of the Abstract factory design pattern. Each concrete implementation should be however constrained to the concepts and relationships that characterize this basic implementation.

```
public abstract class AbstractFactory {
   public abstract AbstractProductA createProductA();
   public abstract AbstractProductB createProductB();
}

public class ConcreteFactory1 extends AbstractFactory {
   public AbstractProductA createProductA(){
        return new ProductA1(); }
   public AbstractProductB createProductB(){
        return new ProductB1(); }
}

public class ConcreteFactory2 extends AbstractFactory {
   public AbstractProductA createProductA(){
        return new ProductA2(); }
   public AbstractProductB createProductB(){
        return new ProductB2(); }
}

public abstract class AbstractProductA {}
public class ProductA1 extends AbstractProductA {}
public class ProductA2 extends AbstractProductA {}

public abstract class AbstractProductB {}
public class ProductB1 extends AbstractProductB {}
public class ProductB2 extends AbstractProductB {}
```

Design pattern clues

The design pattern clues that can be identified in this implementation are the following:

- *Concrete product returned*: each of the four getter methods implemented in the concrete factories returns an object which belongs to a subclass extending the class that represents the declared method return type; for example, the method `ConcreteFactory1.createProductA()` returns an instance of the `ProductA1` class, which extends the declared return type `AbstractProductA`.
- *Abstract product returned*: the two abstract methods declared by the abstract factory both return a reference to an abstract class, identifying the connection between the factory and the abstract products;
- *Parent product returned*: the two references returned by the concrete factory methods are both related to classes that are parent classes inside a certain hierarchy;

Elemental design patterns

In this basic implementation of the Abstract factory design pattern we can find the following EDPs:

- *Create object*: the two concrete factories both contain two Create object EDPs, when returning instances of the concrete products related to them;
- *Abstract interface*: the methods declared by the abstract factory are examples of the Abstract interface EDP, which represents abstract methods or methods belonging to an interface;
- *Inheritance*: in this example we can see three inheritance hierarchies; the first one involving the abstract factory as the parent class and the two concrete factories as its children; the second and the third ones regard the two abstract products and their concrete sub-classes. We have therefore a total of six inheritance EDP instances, one for each child class.

These EDPs don't let us assert that this implementation is an instance of the Abstract factory design pattern. The real problem is enclosed in the nature of EDPs, as they don't seem to capture the rationale behind each design pattern.

Micro patterns

We can find six micro patterns in this implementation of the Abstract factory design pattern:

- *Data manager*: a class implements only getter or setter methods, according to the meaning of getter and setter methods agreed in Chapter 3. In this interpretation, classes `ConcreteFactory1` and `ConcreteFactory2` could be considered instances of the *Data Manager* micro pattern;
- *Sink*: a Sink class implements methods which do not propagate calls to any other classes. Therefore the two concrete factories, as implemented in the sample instance, are to be considered as Sinks;
- *Trait*: a Trait class is a class that doesn't have any state, i.e. that doesn't declare any instance field; each of the classes belonging to the sample pattern instance is therefore a Trait;
- *Pure type*: a Pure type class has only abstract methods, doesn't have any static member, and doesn't declare any field; the class `AbstractFactory` can be considered as a Pure type;
- *Pseudo class*: a Pseudo class is a class that could be rewritten as an interface: it doesn't have any concrete method, and its fields (if it has some) are all static; classes `AbstractFactory`, `AbstractProductA` and `AbstractProductB` are all Pseudo classes;
- *Implementor*: an Implementor class is a class which overrides the inherited abstract methods; `ConcreteFactory1` and `ConcreteFactory2` are two Implementors.

We should remark two points about the detection of these micro patterns. First of all, as we have seen for the Data manager micro pattern, some micro patterns lend themselves to different interpretations, so some more formalization about their concrete meaning seems to be needed. The unified micro-structures catalogue proposed in Chapter 3 aimed at solving also these ambiguities. Second, all the micro patterns identify characteristics that are placed at the class granularity level. For example, the Sink regards classes whose methods don't propagate calls to other classes. But what if a refactoring of a previous Sink class would have a method that makes some calls to another class? The new implementation will not be a Sink anymore. This problem is particularly relevant in a design pattern detection perspective. In fact, the micro patterns described so far are related to this basic implementation of the Abstract factory design pattern. But as it is known, each design pattern could have potentially infinite variants, so that if we compare different implementations of the same pattern they could present even completely different micro patterns. This is due to the fact that micro patterns identify only class-level characteristics, and do not pinpoint more fine-grained characteristics, as design pattern clues for example do. In order to solve this issue, we propose the detection of micro patterns basing on the rate of methods and attributes that satisfy a particular micro pattern condition. This aspect will be deeply discussed in Chapter 7.

3) Micro-structures detection in the Template method design pattern

A broadly exploited behavioural design pattern is the Template method. This pattern defines the general structure of an algorithm in terms of abstract methods representing its general operations, letting the subclasses of the template abstract class implement the actual behavior.

The basic implementation of the Template method design pattern is the following:

```
public class AbstractClass {

   public void templateMethod(){
        operation1();
        operation2();
   }

   abstract void operation1();
   abstract void operation2();
}
public class ConcreteClass extends AbstractClass {

   public void operation1(){
        …
   }

   public void operation2(){
        …
   }
}
```

The `templateMethod()` method only calls the two abstract methods `operation1()` and `operation2()`, that represent the general behavior of the algorithm represented by `templateMethod()`. The actual behavior is therefore to be specified by the concrete classes extending the class declaring the template method.

Design pattern clues

In this implementation of the Template method two design pattern clues are detected:

- *Template method*: an abstract method is called within the body of a concrete method. Both caller and callee methods belong to the same class;
- *Template implementor*: a class extends a class containing an instance of the Template method clue.

As we can see, the two clues perfectly fit to the definition of the Template method, and let the identification of both pattern roles possible.

Elemental design patterns

In this implementation of the Template method three EDPs can be found: 2 Abstract interface EDPs, one for each abstract method defined in the `AbstractClass`, and one Inheritance EDP, that defines the inheritance relationship between `ConcreteClass` and `AbstractClass`.

Micro patterns

One micro pattern could be detected in this implementation:

- *Outline*: a class where at least two methods invoke an abstract method on "this".

As we can see, the Outline micro pattern requires two methods invoking abstract methods. This is too restrictive for our scopes, as only one method (the template method) calling abstract methods is sufficient to grant the presence of the abstract class role. In Chapter 3 we proposed a relaxed definition for this micro pattern, granting its validity even if only one abstract method is invoked.

4) Micro-structures detection in the State design pattern

The basic implementation of the State design pattern is the following:

```
public abstract class State{
  public abstract void handle();
}

public class ConcreteStateA extends State{
  public void handle() {
        …
  }
}

public class ConcreteStateB extends State{
  public void handle() {
        …
  }
}

public class Context{
  private State state;

  public void request(){
        …
        state.handle();
  }
}
```

Design pattern clues

No design pattern clues can be found for this pattern. This is because no peculiarities for this pattern can be identified statically. We experienced, by analyzing different instances of the State pattern, that no hint for its detection or for the detection of one of its roles seems to be pointed out by some code detail. Hence, the identification of this pattern through the use of clues (and, in general, the use of any micro-structure in a static way) is a hard task.

Elemental design patterns

In this basic implementation of the State design pattern three EDPs can be found: two Inheritance EDPs, one for `ConcreteStateA` and the other for `ConcreteStateB`, that assert that these classes extend the State class, and one Delegate EDP, represented by the `state.handle()` method invocation between the `Context` class and the `State` class. The Delegate EDP is detected as the caller and callee class aren't in the same hierarchy and the caller and target methods have different signatures.

Micro patterns

Four micro patterns can be detected, three regarding the `State` class, the other regarding the `Context` class. The three micro patterns for the `State` class are:

- *Function poiner*: the class has a single public instance method, but no fields;
- *Trait*: the class is abstract and has no state (i.e. no variables);
- *Pure type*: the class declares only abstract methods, no static members, and no fields;

For the `Context` class we have the following micro pattern:

- *Function object*: the class has a single public instance method, and at least one instance field;

It seems clear that no micro pattern, as any other micro-structure, is able to capture the essence of this pattern, residing behind its behavior. The detected micro pattern can only give information from a structural point of view.

5) Micro-structures detection in the Composite design pattern

Considering the structural design patterns category, we propose the basic implementation of the Composite and we discuss the micro-structures that can be identified in it.

```
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(){
        …
   }
}
```

Design pattern clues

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

- *Abstract cyclic call*: the `Composite.operation()` method invokes the abstract method `Component.operation()` from within an enhanced for cycle;

- *Component method*: the two methods `Component.add()` and `Component.remove()` are instances of this clue, as they receive an object belonging to the same class as an input parameter;
- *Node class*: `Composite` is a Node class, as it extends a class declaring component methods, overriding them;
- *Leaf class*: `Leaf` is a Leaf class, as it extends a class declaring component methods without overriding them;
- *Same interface container*: `Composite` contains a list of `Component`s, that are objects that share the same interface with the `Composite` class;
- *Multiple redirections in family*: the Redirect in family EDP is detected inside a cycle, therefore it is supposed to work on a set of elements, like in this case, where the `operation()` method is invoked on each `Component` object belonging to the `Components` list.

Elemental design patterns

In this implementation 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 `Components` 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.

Micro patterns

The micro patterns for the `Component` class are:

- *Trait*: `Component` is an abstract class with no state;
- *Sink*: its methods do not invoke methods on any other class.

The `Composite` class is characterized by the following micro-patterns:

- *Function object*: it has only one public instance method and one instance field;
- *Box*: the only instance field is mutable;
- *Implementor*: it overrides the inherited abstract methods;
- *Overrider*: `Composite` also overrides the inherited non-abstract methods;

Finally, for the `Leaf` class we have:

- *Implementor*: it overrides the inherited abstract methods;
- *Stateless*: `Leaf` is a concrete class which has no fields.

6) Micro-structures detection in the Decorator design pattern

Let's now analyze the Decorator. Its structure is very similar to that of the Composite, so it is interesting to see if the micro-structure detection process can help in distinguishing these two patterns.

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

public class ConcreteComponent extends Component{
   public void operation(){
        …
   }
}

public abstract class Decorator extends Component {

   private Component component;

   public void operation(){
        component.operation();
   }
}

public class ConcreteDecorator extends Decorator {
   public void operation(){
        super.operation();
        addedBehaviour();
   }

   public void addedBehaviour(){
        …
   }
}
```

Design pattern clues

The following design pattern clues can be found in this basic implementation of the Decorator design pattern:

- *Instance in abstract class*: the abstract class `Decorator` maintains an instance to a different class (namely the `Component` class);
- *Instance in abstract referred*: the method `Decorator.operation()` invokes a method on the `Component`'s instance;
- *Reference to abstract class*: the instance declared by the `Decorator` class belongs to an abstract class;
- *Same interface instance*: the `Component` instance declared by the `Decorator` class is compatible with the same interface; in this case, differently from what happens for the

Composite, we do not deal with a set of objects compatible with the same interface, but only one is considered;

Elemental design patterns

In the basic implementation of the Decorator the following EDPs have been detected. As for the Composite design pattern, an identical Abstract interface EDP has been detected for the `Component` class. Three Inheritance EDPs connect `ConcreteComponent`, `Decorator` and `ConcreteDectorator` to the corresponding superclasses. A Redirect in family EDP has been detected in the `Decorator` class, similarly to that found in the `Composite` class in the Composite design pattern. Finally, the super method invocation in `ConcreteDecorator.operation()` is an instance of the Extend method EDP, where a method belonging to the superclass is enriched with some added behavior.

Micro patterns

In this implementation of the Decorator, the following micro patterns can be identified. For the `Component` class we have:

- *Trait*: `Component` is an abstract class with no state;
- *Sink*: its methods do not invoke methods on any other class;

For the `Decorator` class we can identify the following micro patterns:

- *Function object*: the class has only one public instance method and one instance field;
- *Box*: the only instance field is mutable;
- *Overrider*: `Decorator` also overrides the inherited non-abstract methods;

The `ConcreteComponent` class is characterized by the following micro patterns:

- *Implementor*: it overrides the inherited abstract methods;
- *Stateless*: `ConcreteComponent` is a concrete class which has no fields.

Finally, the `ConcreteDecorator` class has the following micro patterns:

- *Stateless*: as the `ConcreteComponent`, `ConcreteDecorator` has no fields;
- *Overrider*: it overrides the non-abstract methods inherited by `Decorator`.

4.4.2. Micro-structures relevance evaluation for design pattern detection

The examples of micro-structures detection in the six different design patterns lead to meaningful considerations about their exploitation in a design pattern detection activity. As it results clear from the examples, the various categories of micro-structures identify very different peculiarities that can be found in the design pattern instances.

Without minding at this heterogeneity of elements (that is obviously unavoidable, due to the different characteristics and detail levels of each kind of micro-structures), the real questions are: how can these elements help us in the detection of design patterns? Which are the most relevant elements? Do they provide enough information for design patterns detection? Trying to give an answer to these questions is the main scope of this section.
To go further into discussion, we shall remind that design patterns are at a first glance a composition of roles based on a well defined structure. Each role is usually played by a single class in the design pattern, with the aim to perform a well-defined task inside the pattern (like the singleton role in the Singleton design pattern, where its instance has to be unique at run-time). Or else, they can provide basic functionalities that are to be used by the other design pattern roles: for example, the Concrete products in the Abstract factory design pattern aren't designed to perform particular functionalities, but they are just classes to be instantiated by the factories. The first kind of roles can be defined as *active roles*, as they actually contribute to the behaviour of the pattern. The other roles can be said to be *passive roles*, as they are exploited by the active roles in order to perform their operations, and they don't provide any other functionality to the pattern itself.
Therefore, the activity of design pattern detection in a software system through static analysis should be supported by two intermediate steps:

- a first identification of the basic structure of the pattern, i.e. the extraction of classes whose relationships with each other in terms of referencing and inheritance are compatible with those specified for the pattern to be detected;
- the identification of each role played inside a pattern, analyzing the structures extracted in the first step in order to understand if the classes that compose it are actually playing the supposed roles.

Hence, micro-structures are to be considered useful for design pattern detection if they help us in identifying both the pattern architectures and those characteristics that are proper of each single role of design patterns.
In the following, we analyze whether the detected micro-structures are to be considered relevant for the identification of both pattern structures and roles, according to the constraints asserted by each pattern definition.
First of all, we define some fundamental characteristics that cannot be missed while implementing each single role of the various design patterns. These characteristics (that are necessarily informally introduced, due to the informal nature of design patterns) can be derived both by the design pattern catalogue [GHJV94] and by personal experiences, capturing all the fundamental commonalities that can be found inside different implementations of each single design pattern role.

For the Singleton design pattern we have only the Singleton role:

Singleton
- It must grant the presence of a unique instance of itself during execution.

This is the only request for the Singleton role, and therefore for the Singleton design pattern. In the design pattern catalogue nothing is specified about how to grant this request, as many different implementations of this design pattern (as obviously for all the other patterns) may exist.

For the Abstract factory design pattern, four different roles have been identified (we don't consider the presence of the client, as we aim at the identification of the roles of the core structure of the pattern), that should have these peculiarities:

Abstract factory
- it must provide getter methods to obtain references to the abstract products.

Concrete factories
- they must extend the Abstract factory;
- they must redefine the methods provided by the Abstract factory in order to allow the return of the correct concrete products.

Abstract products
- They must be realized by abstract classes.

Concrete products
- They must be realized by concrete classes that extend their abstractions.

For the Template method, two different roles with the following characteristics are specified:

Abstract class
- It must define a concrete method invoking at least one abstract method defined in the same class;

Concrete class
- It must extend the abstract class, thus implementing the abstract methods defined by it.

For the State design pattern we can define the following characteristics that cannot be currently represented with micro-structures, but that we report for completeness:

Context

- It must maintain a reference to the abstract state;
- It must invoke the `handle()` method on the abstract state;

State

- It must provide an interface for the concrete states;

Concrete states

- They must extend the abstract state;
- They must provide mechanisms to switch from one state to another through the interface method provided by the abstract state.

The last characteristic cannot be captured by any of the currently available micro-structures, as it is strictly related with the behavioural and dynamic nature of the pattern.

For the Composite design pattern we have the following characteristics:

Component

- It must provide methods to handle objects of the same type;
- It must implement an operational method as well.

Composite

- It must extend the component;
- It must maintain a list of components;
- It must implement the abstract handling methods defined by the component;
- It must invoke the operational method on all the components belonging to the list.

Leaf

- It must extend the component;
- It must not deal with objects compatible with its same interface, as the composite does;
- It should override the operational method provided by the component.

For the Decorator design pattern we can define the following peculiarities:

Component

- It must be an abstract class;
- It must define an operational method.

Decorator

- It must extend the component;

- It must maintain an instance of the component;
- It must override the operational method invoking it on the declared component instance.

Concrete components
- They must extend the component;
- They should extend the operational method.

Concrete decorators
- They must extend the decorator;
- They must enrich the operational methods with some other behavior.

The relevance of micro-structures is to be evaluated for the analyzed patterns, considering both the extraction of pattern structures and the identification of each single role. Basing on our experience, on the characteristics defined for the various roles and on the examples provided before, for each design pattern role and for each micro-structure we indicate if each of them is:

- *Relevant*: it helps in identifying the relationships that subsist among the various pattern roles, as well as the peculiarities that characterize each role; hence it points out some of the constraints we have just introduced;
- *Irrelevant*: it is not useful for the extraction of any kind of meaningful information about the pattern.

In order to avoid excessive information, the considerations about the relevance of each micro-structure will be reported in the next sub-section, where actual implementations of design patterns are analyzed. Here we only provide some general considerations about the micro-structures relevance, which can be easily verified considering the results provided in the next part. As we will notice from the results tables, only the design pattern clues are to be considered in any case relevant for the detection of the roles of each of the studied pattern. This observation is directly related with the nature of design pattern clues, which have been introduced for design pattern detection purposes. Micro patterns revealed themselves useful only in few cases, while EDPs are always useful for the extraction of pattern architectures. Anyhow, by tagging some micro-structures as relevant for the detection of particular roles inside a design pattern, we cannot assert that the identification of the design patterns is automatically granted. In fact, a design pattern is something more than an aggregation of roles characterized by the simple properties we have defined: these roles must be interconnected, they may reveal non-trivial interactions, and they surely present behavioural characteristics which are not yet revealed by any micro-structure, as they actually capture static aspects of a design pattern implementation. Hence the micro-structures currently assert that the analyzed code structure or architecture may represent a potential instance of a design pattern rather than a precise one. The information they point out (if meaningful) catches some fundamental characteristics that usually need for further inspection to identify the more complex properties the design pattern is characterized by.

4.4.3. Micro-structures detection in design pattern implementation variants

The third step for the validation of the usefulness of micro-structures for design pattern detection consists in analyzing actual design pattern implementations that are correct variants of the standard implementations presented before. For each pattern we considered a set of instances (12 Singletons, 16 Abstract factories, 12 Template methods, 15 Composites and 9 Decorators) that have been analyzed role by role.

Role name	Micro-structure category	Micro-structure	No. of instances presenting the micro-structure	% of instances presenting the micro-structure	Relevance of the micro-structure for the design pattern structure	Relevance of the micro-structure for the design pattern role
Singleton	EDPs	Create object	12	100%	Relevant	Irrelevant
	DP Clues	Protected instantiation	12	100%	Relevant	Relevant
		Private self instance	7	58%	Relevant	Relevant
		Static self instance	8	67%	Relevant	Relevant
		Single self instance	11	92%	Relevant	Relevant
		Controlled self instatiation	8	67%	Relevant	Relevant
		Concrete product getter	12	100%	Relevant	Irrelevant
	Micro patterns	Function object	12	100%	Relevant	Irrelevant
		Common state	10	83%	Relevant	Irrelevant
		Restricted creation	11	92%	Relevant	Relevant
		Data manager	1	8%	Irrelevant	Irrelevant
		Sink	2	17%	Irrelevant	Irrelevant

***Table 4.2** – Results for the Singleton instances*

The State design pattern is not considered in the rest of the chapter because there are not relevant micro-structures for the detection of its roles. For each role, only the micro-structures that should characterize it (and that have been detected in the sample implementations, as explained in Section 4.4.1) have been considered.
Table 4.2 resumes the results of micro-structures detection for the Singleton design pattern instances. For this pattern the results are encouraging. Considering both its architecture and its single role, many relevant elements can be detected in the majority of the analyzed instances. In particular, the Create object EDP, which is relevant for the Singleton structure, and the Protected instantiation clue and the Function object micro pattern, that are important for the existence of the Singleton role, are found in all of the 12 instances. The two relevant elements that are found in fewer instances are the Private and Static self instance clues, that can be detected in around the 60% of instances.

Role name	Micro-structure category	Micro-structure	No. of instances presenting the micro-structure	% of instances presenting the micro-structure	Relevance of the micro-structure for the design pattern structure	Relevance of the micro-structure for the design pattern role
Abstract factory	EDPs	Abstract interface	13	81%	Relevant	Irrelevant
	DP Clues	Abstract product returned	13	81%	Relevant	Relevant
		Parent product returned	13	81%	Relevant	Relevant
	Micro patterns	Trait	5	31%	Irrelevant	Irrelevant
		Pure type	3	18%	Irrelevant	Irrelevant
		Pseudo class	3	18%	Irrelevant	Irrelevant
Concrete factories	EDPs	Create object	15	94%	Relevant	Irrelevant
		inheritance	15	94%	Relevant	Irrelevant
	DP Clues	Concrete products returned	0	0%	Relevant	Relevant
	Micro patterns	Data manager	0	0%	Relevant	Irrelevant
		Sink	1	6%	Irrelevant	Irrelevant
		Trait	0	0%	Irrelevant	Irrelevant
		Implementor	11	69%	Relevant	Irrelevant
Abstract products	Micro patterns	Trait	1	6%	Irrelevant	Irrelevant
		Pseudo class	10	63%	Irrelevant	Irrelevant
Concrete products	EDPs	Inheritance	13	81%	Relevant	Irrelevant
	Micro patterns	Trait	0	0%	Irrelevant	Irrelevant

Table 4.3 *– Results for the Abstract factory instances*

Table 4.3 collects the results of micro-structures detection for the Abstract factory design pattern instances. For this pattern, EDPs behave well for the extraction of its architecture. Between 81% and 94% of instances implement EDPs that help in identifying the general architecture of the pattern. As far as the single roles are concerned, the 81% of the Abstract factory roles implement the clues detected for this role in Section 4.4.1. Other micro-structures categories are in general irrelevant for the detection of any particular role in this pattern.

Table 4.4 resumes the results of micro-structures detection for the Template method instances. This behavioural pattern is well detected by the use of micro-structures, both in terms of elements useful for the extraction of its structure and for the detection of the two roles belonging to it. The elements that have been defined for this pattern and that are fundamental for its detection have been detected in the 100% of the considered instances.

Role name	**Micro-structure category**	**Micro-structure**	**No. of instances presenting the micro-structure**	**% of instances presenting the micro-structure**	**Relevance of the micro-structure for the design pattern structure**	**Relevance of the micro-structure for the design pattern role**
Abstract class	EDPs	Abstract interface	12	100%	Relevant	Irrelevant
		Conglomeration	12	100%	Relevant	Irrelevant
	DP Clues	Template method	12	100%	Relevant	Relevant
	Micro patterns	Outline	12	100%	Relevant	Relevant
Concrete class	EDPs	Inheritance	12	100%	Relevant	Irrelevant
	DP Clues	Template implementor	12	100%	Relevant	Relevant

Table 4.4 *– Results for the Template method instances*

This is the only case we currently know of a design pattern automatically detectable through the only identification of micro-structures inside the code. In particular, this is quite odd, considering that the Template method is a behavioural design pattern. In fact, this category of patterns is generally characterized by a small set of micro-structures, that often don't characterize their roles very well, but they rather lay on dynamic characteristics that cannot be formalized in elements that are detectable from source code by static analysis techniques (the State design pattern is probably the best representative of this group of patterns).

Table 4.5 resumes the results about the analysis of the Composite design pattern instances. As we can notice, EDPs behave well as far as the identification of the Composite's architecture is concerned. The fundamental inheritance relationship between the

Composite and the Component role is identified in the 93% of the analyzed instances. A 73% rate of inheritance between the Leaf and the Component is due to the fact that 4 instances do not provide any class playing the Leaf role. Analyzing the micro-structures relevant for each single role, we notice that 13 Composite roles have at least a fundamental Redirect in family, and for 12 of these instances this EDP is identified within a cycle, which lets us assume that this method invocation is repeated for the elements of a certain collection. This is also supported by the presence of 13 Same interface container clues, that have been detected in the same instances presenting the Multiple redirections in family clue. Only eight Component roles declare actual component methods, and consequently only eight Composite roles can be considered Node classes.

Role name	**Micro-structure category**	**Micro-structure**	**No. of instances presenting the micro-structure**	**% of instances presenting the micro-structure**	**Relevance of the micro-structure for the design pattern structure**	**Relevance of the micro-structure for the design pattern role**
Component	EDPs	Abstract interface	12	80%	Relevant	Irrelevant
	DP Clues	Component method	8	53%	Irrelevant	Relevant
	Micro patterns	Trait	4	27%	Irrelevant	Irrelevant
		Sink	4	27%	Irrelevant	Irrelevant
Composite	EDPs	Inheritance	14	93%	Relevant	Relevant
		Create object	1	7%	Relevant	Relevant
		Redirect in family	13	87%	Relevant	Relevant
	DP Clues	Abstract cyclic call	10	67%	Relevant	Relevant
		Node class	8	53%	Relevant	Relevant
		Same interface container	13	87%	Relevant	Relevant
		Multiple redirections in family	13	87%	Relevant	Relevant
	Micro patterns	Function object	0	0%	Irrelevant	Irrelevant
		Box	6	40%	Irrelevant	Irrelevant
		Implementor	9	60%	Relevant	Irrelevant
		Overrider	1	7%	Relevant	Irrelevant
Leaf	EDPs	Inheritance	11	73%	Relevant	Irrelevant
	DP Clues	Leaf class	4	27%	Relevant	Relevant
	Micro patterns	Implementor	11	73%	Relevant	Irrelevant
		Stateless	7	47%	Relevant	Irrelevant

Table 4.5 *– Results for the Composite instances*

Table 4.6 collects the results obtained during the analysis of the Decorator instances. As it appears clear in [GHJV94], the architectures of the Composite and the Decorator design patterns in terms of classes and static relationships among them is very similar. An abstract component class is on top of the pattern hierarchy in both cases, and both the composite and the decorator classes extend their corresponding component class also maintaining a reference to it. The distinction between these two patterns is therefore to be searched inside the essence of each role. While the structures of these patterns can be well detected through the identification of EDPs, the micro-structures characterizing the various roles must necessarily be different and specific for each pattern, in order to distinguish Composite from Decorator instances.

Role name	**Micro-structure category**	**Micro-structure**	**No. of instances presenting the micro-structure**	**% of instances presenting the micro-structure**	**Relevance of the micro-structure for the design pattern structure**	**Relevance of the micro-structure for the design pattern role**
Component	EDPs	Abstract interface	9	100%	Relevant	Irrelevant
	Micro patterns	Trait	1	11%	Irrelevant	Irrelevant
		Sink	4	44%	Irrelevant	Irrelevant
Decorator	EDPs	Inheritance	6	67%	Relevant	Irrelevant
		Redirect in family	5	56%	Relevant	Irrelevant
	DP Clues	Instance in abstract class	6	67%	Relevant	Relevant
		Instance in abstract referred	5	56%	Relevant	Relevant
		Reference to abstract class	6	67%	Relevant	Relevant
		Same interface instance	6	67%	Relevant	Relevant
	Micro patterns	Function object	3	33%	Irrelevant	Irrelevant
		Box	5	56%	Irrelevant	Irrelevant
		Overrider	0	0%	Irrelevant	Irrelevant
Concrete components	EDPs	Inheritance	9	100%	Relevant	Irrelevant
	Micro patterns	Stateless	6	67%	Relevant	Irrelevant
		Implementor	7	78%	Relevant	Irrelevant
Concrete decorators	EDPs	Inheritance	9	100%	Relevant	Relevant
		Extend method	4	44%	Relevant	Relevant
		Redirect in famlily	6	67%	Relevant	Relevant
	Micro patterns	Stateless	4	44%	Irrelevant	Irrelevant
		Overrider	1	11%	Irrelevant	Irrelevant

***Table 4.6** – Results for the Decorator instances*

The distinction between the Composite and the Decorator role is to be searched in the kind of reference to the Component that they maintain. While for the Composite design pattern we had a Same interface container clue, in the Decorator we only need one single instance of the Component class, represented by the Same interface instance clue. The method invocation to the Component is again a Redirect in family, but in this case enriched with the presence of an Extend method, which helps us in understanding that a certain method has been added new behavior.

4.5. An association among pattern roles and micro-structures for their detection

Basing on the pattern roles characteristics defined before, on the relevance of the various micro-structures discussed so far and on the results obtained in the identification of micro-structures inside real pattern instances, we collect in the next tables the micro-structures that are useful to identify each particular pattern characteristic.
The micro-structures useful for the identification of the Singleton design pattern belong to the design pattern clues and to the micro patterns (Table 4.7). As this pattern is constituted by a single class, without any particular method invocation, no fundamental EDPs for it have been detected.

Singleton

Role name	*Characteristic*	*EDPs*	*DP Clues*	*Micro patterns*
Singleton	It must grant the presence of a unique instance of itself during execution		Single self instance, Private self instance, Static self instance, Protected instantiation	Function object, Restricted creation

***Table 4.7** – Micro-structures usefulness in the Singleton*

The issues that characterize the various roles belonging to the Abstract factory can be revealed by a good set of micro-structures belonging to all the categories considered in this comparison (Table 4.8). The structural constraints of this pattern are pointed out mainly by EDPs (micro patterns reveal possibly useful only in the detection of the Concrete factory class). The essence of this pattern is enclosed in the Abstract factory role. This role can be identified with the use of design pattern clues, which helps in identifying the role characteristics, besides its structural constraints.

For the identification of the Template Method pattern (Table 4.9) we can rely on the detection of only two clues, which represent both necessary and sufficient conditions for the existence of both the Abstract class and the Concrete class roles. EDPs can be used only in the detection of the pattern structure, while the Outline micro pattern can be used to detect the Abstract class role.

Abstract factory

Role name	*Characteristic*	*EDPs*	*DP Clues*	*Micro patterns*
Abstract factory	It must provide getter methods for the obtainment of references to the abstract products	Abstract interface	Abstract products returned, Parent products returned	
Concrete factories	They must extend the Abstract factory	Inheritance		
	They must redefine the methods provided by the Abstract factory in order to allow the return of the correct concrete products		Concrete products returned	Implementor
Abstract products	They must be realized by abstract classes	Abstract interface		
Concrete products	They must be realized by concrete classes that extend their abstractions	Inheritance		

Table 4.8 *– Micro-structures usefulness in the Abstract factory*

Template method

Role name	*Characteristic*	*EDPs*	*DP Clues*	*Micro patterns*
Abstract class	It must define a concrete method calling at least one abstract method defined in the same class	Abstract interface, Conglomeration	Template method	Outline
Concrete class	It must extend the abstract class, thus implementing the abstract methods defined by it	Inheritance	Template implementor	

Table 4.9 *– Micro-structures usefulness in the Template method*

For the Composite design pattern (Table 4.10) it is evident how the detection of its structure can be pursued exploiting EDPs. The issues related to each single role apart from their architecture are detectable through the use of clues.

Composite

Role name	*Characteristic*	*EDPs*	*DP Clues*	*Micro patterns*
Component	It must provide methods to handle objects of the same type	Abstract interface	Component method	
	It must implement an operational method		Multiple redirections in family	
Composite	It must extend the component	Inheritance		
	It must maintain a list of components		Same interface container	
	It must implement the abstract handling methods defined by the component		Node class	
	It must invoke the operational method on all the components belonging to the list	Redirect in family	Multiple redirections in family	
Leaf	It must extend the component	Inheritance		
	It must not deal with objects of the same interface, as the composite does		Leaf class	
	It should override the operational method provided by the component		Leaf class	Implementor

Table 4.10 *– Micro-structures usefulness in the Composite*

As for the Composite, also for the detection of the Decorator pattern structure EDPs are useful (Table 4.11). The design pattern clues are useful to identify the fundamental Decorator role, which is different from the Composite role of the homonym pattern as the Decorator maintains a single instance of the class with the same interface, instead of a collection of elements.

As it appears clear from the previous tables, and considering the discussion about micro-structures detection inside the real examples we provided, the main micro-structures involved in the identification of pattern roles characteristics are the EDPs, and the design pattern clues. EDPs are to be used mainly for the identification of architectural constraints on the patterns (which specify the abstract roles, the inheritance hierarchies and the references among the various classes).

Decorator

Role name	*Characteristic*	*EDPs*	*DP Clues*	*Micro patterns*
Component	It must be an abstract class	Abstract interface		
	It must define an operational method	Redirect in family		
Decorator	It must extend the component	Inheritance		
	It must maintain an instance of the component		Reference to abstract class, Same interface instance	
	It must override the operational method invoking it on the declared component instance	Redirect in family		
Concrete components	They must extend the component	Inheritance		
	They should extend the operational method	Extend method		
Concrete decorators	They must extend the decorator	Inheritance		
	They must enrich the operational methods with some other behaviour	Redirect in family, Extend method		

Table 4.11 *– Micro-structures usefulness in the Decorator*

Design pattern clues are especially useful and suited to identify the main characteristics of each single design pattern role, aside from design patterns architectures. Finally, as it appears clear from the tables, micro patterns did not prove in general very useful for the detection of the discussed patterns.

4.6. Concluding remarks

In this chapter we have presented the considered micro-structures under six aspects that allowed us to provide a comparative evaluation among them.
The main contribution of this chapter is related to a deep comprehension of the building blocks composing design patterns. This understanding can reveal particularly useful for a better application of design patterns, for the improvement of program comprehension and for the evaluation of design quality. Micro-structures for design pattern detection have

then been analyzed according to two main aspects, namely their usefulness for the detection of pattern structures and for the identification of single pattern roles. In this context, the micro-structures which are particularly relevant for the identification of pattern structures are mainly EDPs. On the other hand, DP clues revealed more suitable for the detection of the roles played within the patterns. Micro patterns have revealed themselves useful only in particular cases (e.g., for the recognition of the Template method). These findings were also confirmed by an analysis of the micro-structures implemented in sample sets of design pattern instances, letting us specify those micro-structures that are crucial for the presence of each of the considered patterns inside the analyzed systems.

As it also appeared from the comparison and from the pursued experimentations, the considered micro-structures are of various types and capture different aspects at different abstraction levels. For example, EDPs capture object-oriented best practices and are independent from any programming language; clues aim to identify basic structures that are peculiar to each design pattern; micro-patterns express common programming techniques. Moreover, these micro-structures have been defined with different purposes. Essentially, EDPs and DP clues have among their main objectives the recognition of design patterns, while micro patterns focus on the description of programming techniques.

The heterogeneity of the considered micro-structures may lead to two conclusions. The first one is related to the fact that there is little agreement in what design patterns are built of and what kind of techniques or artifacts should be used for their detection. The second remark testifies the interest in understanding design patterns, and the worth of the effort necessary for their recognition in the context of reverse engineering, due to the meaning behind them which enables us to understand also the "why" of a design, not only the "how" of an implementation detail.

The micro-structures detection process is the core concept residing behind the detection of design patterns within MARPLE [ATZM08], for the refinement of design pattern detection results obtained by different tools as described in the next chapter, and for software architecture reconstruction activities as described in Chapter 6.

Chapter 5

Micro-structures for the validation and refinement of design pattern detection tools results

Abstract

In this chapter 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 subject system. From this observation, we introduce an approach based on micro-structures that is aimed to discard the false positives from the detected results, hence improving the precision of the analyzed tools.

5.1. Detection of design patterns through four design pattern detection tools

Several tools for design pattern detection exist. Each of them is based on a different approach, adopts different strategies to detect patterns, and in general can identify only a subset of the defined patterns.
The work described in this chapter is focused on the experimentation of four well-known tools, namely Design Pattern Detection Tool [TC06], PINOT [SO06], FUJABA [NNZ00] and Web of Patterns [DE07]. In this section we report the results obtained by these tools on the analysis of the JHotDraw 6.0b1 framework [JHD]. We will focus our attention on this system as one of the reasons behind the development of JHotDraw is the demonstration of 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 to. In this way, 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 to which patterns. Table 5.1 resumes the results produced by the four considered tools on JHotDraw 6.0b1, in terms of the number of instances they are able to detect for each pattern.

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

***Table 5.1** – Results of the design pattern detection process obtained by four tools on the analysis of JHotDraw 6.0b1*

[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.

From the analysis of the above table, different considerations shall be done. First of all, at now no tool is able to detect or provide techniques for the identification of the whole set of design patterns defined by Gamma. This may be due to the difficulty in the definition of a detection strategy for some design patterns (especially for the behavioural ones), or to the lack of structural or programming hints that can help for the detection, as well as to the lack of formalization of the patterns themselves, or to the different detection approaches adopted by the various tools. A second (and more important) consideration is related to the different results obtained by the tools in the detection of the same pattern. As it can be noticed, it doesn't exist a pattern for which the tools return the same number of instances. And, 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. This is once again due to the fact that the tools adopt different detection strategies (hence some instances that do not strictly comply with the constraints adopted by each single tool will not be detected by it), and especially to the presence of different design pattern variants inside the code.
As the different tools identify a considerable number of false positives, hence worsening the precision rates, in this chapter we present a methodology aimed at discarding false positive instances through the help of micro-structure-based refinement rules.
Our approach reveals useful as it aims at improving the precision of design pattern detection tools, therefore obtaining results that are more close to the actual design pattern instances implemented in the analyzed systems. As we are only focused on the refinement of results obtained by third-party tools, in this chapter we do not describe any new approach for design pattern detection. We consequently do not face or propose a possible solution to the variants problem, as we just analyze the instances provided by other tools as they are. For the same reasons, the refinement approach described here (which is not a detection approach) doesn't aim to maximize recall values, but only to improve the precision rates of common design pattern detection tools.

5.2. Refinement rules definition

The main problem concerned with design pattern detection is the variants problem. Due to the mainly informal nature of design patterns, tools for design pattern detection generally adopt different detection algorithms and produce different results, even while analyzing the same target systems. Therefore, the average precision and recall [BR99] values generally differ among the various tools and the various design patterns they are able to detect. Given a subject system *S*, we indicate with *tp* the number of real design pattern instances implmentend in *S* and identified by the pattern detection tools (*true* *positives*); *fp* indicates the number of instances which have been detected by the tool on *S* but which are not correct realizations of the subject pattern (*false positives*), while *fn* indicates the number of pattern instances implemented in *S* which cannot be identified by the detection tool (*false negatives*). The precision *P* of a design pattern detection tool is computed as $P = tp / (tp + fp)$, and indicates how much of the detected instances are actual and correct pattern implementations. The more *P* is close to 1, the more the tool is precise and the less false positives the tool detects. The recall *R* is defined as $R = tp / (tp + fn)$, and indicates how many of the actually implemented pattern instances the tool is able to recover.

In order to increase the accuracy 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 the micro-structures that can be detected in each pattern. First of all, we must recall that every design pattern is constituted by one or more roles, according to [GHJV94], each one played by a single class. Micro-structures do not place themselves on the general design pattern level of abstraction, 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 that characterize them, in a similar way to what discussed in Chapter 4, Section 4.5.

The rules will be based on two micro-structures: the EDPs, that 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, as also seen in Chapter 4. Micro patterns are now not considered, as they have little relevance for the identification of characteristics related to the patterns that are detected by the majority of the considered tools (and as also generally underlined in the previous chapter).

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 kink on the role node). This is also evident from the micro-structures definitions provided in Chapter 3.

Clues and EDPs aren't to be considered sufficient conditions for the correctness of pattern instances. Some of them are on the other hand necessary conditions, while the remaining ones are used to further enrich and characterize the analyzed instances. The evaluation of the necessary conditions will reveal especially useful in the refinement process, as ambiguous instances will be discarded or accepted basing on the verification of these conditions.

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 reported underlined.
For each role we indicate all the clues that may be identified inside it. This does not mean that they all have to be detected inside a class in order to assert that the class plays the specific role. Each role may present only a subset of these elements and still be a correct role for the corresponding pattern.
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. This choice is due to the fact that these patterns are recognizable by the majority of tools. Moreover, defining rules for those patterns that cannot be detected by the tools, or for which no instances have been identified, would be of scarce interest, as no refinement process can be pursued on them.
Even if three out of the four considered tools assert to be able to detect instances of the State and Strategy patterns, we will not provide refinement rules for them, as we do not have identified any peculiar micro-structure which could help in their validation, as also discussed in Chapter 4. This is due to the strictly behavioural nature of these patterns, which cannot be represented in the form of elements that can be statically detected from source code analysis.

Table 5.2 reports and explains the refinement rules for the considered creational patterns.

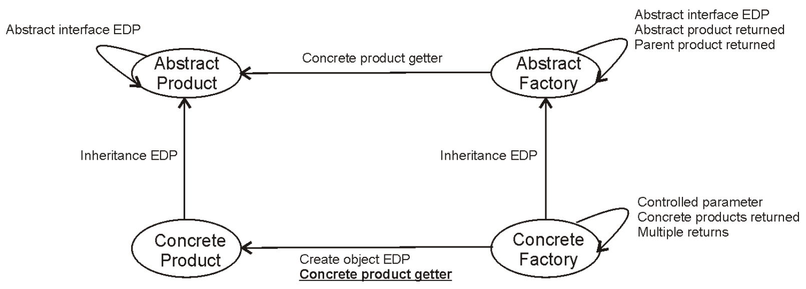
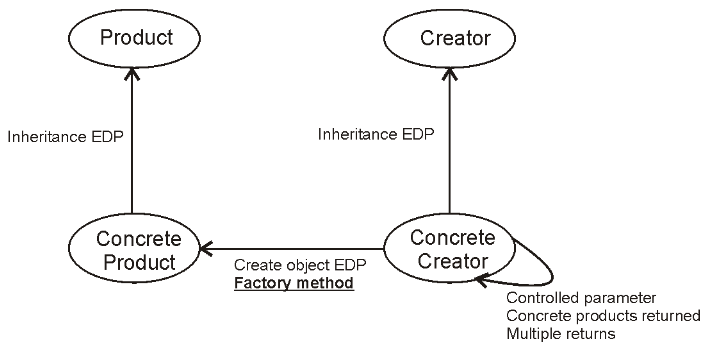
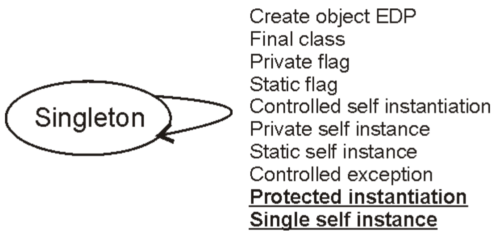
Pattern name	Refinement rule	Explanation
Abstract factory	 Abstract interface EDP Abstract Product Concrete product getter Abstract Factory Abstract interface EDP Abstract product returned Parent product returned Inheritance EDP Inheritance EDP Controlled parameter Concrete products returned Multiple returns Concrete Product Create object EDP Concrete product getter Concrete Factory	The Abstract factory pattern is composed by four core roles. The necessary clue for this pattern is the Concrete product getter, which grants that the Concrete factory implements at least one method which returns an instance of the Concrete product.
Factory method	 Product Creator Inheritance EDP Inheritance EDP Concrete Product Create object EDP Factory method Concrete Creator Controlled parameter Concrete products returned Multiple returns	Factory method is very similar in structure to the Abstract factory. In this pattern the Factory method clue is necessary. It assures the existence of a method which creates instances of the Concrete product within the Concrete creator.
Singleton	 Singleton Create object EDP Final class Private flag Static flag Controlled self instantiation Private self instance Static self instance Controlled exception Protected instantiation Single self instance	The Singleton design pattern is constituted by a single role. The necessary clues for this pattern are Protected instantiation (which avoids the creation of instances from external classes), and the Single self instance (which grants the presence of only one instance for the Singleton class).

***Table 5.2** – Refinement rules for the considered creational design patterns*

Table 5.3 describes the refinement rules for the considered structural design patterns.

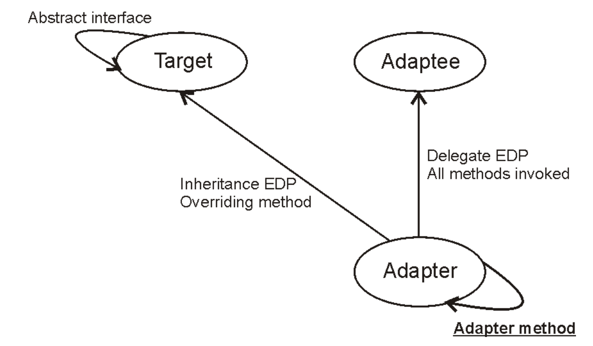
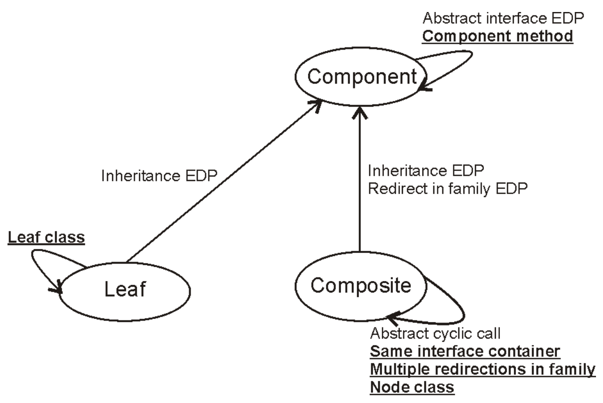
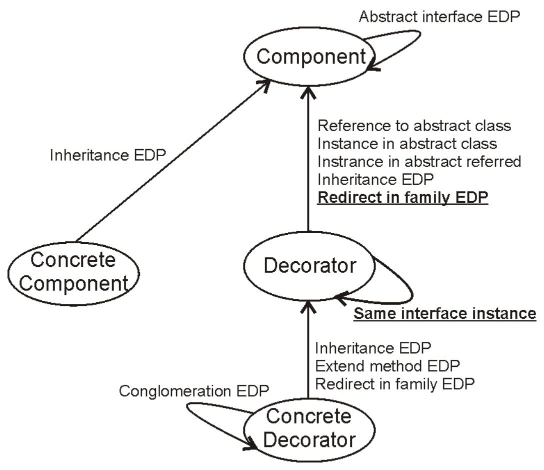
Pattern name	Refinement rule	Explanation
Adapter	 Abstract interface Target Adaptee Delegate EDP All methods invoked Inheritance EDP Overriding method Adapter Adapter method	The Adapter design pattern is constituted by three roles: Target, Adaptee and Adapter. The Adapter role overrides the methods provided by the Target in order to be able to invoke the methods declared by the Adaptee, letting therefore the Target interface be compatible with the Adaptee. This property is granted by the necessary Adapter method clue.
Composite	 Abstract interface EDP Component method Component Inheritance EDP Inheritance EDP Redirect in family EDP Leaf class Leaf Composite Abstract cyclic call Same interface container Multiple redirections in family Node class	The Composite pattern is formed by two core roles, Component and Composite, and one additional role, Leaf, which identifies child component elements with no more children. Component must define component methods (Component method clue). The Composite must have a collection of Component elements (Same interface container clue), and override the component methods defined by the component (Node class). The component methods are to be invoked on all the components belonging to the collection (Multiple redirections in family).
Decorator	 Abstract interface EDP Component Inheritance EDP Reference to abstract class Instance in abstract class Instrance in abstract referred Inheritance EDP Redirect in family EDP Concrete Component Decorator Same interface instance Inheritance EDP Extend method EDP Redirect in family EDP Conglomeration EDP Concrete Decorator	The Decorator pattern is structurally similar to the Composite. Three core roles constitute this pattern, namely Component, Decorator and Concrete decorator. The Decorator must maintain a single reference to the Component (Same interface instance clue), while the concerete decorators must enrich the methods defined by the Decorator the new behavior (Extend method EDP).

Table 5.3 *– Refinement rules for the considered structural design patterns*

Table 5.4 introduces the refinement rules for the considered behavioural patterns, namely the Template method and the Visitor.

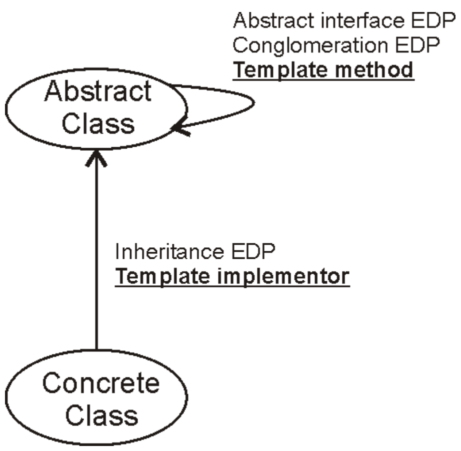
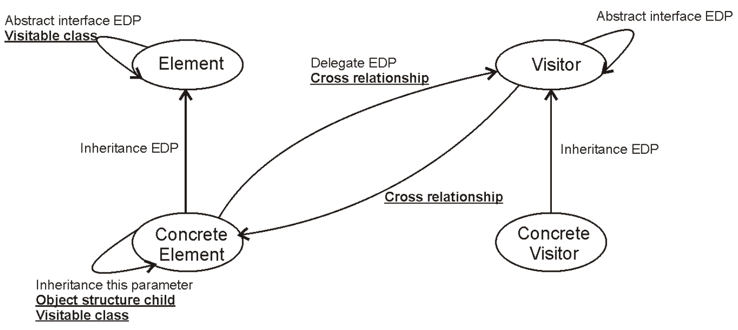
Pattern name	Refinement rule	Explanation
Template method	 Abstract interface EDP Conglomeration EDP Template method Abstract Class Inheritance EDP Template implementor Concrete Class	The Template method pattern is formed by two roles. The Abstract class is characterized by the necessary Template method clue, which grants that a concrete method invokes abstract methods inside its body. The Concrete class is characterized by the Template implementor clue, as it gives an implementation to the abstract methods invoked by the template method defined in the Abstract class.
Visitor	 Abstract interface EDP Visitable class Element Delegate EDP Cross relationship Visitor Abstract interface EDP Inheritance EDP Inheritance EDP Cross relationship Concrete Element Concrete Visitor Inheritance this parameter Object structure child Visitable class	The Visitor design pattern is formed by four roles. The Concrete elements to be visited must belong to a well defined object structure (Object structure child clue), like trees. They must also provide methods to accept visitor classes in order to be inspected (Visitable class clue).

***Table 5.4** – Refinement rules for the considered behavioural design patterns*

Now that we have introduced the refinement rules for the considered patterns, we give a description of the refinement process, and provide some refinement examples in order to gain confidence with it. Providing a detailed description of the refinement of each of the detected instances would take too much space and would be of scarce interest.

5.3. The pattern instances refinement process

Figure 5.1 resumes the adopted refinement process, which is mainly divided in four consecutive phases. In the figure, the grey rectangles represent the tools involved in the process. Rounded rectangles are related to the needed artifacts and representations, while normal rectangles represent the pursued activities and operations.

First of all, design pattern instances are identified from the system through the different detection tools; then they manually evaluated, in order to understand which of them are correct instances, and which are on the other hand 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.

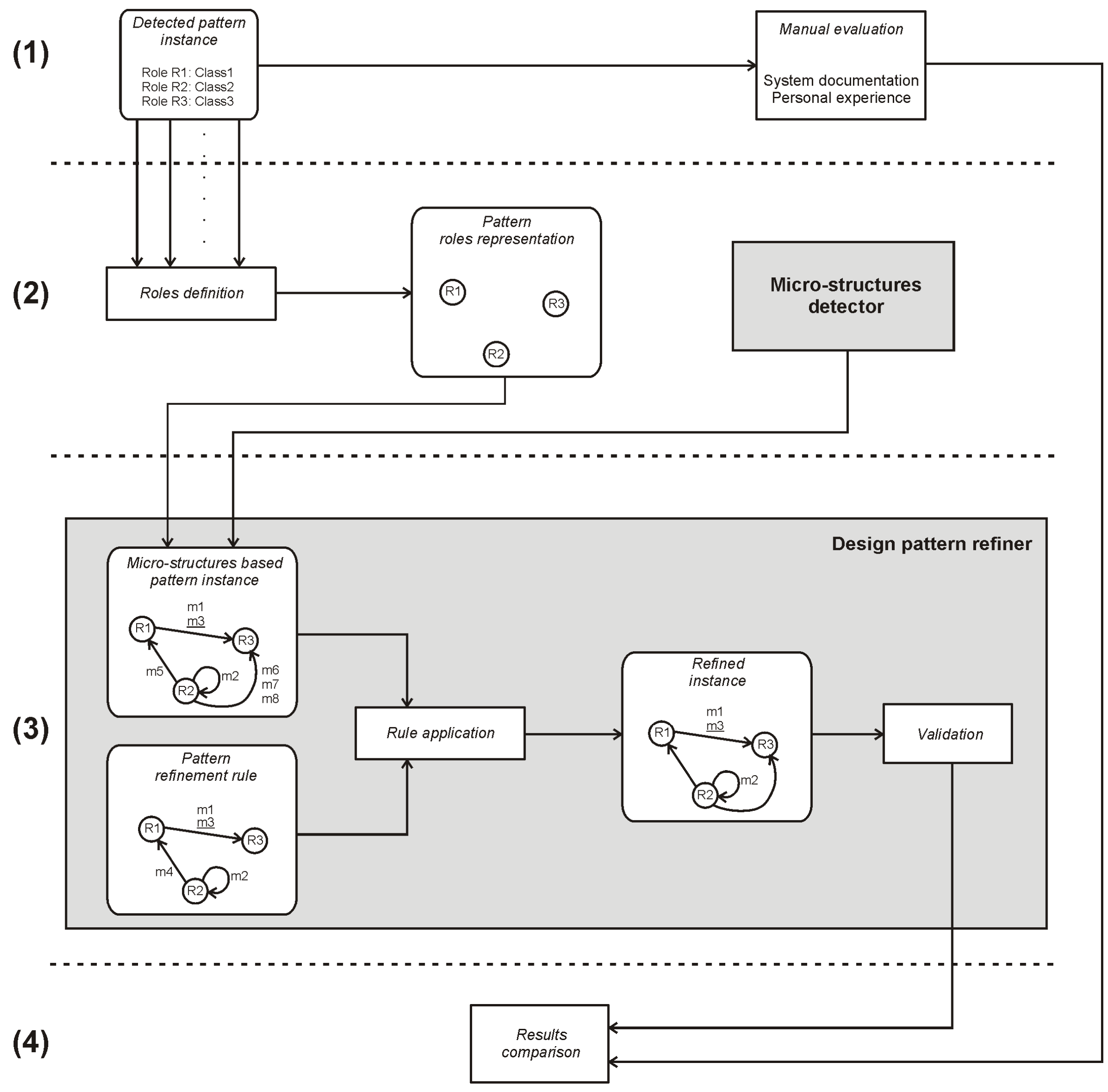


Figure 5.1 – *An overview of the refinement process*

Manual evaluation is a necessary operation, as it currently is the only way to verify the actual correctness of a pattern. For the future, the evaluation step could be supported by an automated comparison with a repository of valid instances (a work in progress with [ATZ08]).

The results obtained by the detection tools are represented in different forms, depending on the used tool. In general, the tools provide graphical or textual representations, where each role is associated with a particular class.
In order to be refined by the corresponding micro-structure-based rule, each instance must be defined in a graph form, where each node represents a role and each edge represents the set of micro-structures relating two roles. In the second phase of the refinement process we define the roles for each detected instance: each role identified by the tool is translated in 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 is to be completed with the actual class or classes playing that specific role. This is currently supported by a manual process, but we are working on the development of scripts that help in automating this process, at least for the most common tools for design pattern detection.
Considering the third phase, the defined graph nodes constitute the first input for the *Design pattern refiner*, a graphical front end devoted to the validation of pattern instances. 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 pattern instance: the roles in the graph are associated according to the micro-structures detected in the classes composing the instance under analysis. The DP refiner then applies the adequate refinement rule on each micro-structure-based instance, in order to check which of the micro-structures defined by the rule are actually implemented in the analyzed instance. Basing on this application, on the micro-structures peculiar for the pattern, and on the necessary pattern micro-structures, as defined in Section 5.2, in the validation step each instance is automatically accepted as a true pattern instance, or classified as a false positive and hence discarded.
In the fourth and final phase, the results are compared to the manual evaluation of the detected instances, in order to verify whether the refinement process provides the same results or not. At now, the comparison is manually performed. We plan to automate this phase in a future integration of the refinement process just described to the benchmark platform for design pattern detection evaluation we are currently developing [ATZ08].
As with our approach we actually "restrict" the set of patterns identified by each tool, one could argue that this is somehow equivalent to intersect the results provided by two different detection tools. However, this is not the case. In fact, the two tools may identify the same sets of false positives instances, that won't obviously be discarded by the intersection of the tools' results. On the other hand, our approach doesn't make such intersections, but considers the results of each tool singularly, trying to discard the false positives, which may not be avoided while matching the results provided by two different tools.

5.4. Application of the rules to the detected instances

Each of the instances detected by the four considered tools has been analyzed according to the rules defined in Section 5.2 and according to the process just presented, in order to check both their validity and the usefulness of the rules to validate the patterns.

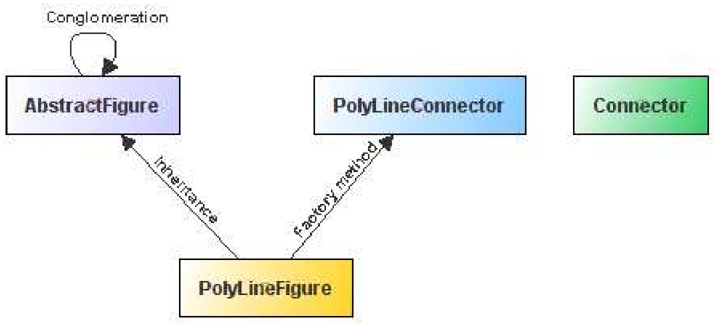
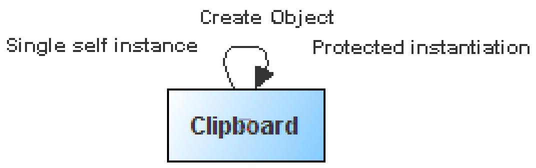
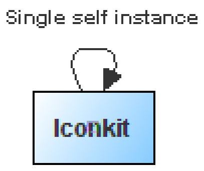
Pattern name	Detected instance	Refinement result and considerations
Factory method	DrawingView	In this instance, only the Creator role is present. The application of the Factory method rule to this instance does not validate it, as it lacks the remaining pattern roles and the fundamental micro-structures defined by the rule.
	 Conglomeration AbstractFigure PolyLineConnector Connector Inheritance Factory method PolyLineFigure	In this instance, `AbstractFigure` is the Creator, `PolyLineFigure` the Concrete creator, while `PolyLineConnector` is the Concrete product. This instance is validated by the rule, as the structural relationships among the role exist, and the necessary micro-structures for this pattern are implemented (the Factory method clue).
Singleton	 Create Object Single self instance Protected instantiation Clipboard	The `Clipboard` class has a single instance of itself and protected instantiation mechanisms to prevent the creation of `Clipboard` instances from other classes. Hence, `Clipboard` is a correct instance of the Singleton pattern and is validated by the rule.
		`Iconkit` only presents the Single self instance clue, which is a necessary condition for the existence of the pattern. Anyway, no protected instantiation mechanism is provided, therefore `Iconkit` is not to be considered a correct Singleton instance, as the Single self instance alone is not enough to grant the instance uniqueness property of this pattern.

***Table 5.5** – Application and results of the refinement process on sample creational design patterns instances*

We now provide some examples of the results obtained with the refinement process. 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. Table 5.5 reports some examples of the application of the refinement process on some instances of creational design patterns.
Table 5.6 describes examples of the application of the refinement process on instances of structural design patterns.

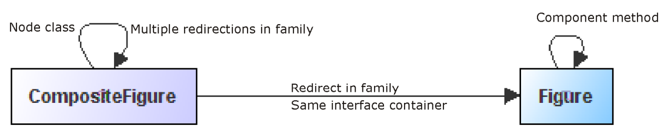
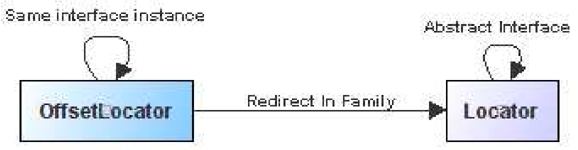
Pattern name	**Detected instance**	**Refinement result and considerations**
Composite	 Node class Multiple redirections in family Component method CompositeFigure Redirect in family Same interface container Figure	The two main roles for the Composite pattern have been identified: `Figure` is the Component (this is also verified by the presence of the Component method clue), and `CompositeFigure` is the the Composite class. The presence of the necessary Multiple redirections in family and Same interface container clues grant us that this is a valid instance of the pattern, correctly accepted by the refinement rule.
Decorator	 Same interface instance Abstract Interface OffsetLocator Redirect In Family Locator	In this instance of the Decorator pattern, `Locator` is the Component and `OffsetLocator` is the Decorator. These roles constitute the skeleton for the Decorator pattern according to the refinement rule, as no concrete roles have been detected. These two roles satisfy the constraints defined for them. This instance is a valid instance of the pattern, and the rule validate it.

***Table 5.6** – Application and results of the refinement process on sample structural design patterns instances*

Table 5.7 introduces examples of the application of the refinement process on instances of behavioural design patterns.

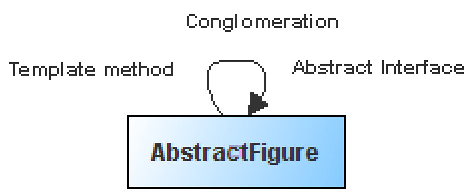
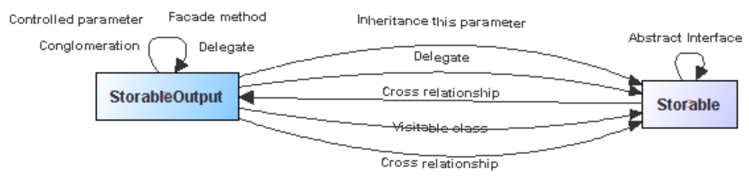
Pattern name	**Detected instance**	**Refinement result and considerations**
Template method	 Conglomeration Template method Abstract Interface AbstractFigure	In this instance of Template method, `AbstractFigure` is a correct Abstract class, as it presents all the elements that characterize this role, and especially the Template method clue. The tool wasn't able to detect a Concrete class for this instance, from a further analysis we identified class `PolyLineFigure` as a correct Concrete class, implementing the Template implementor clue. This instance is correct, and is validated by the rule.
Visitor	 Controlled parameter Facade method Conglomeration Delegate Inheritance this parameter Delegate Cross relationship Visitable class Cross relationship Abstract Interface StorableOutput Storable	In this instance of Visitor, `Storable` is the abstract Visitor class, while `StorableOutput` should be a Concrete element. This instance is not correct. 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, this implies that the abstract element (i.e. the root of the object structure) is not present too. Therefore, this instance cannot be considered correct, and the refinement rule refuses it.

Table 5.7 *– Application and results of the refinement process on sample behavioural design patterns instances*

5.5. Refinement results evaluation

The results of the refinement process applied to the instances detected by the four analyzed tools are reported in Tables 5.8 to 5.11. 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 a "not available" (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 available" (n/a) value is reported. Table 5.8 describes the refinement results on the instances detected by Design Pattern Detection Tool.

Design Pattern Detection Tool

Pattern category	*Pattern name*	*Detected instances*	*Correct instances*	*Validated*	*Precision before refinement*	*Precision 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%

Table 5.8 *– Results of the refinement process on the instances detected by 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 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 already captured by the rule. Table 5.9 reports the results obtained for PINOT.

PINOT

Pattern category	*Pattern name*	*Detected instances*	*Correct instances*	*Validated*	*Precision before refinement*	*Precision 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 5.9 *– Results of the refinement process on the instances detected by PINOT*

In this case, 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 5.10 reports the results obtained for FUJABA. 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 Design Pattern Detection tool: the pattern is too generic to be correctly refined by the rule.

FUJABA

Pattern category	*Pattern name*	*Detected instances*	*Correct instances*	*Validated*	*Precision before refinement*	*Precision 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 5.10 *– Results of the refinement process on the instances detected by FUJABA*

Finally, Table 5.11 indicates the results obtained for Web of Patterns.

Web of Patterns

Pattern category	*Pattern name*	*Detected instances*	*Correct instances*	*Validated*	*Precision before refinement*	*Precision 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

Table 5.11 *– Results of the refinement process on the instances detected by Web of Patterns*

In this case, the rule didn't succeed in accepting the correct Abstract factory instances, hence the precision rate decreased to 0%. The Template method instance has been correctly accepted, and the Adapter and Composite instances correctly discarded as false positives.

5.6. Concluding remarks

In this chapter we have presented an innovative approach to the refinement and validation of the results provided by the experimentation of common design pattern 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 experimentations, 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 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 obviously not intended to improve the recall of each single tool, as it is devoted uniquely to the analysis of already detected instances, and it doesn't

allow for the detection of further pattern instances in the subject systems. Moreover, at our knowledge no similar approach currently exists in the literature, confirming the novelty and originality of our work; therefore any comparisons with other works can’t be made.
In this chapter we have described and refined the results provided by four design pattern detection tools on the analysis of a single system (JHotDraw 6.0b1), in order to provide an exhaustive example and explanation of the refinement process.
For the future, we plan to extend our experimentations on the analysis of more systems, as well as on the analysis of repositories of design pattern instances. In this way, it will be interesting to integrate the refinement approach within the benchmark platform for design pattern detection evaluation [ATZ08]. In this way, the approach will be extendedly used on the results provided by more tools on the analysis of more systems, and will be useful to improve the comparisons among the instances detected by the different tools.

Chapter 6

Micro-structures for software architecture reconstruction

Abstract

In this chapter we investigate the usefulness of micro-structures for software architecture reconstruction activities. In particular, we focus on elemental design patterns and micro patterns as possible sources for the reconstruction of architectural information out of the analyzed systems. We indicate which are the artifacts that we want to generate, and we provide an evaluation between elemental design patterns and micro patterns, in order to understand which of them are more suitable for the generation of each artifact. Moreover, we introduce some structural and object-oriented antipatterns that can be detected inside a software system by analyzing the considered micro-structures.

6.1. Elemental design patterns and micro patterns for SAR purposes

One of the aims of this thesis is investigating the possibility to recover architectural information from the micro-structures that are identified within a subject system. To our knowledge, micro-structures have never been considered before for SAR activities in the literature. Indeed, they are able to capture structural relationships among the classes and modules composing a software system, and can be exploited to extract relevant structural information out of it. An example of this capability has been discussed in Chapter 4, where we underlined how EDPs are suitable to recover the structural relationships existing among pattern roles.
In this context, we are focused on the extraction of structural information basing on the analysis of the micro-structures detected in a target system. The type of information we consider is exclusively static. In fact, micro-structures are detected from source code analysis, and the behavior of the system is not considered while detecting these elements. Moreover, at now micro-structures don’t codify any behavioural information, so that

dynamic or behavioural system views or artifacts cannot be currently recovered with the adopted approach.
We are interested in recovering the following kind of information:

- *System views*: graphical views are considered as one of the best means to cope with the analysis of system complexity and to have a global understanding of it, without minding at its details, as also underlined by many experts in the SAR field (some of the definitions presented in Chapter 1 focused on this aspect). As discussed in Chapter 3, some micro-structures actually represent relationships between the entities composing a system. Therefore, they can be exploited in this sense, as they can be matched on the relationships that exist among classes, and can be consequently translated to graphical forms. The exploitation of micro-structures for the generation of views is introduced in Section 6.2.1;
- *Software metrics*: metrics are exploited both to understand the complexity of the analyzed systems and their overall quality and stability. We are interested in the computation of a set of metrics (discussed in Section 6.2.2) that can be derived from the analysis of the micro-structures detected in the analyzed systems;
- *Software antipatterns*: an antipattern is a software structure that, on the contrary of design patterns, seems to be an adequate solution to a certain design or programming issue, but it is actually far from the optimal practice. The presence of antipatterns inside an object-oriented system reflects in a system being not modular, far from the object-oriented best practices, and difficult to maintain and reuse. Hence, the identification of these structures inside a system helps in the detection of important critical components composing it, that can be seen by the engineers as the main candidates for possible refactoring or restructuring activities. The detection of a set of antipatterns is supported by both metrics analysis (discussed in Section 6.2.3), and by micro pattern analysis (discussed in Section 6.3.2);
- *Classes of particular interest*: there may be classes and modules inside a system that, besides their architectural and structural context, may present particular qualities that are relevant to be indicated. These entities are presented in Section 6.3.3.

These different kinds of information can all be obtained by the analysis and exploitation of the micro-structures detected in the subject systems by the Micro-structures detector module introduced in Chapter 3. In this way, a common source of information for both design pattern detection and refinement and for SAR activities is adopted. As a consequence, according to our approach it is not necessary to further inspect or analyze the target system, as all the necessary information for SAR is enclosed in the detected

micro-structures. This is a first step to have an integrated tool supporting both DPD and SAR activities.
Table 6.1 outlines the sets of EDPs and micro patterns that are used for the recovery of the architectural information, the computation of metrics and the detection of antipatterns and particular classes we are interested in. The motivations for the reported choices will be further detailed in the subsequent sections. In this process, DP clues are not considered, as they represent useful hints only for design pattern detection and refinement, and do not devise particular structural constraints besides those already derivable by the analysis of EDPs and micro patterns.
As we can notice from the table, only subsets of EDPs and of micro patterns are actually used to recover system architectural information. The elements that are not considered in these sets have been discarded either because the information they convey and that we consider useful is already embodied in some other element we actually consider, or else because in our opinion they do not represent any useful architectural or structural information at all. For example, even if we are able to detect the whole set of micro patterns from the source code of a system, indicating all of them in the output of the reconstruction process would obviously result in an excessive amount of information that is not granted to be relevant for the purposes of the reconstruction activity.

Artifact	**Exploited EDPs**	**Exploited micro patterns**
System views	Create object, retrieve, inheritance, delegate, redirect, revert method, extend method, delegate in family, redirect in family, delegate in limited family, redirect in limited family.	
Software metrics	Create object, retrieve, inheritance, delegate, redirect, revert method, extend method, delegate in family, redirect in family, delegate in limited family, redirect in limited family.	
Software antipatterns	Create object, retrieve, delegate, redirect, delegate in family, delegate in limited family, redirect in limited family.	Cobol like, pool, pseudo class, record.
Classes of particular interest		Function pointer, function object, immutable, canopy, data manager, sink, outline.

Table 6.1 *– EDPs and micro patterns for the obtainment of architectural and structural system information*

We have developed a module for software architecture reconstruction activities, that is devoted to the generation of views about the analyzed systems, the computation of metrics and the detection of antipatterns and of particularly interesting classes. The module is structured according to Figure 6.1.

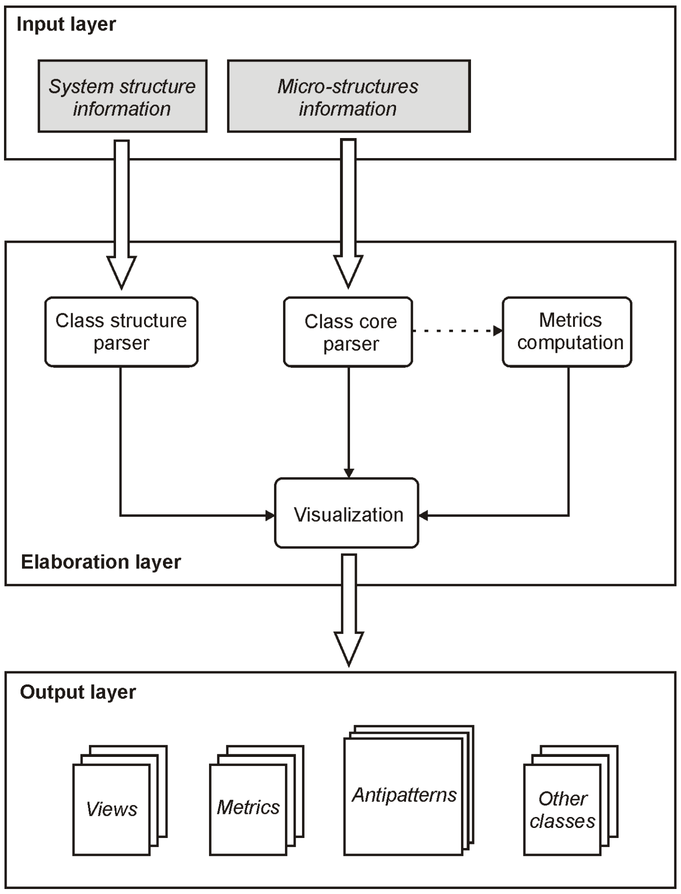


***Figure 6.1** – The architecture of the SAR module*

As many other tools for software architecture reconstruction and program understanding, our module presents a canonical three-layer architecture, which follows the extract-abstract-present model presented in Chapter 1 and described in Tilley [TPS96]. The *input layer* is formed by two kinds of input, represented by corresponding XML files, which are provided by the Micro-structures detector module described in Chapter 3. The *System structure information* input collects the whole set of packages and types, i.e. the classes and interfaces composing the subject system. For each type, the set of the defined methods and attributes is also specified. No information about the relationships among the various entities is reported in this input. Indeed, neither the second input contains this kind of information. In fact, the *Micro-structures information* just collects the whole set of micro-structures that have been detected in a system: for each type, it reports which DP clues, which EDPs and which micro patterns are implemented within it.

The structural relationships among classes and all the other functionalities resumed in Table 6.1 are all derived and computed by the *elaboration layer*, basing on the information provided by the two XML files just described (the analysis of the XML inputs is performed using the Apache XMLBeans technology [XML]). This layer is composed by four sub-modules. The *class structure parser* is devoted to the generation of an abstracted representation of the types (classes and interfaces, with their corresponding methods and attributes) and packages composing the system, basing on the information provided by the input layer. The *class core parser* analyzes the micro-structures detected from source code and consequently maps them on the relationships among classes. Recall that in this process only EDPs and micro patterns are considered.
The *metrics computation* sub-module computes common object oriented metrics basing on the relationships generated by the *class core parser*. The metrics that are currently computed will be introduced in Section 6.2.2. The *visualization* sub-module organizes the whole abstracted information generated by the *elaboration layer* in order to be exploitable by the end users. As far as the entities composing the analyzed system are concerned, packages and types will be represented in package or class views (introduced in Section 6.2.1) as graph nodes, while the relationships among them identified by the *class core parser* will be depicted as edges connecting them. The generated views are produced exploiting the functionalities provided by the JGraph libraries [JGraph]. The results are finally presented by the *output layer*, which provides the users with the set of generated structural views, the metrics computed on the system, and tags related to the identified antipatterns and other eventual interesting classes, in terms of the micro patterns representing them and shown in Table 6.1.
We now go further into details, explaining how the EDPs and the micro patterns are actually exploited to achieve the presented functionalities.

6.2. Elemental design patterns for SAR

Elemental design patterns are exploited for the generation of views about the analyzed systems, the computation of metrics and the detection of structural antipatterns. In the following sub-sections we present the views, the metrics and the antipatterns that it is possible to generate and calculate with the SAR module, and how the EDPs are exploited for their obtainment.

6.2.1. Views

The core concepts residing behind the views currently provided by the SAR module are the entities composing the system (i.e. packages, classes and interfaces), and the relationships connecting them with one another. While packages, classes and interfaces are derived by the *class structure parser* sub-module, analyzing the system structure information, the relationships among them are identified by the *class core parser*, analyzing the micro-structures information input. Through the analysis of the EDPs given as input it is possible to generate visual representations of the association, generalization and implementation relationships. Table 6.2 indicates the EDPs that are exploited in the generation of the available relationships.

Relationship	Object elements EDPs	Type relation EDPs	Method invocation EDPs
Association	Create object, Retrieve		Delegate, redirect, delegate in family, delegate in limited family, redirect in limited family
Generalization		Inheritance	Revert method, extend method
Implementation		Inheritance	

Table 6.2 *– The EDPs exploited for the generation of the relationships among packages and classes*

The main way to generate an association relationship is through a Create object EDP. Every time a realization of this EDP is encountered (i.e. in correspondence of every "`Object obj = new…`" statement), the class creating the instance establishes a connection to the class an instance of which is being created; hence, the source class is physically associating itself with the destination class. In the same way, the method invocation EDPs used for the generation of association relationships are those that imply the existence of a reference to another class. As far as the generalization and implementation relationships are concerned, they are obviously derived from the Inheritance EDP. Actually, this EDP doesn't specify whether the extended entity is a class (therefore a generalization relationship must be created) or an interface (leading to the creation of an implementation relationship). This distinction is obtained by analyzing the system structure information input, in which classes and interfaces are distinguished. Generalization relationships can also be generated by the analysis of the Revert method and Extend method EDPs, which imply the presence of a parent class for the class performing these kinds of method invocations. Also the Retrieve EDP is used to generate association relationships, as it

actually establishes a connection between two classes, where the first one retrieves a reference to a certain declared object from the second class.
Three different views are currently available. They are the *package view*, the *class compact view* and the *class extended view*.
The *package view* represents the packages composing a system and the relationships among them. It is used in order to obtain an immediate understanding of the dependencies among the various parts composing the subject system. Figure 6.2 reports the package view of JHotDraw 6.0b1.

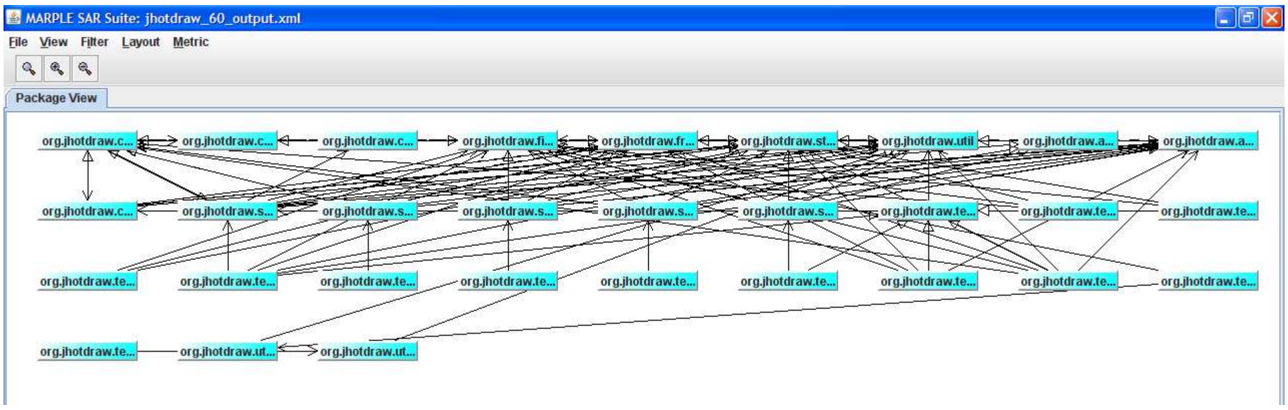


Figure 6.2 *– The package view of JHotDraw 6.0b1*

Between two related packages, only one relationship is graphically depicted. Anyway, many different actual relationships may subsist between them. A possible improvement will regard tagging each single relationship arrow with the number of actual relationships it resumes. Or else, it will be possible to color the arrows according to the number of relationships they enclose (for example assigning brighter colors to those arrows that represent a larger number of relationships). The user can also filter the shown relationships allowing the tool to display only associations, generalizations, implementations or a combination of them.
The *class compact view* reports a class diagram about a particular package, with all the classes and interfaces composing it and the relationships among them. A sample view is reported in Figure 6.3, showing the classes composing the `org.jhotdraw.figures` package of JHotDraw 6.0b1.

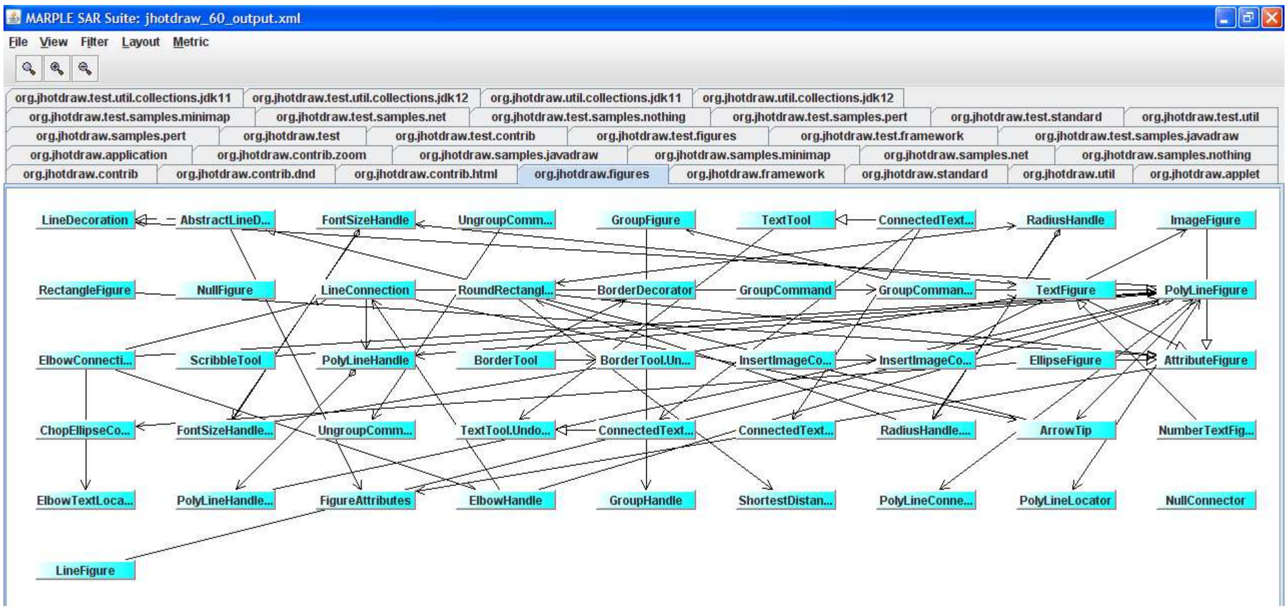


Figure 6.3 – *The class compact view for package org.jhotdraw.figures*

The user can choose the package to be inspected selecting the corresponding tab panel. Actually, this view can be quite overwhelmed, while dealing with packages having a high number of classes and relationships among them. As for the package view, the users may filter the relationships through the Filter menu, in order to show only associations, generalizations, implementations or a combination of them. Anyhow, in order to provide a more effective navigation through the classes composing a package, the *class extended view* (shown in Figure 6.4) has been introduced. In this view, each class is shown detached from the other ones. Hence, the software engineer can focus on single classes, managing only the relationships it has with the other classes, without minding globally at the rest of the package and consequently avoiding the confusion that may arise from the presence of a huge number of classes and interrelationships.

6.2.2. Metrics

Four main metrics are computed in the SAR module. They are:

- *Local dependencies*: given a type (either a class or interface), the local dependencies of this type is the number of types this type depends on, within the same package. It is obtained by counting the number of associations going out from the subject type to types belonging to the same package;
- *Local dependents*: given a type, the local dependents of this type is the number of types that depend on the functionalities provided by the subject type, within the same

package. It is obtained by counting the number of associations coming in the considered type from types belonging to the same package;

- *External dependencies*: given a type, the external dependencies of this type is the number of types this class depends on, considering the overall system and not the package the subject type is contained in. It is derived by counting the number of associations going out from the type to types belonging to different packages;
- *External dependents*: given a type, the external dependents of this type is the number of types that depend on the functionalities provided by the subject type, but not belonging to the same package. It is computed by counting the number of associations coming in the considered type from types belonging to packages different from the package the subject type is contained in.

Dependencies and dependents can be related to packages as well. In this case, only the global case is considered, as a package can only expose relationships with other (external) packages. Therefore the dependencies of a package are the number of packages the subject package depends on, while the dependents of a package are the number of packages that depend on the functionalities provided by the subject package.

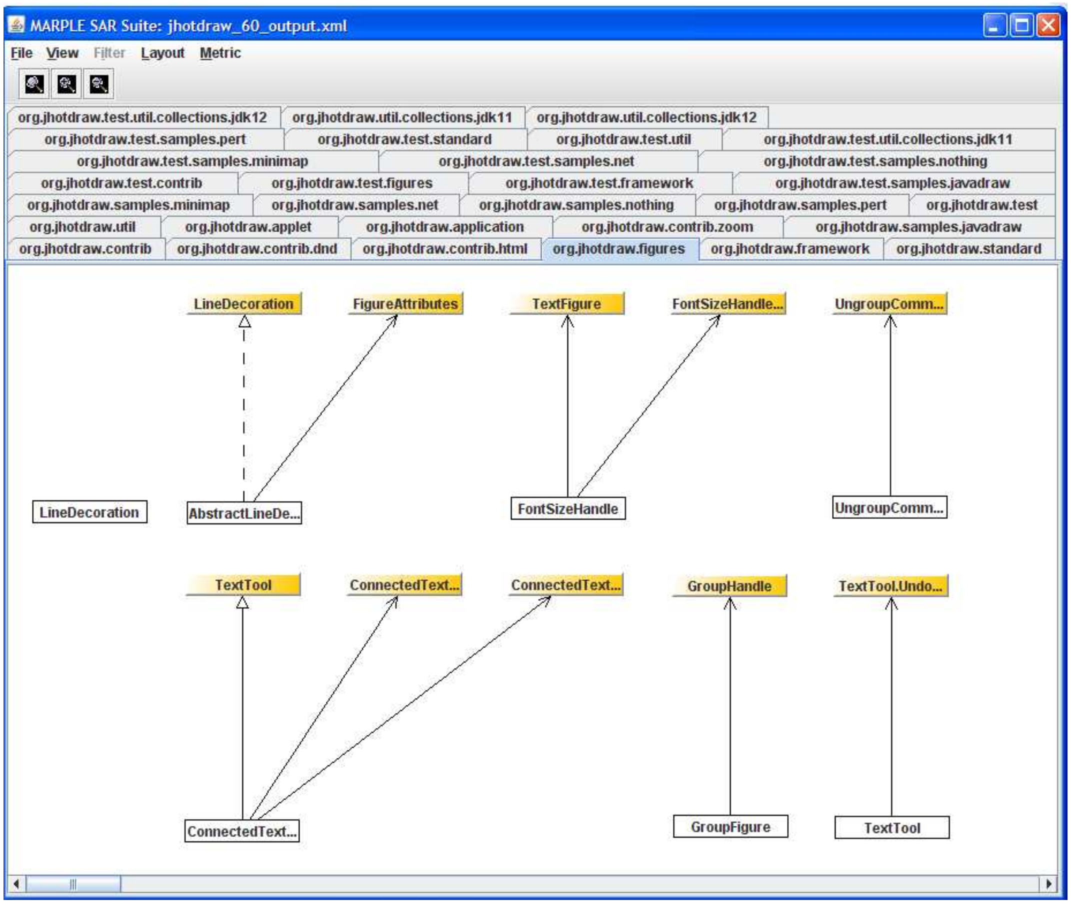

MARPLE SAR Suite: jhotdraw_60_output.xml
File
View
Filter
Layout
Metric
org.jhotdraw.test.util.collections.jdk12
org.jhotdraw.util.collections.jdk11
org.jhotdraw.util.collections.jdk12
org.jhotdraw.test.samples.pert
org.jhotdraw.test.standard
org.jhotdraw.test.util
org.jhotdraw.test.util.collections.jdk11
org.jhotdraw.test.samples.minimap
org.jhotdraw.test.samples.net
org.jhotdraw.test.samples.nothing
org.jhotdraw.test.contrib
org.jhotdraw.test.figures
org.jhotdraw.test.framework
org.jhotdraw.test.samples.javadraw
org.jhotdraw.samples.minimap
org.jhotdraw.samples.net
org.jhotdraw.samples.nothing
org.jhotdraw.samples.pert
org.jhotdraw.test
org.jhotdraw.util
org.jhotdraw.applet
org.jhotdraw.application
org.jhotdraw.contrib.zoom
org.jhotdraw.samples.javadraw
org.jhotdraw.contrib
org.jhotdraw.contrib.dnd
org.jhotdraw.contrib.html
org.jhotdraw.figures
org.jhotdraw.framework
org.jhotdraw.standard
LineDecoration
FigureAttributes
TextFigure
FontSizeHandle...
UngroupComm...
LineDecoration
AbstractLineDe...
FontSizeHandle
UngroupComm...
TextTool
ConnectedText...
ConnectedText...
GroupHandle
TextTool.Undo...
ConnectedText...
GroupFigure
TextTool

***Figure 6.4** – The class extended view for package org.jhotdraw.figures*

These metrics are used in different approaches and tools for software architecture reconstruction as for example [SA4J, JDepend]. Further analyzed (as we will see later on), they are well established means to assert the quality of a system in terms of its stability, cohesion, and ease of reuse. As these metrics consider the number of associations related to each single class, and as associations are derived by analyzing the EDPs characterizing each class, we can state that these metrics are derived by EDPs as well, without further reasoning about the subject system.
Dependencies and dependents can be considered as a first mean to understand the complexity of a system. A high quality system must pursue the "high cohesion – low coupling" principle [Lar04]. The number of dependencies of a class is to be considered as an indication of the level of coupling of each single class. Classes with a high number of dependencies consequently augment the coupling of the system, worsening its overall quality. In the same way, the number of dependencies of a package can be seen as an indication of the cohesion of the same package. The fewer dependencies the package has with the rest of the system, the more cohesive the package is, consequently improving the quality of the system. On the other hand, the number of dependents of a class gives an overview of how many classes in the system are affected if the subject class is changed. Dependencies and dependents are also strictly related to the identification of structural antipatterns, as discussed in Section 6.2.3.
We shall make an important clarification. The meaning of "local" and "external" within MARPLE SAR is different from that adopted for example by SA4J. In MARPLE SAR, the local relationships of a type are all those relationships that involve the type itself and only types belonging to the same package. On the other hand, external relationships regard the types declared in other packages which are related with the subject type.
SA4J considers local and global relationships. The local relationships of a type involve all the *immediate* dependencies and/or dependents of the type itself, no matter the package they belong to. The global relationships of a type are related to all the *non-immediate* entities related to the subject type. Figure 6.5 depicts two sample packages and possible relationships among their classes. Table 6.3 indicates which are the local, external or global dependencies and dependents detected by MARPLE SAR and SA4J for each of the considered types. As it can be noticed from the table, the relationships identified by SA4J are more complex than those considered by MARPLE SAR. As an example, class 4 of Package_1 doesn't have any external dependents according to MARPLE SAR, while it has five global dependents belonging to both packages according to SA4J.

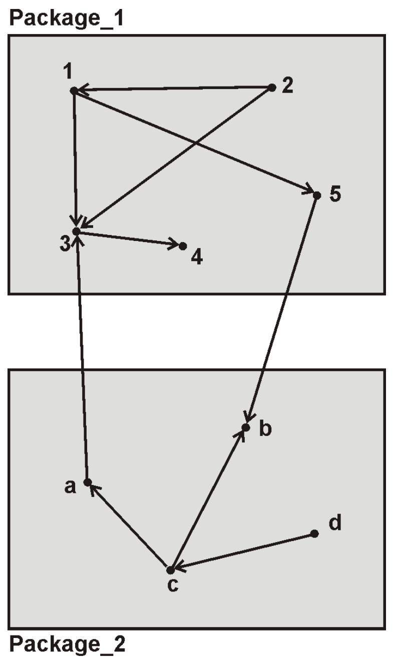


Figure 6.5 - *Two sample packages*

	MARPLE SAR				Structural Analysis for Java			
Class	*Loc. D.cies*	*Loc. D.ents*	*Ext. D.cies*	*Ext. D.ents*	*Loc. D.cies*	*Loc. D.ents*	*Glob. D.cies*	*Glob. D.ents*
1	**3, 5**	**2**	**/**	**/**	**3, 5**	**2**	**4, b**	**/**
2	**1, 3**	**/**	**/**	**/**	**1, 3**	**/**	**3, 4**	**/**
3	**4**	**1, 2**	**/**	**a**	**4**	**1, 2, a**	**/**	**2, c, d**
4	**/**	**3**	**/**	**/**	**/**	**3**	**/**	**1, 2, a, c, d**
5	**/**	**1**	**b**	**/**	**b**	**1**	**/**	**2**
a	**/**	**c**	**3**	**/**	**3**	**c**	**4**	**d**
b	**/**	**c**	**/**	**5**	**/**	**5**	**/**	**d, 1, 2**
c	**a, b**	**d**	**/**	**/**	**a, b**	**d**	**3, 4**	**/**
d	**c**	**/**	**/**	**/**	**c**	**/**	**a, 3, 4**	**/**

Table 6.3- *The local, external and global dependencies and dependents according to MARPLE SAR and SA4J*

Due to the adopted interpretation of the "local" and "global" concepts, SA4J doesn't allow the user to distinguish immediately intra-package from inter-package relationships, which is on the contrary possible with MARPLE SAR. Having a strong and clear distinction between these two kinds of relationships lets the users evaluate the cohesion and coupling within single packages, as well as about the overall system.
Another fundamental metric that can be computed is *abstractness* [Mar95], i.e. the amount of abstract classes and interfaces inside a package with respect to the total number of types composing it. It can be evaluated by considering the Abstract interface EDP, which indicates that inside a given class an abstract class method or an interface method is declared, hence the declaring type is consequently an abstract class or an interface. Packages with high abstractness values are easily extensible and reusable by other parts or modules of the system.
Starting from these basic five metrics (local and external dependencies, local and external dependents, and abstractness), five other metrics (for a total of ten metrics) can be computed and derived, both on classes and on packages.

The metric that can be calculated on classes is:

- *Belonging* [SA4J]: it represents how much a class is being used by its package, dividing the number of local dependencies and dependents by the overall number of dependencies and dependents of the class, considered both at the local and at the external level. If it equals to 1, the class is completely used and referenced within its package, as it doesn't have any external dependencies or dependents;

The metrics that can be obtained on packages are:

- *Instability* [Mar95]: it indicates how much the classes are linked to their package, and it is obtained by dividing the number of external dependencies by the number of external dependencies and external dependents.
  This metric is an indicator of the package's resilience to change. A value of zero indicates a completely stable package (as its classes don't refer to classes belonging to other packages, therefore the package is completely self-contained) and a value of one indicates a completely instable package (as its classes only refer to external types);
- *Distance from the main sequence* [Mar95]: abstractness (A) and instability (I) are strictly related metrics. Given the graph depicted in Figure 6.6, two core categories of packages can be identified: those being totally composed by abstract entities and stable (represented by the (0, 1) point in the diagram), and those that contain only concrete entities and are completely instable (represented by the (1, 0) point in the diagram).

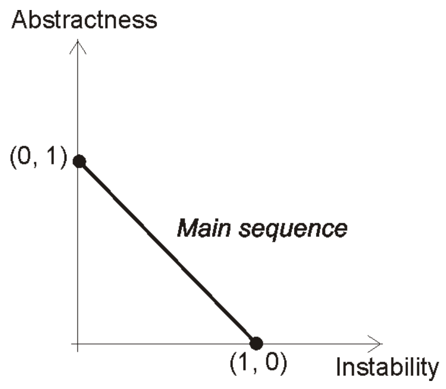


***Figure 6.6** – The relationship between abstractness and instability: the main sequence*

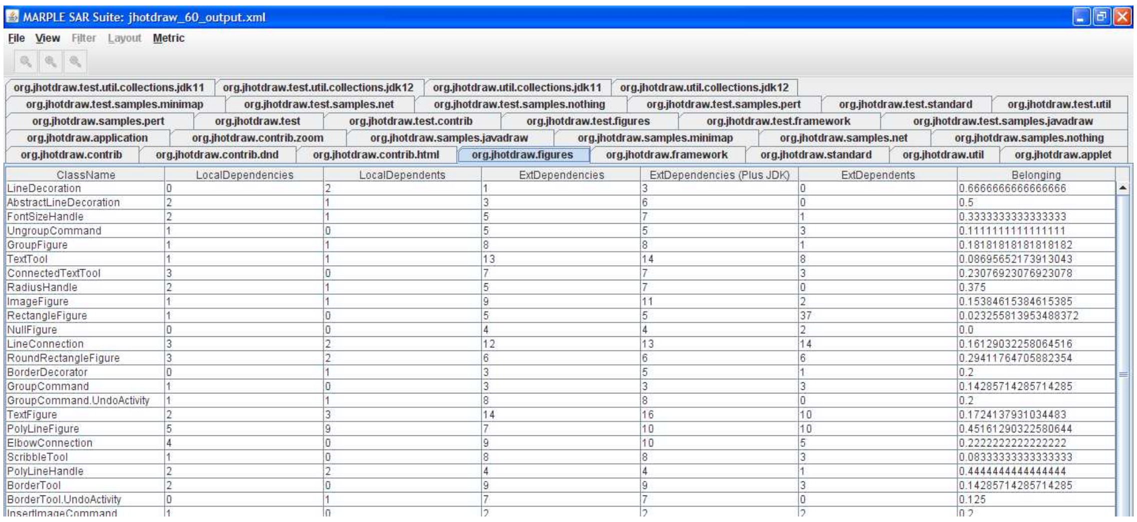
Obviously, not all of the packages of a system can belong to one of these two positions, as they generally have different degrees of abstractness and instability values.
For example, a package with A = 0 and I = 0 is highly stable and totally concrete. Such packages are not desirable, as they are rigid, hence they cannot be extended as they are not abstract. They are also difficult to change, due to their stability.
Packages with A = 1 and I = 1 are not desirable as well, as they are totally abstract, but with no dependents, hence the abstractions are impossible to be extended.
A package with A = 0.5 and I = 0.5 is partially extensible and partially stable, so that the extensions are not subject to maximal instability. Martin states that the package

stability is *in balance* with its abstractness. In Figure 6.5, the line connecting the (0, 1) and (1, 0) points represents those packages whose abstractness is balanced with stability. This line is called the *main sequence.* As it is desirable for packages being as close as possible to the main sequence, Martin defines the (normalized) *distance from the main sequence* as $D = |\text{Abstractness} + \text{Instability} - 1|$. Values for this metric range in the interval [0, 1]. The more a package has a D value close to zero, the more it is near to the main sequence and hence well balanced. The engineers can therefore focus on those packages with a D value not near to zero, as they are the first candidates to be reanalyzed and restructured.

- *Bonding* [SA4J]: it indicates how well the classes within the package are connected with one another, and can be obtained by dividing the number of local dependencies by the total number of dependencies, both local and external.

  This metric gives an idea of how much a certain class exploits the functionalities provided by the other classes belonging to the same package: if it equals to 1, the class lends itself only on classes of the same package.
- *Link density* [SA4J]: it indicates the mean number of relationships among classes within the package, giving an indication of how strong these relationships are; it is obtained dividing the number of local dependencies and dependents by the total number of types contained in the package.

Figure 6.7 reports a sample of the metrics computed through MARPLE SAR on the classes belonging to JHotDraw's `org.jhotdraw.figures` package. For a focused exploitation, users may right click on a package or a type and evaluate the metrics only for that entity.


MARPLE SAR Suite: jhotdraw_60_output.xml
File
View
Filter
Layout
Metric
org.jhotdraw.test.util.collections.jdk11
org.jhotdraw.test.util.collections.jdk12
org.jhotdraw.util.collections.jdk11
org.jhotdraw.util.collections.jdk12
org.jhotdraw.test.samples.minimap
org.jhotdraw.test.samples.net
org.jhotdraw.test.samples.nothing
org.jhotdraw.test.samples.pert
org.jhotdraw.test.standard
org.jhotdraw.test.util
org.jhotdraw.samples.pert
org.jhotdraw.test
org.jhotdraw.test.contrib
org.jhotdraw.test.figures
org.jhotdraw.test.framework
org.jhotdraw.test.samples.javadraw
org.jhotdraw.application
org.jhotdraw.contrib.zoom
org.jhotdraw.samples.javadraw
org.jhotdraw.samples.minimap
org.jhotdraw.samples.net
org.jhotdraw.samples.nothing
org.jhotdraw.contrib
org.jhotdraw.contrib.dnd
org.jhotdraw.contrib.html
org.jhotdraw.figures
org.jhotdraw.framework
org.jhotdraw.standard
org.jhotdraw.util
org.jhotdraw.applet

ClassName	LocalDependencies	LocalDependents	ExtDependencies	ExtDependencies (Plus JDK)	ExtDependents	Belonging
LineDecoration	0	2	1	3	0	0.6666666666666666
AbstractLineDecoration	2	1	3	6	0	0.5
FontSizeHandle	2	1	5	7	1	0.3333333333333333
UngroupCommand	1	0	5	5	3	0.1111111111111111
GroupFigure	1	1	8	8	1	0.18181818181818182
TextTool	1	1	13	14	8	0.08695652173913043
ConnectedTextTool	3	0	7	7	3	0.23076923076923078
RadiusHandle	2	1	5	7	0	0.375
ImageFigure	1	1	9	11	2	0.15384615384615385
RectangleFigure	1	0	5	5	37	0.023255813953488372
NullFigure	0	0	4	4	2	0.0
LineConnection	3	2	12	13	14	0.16129032258064516
RoundRectangleFigure	3	2	6	6	6	0.29411764705882354
BorderDecorator	0	1	3	5	1	0.2
GroupCommand	1	0	3	3	3	0.14285714285714285
GroupCommand.UndoActivity	1	1	8	8	0	0.2
TextFigure	2	3	14	16	10	0.1724137931034483
PolyLineFigure	5	9	7	10	10	0.45161290322580644
ElbowConnection	4	0	9	10	5	0.2222222222222222
ScribbleTool	1	0	8	8	3	0.08333333333333333
PolyLineHandle	2	2	4	4	1	0.4444444444444444
BorderTool	2	0	9	9	3	0.14285714285714285
BorderTool.UndoActivity	0	1	7	7	0	0.125
InsertImageCommand	1	0	2	2	2	0.2

Figure 6.7 – *The metrics computed on the classes of a JHotDraw package*

Many other metrics could actually be calculated following our approach based on micro-structures. Anyway, we decided to focus on this set of metrics for different reasons. First of all, we are interested in pursuing activities related to the architecture of a system. Basic object-oriented metrics, like for example the number of attributes (NOA) or methods (NOM) of a class [LK94], if considered stand-alone, don't provide very interesting information about the architecture of a system and its modules. This because these metrics are generally focused in evaluating the characteristics of single classes, and don't consider the system (or part of it) in its overall structure. To obtain some usable information about the structure of a system they need to be combined with other metrics, and adequately analyzed and interpreted [KB04].

On the other hand, the metrics we consider (the dependencies, dependents and their derived metrics) are in their nature focused on the structure of a system, as they are computed on single types or packages, but depending on the entities connected to these types or packages. They automatically provide a sort of structured information that can be more easily exploitable and can be more useful during the evaluation of the architecture of a software system.

6.2.3. Structural antipatterns

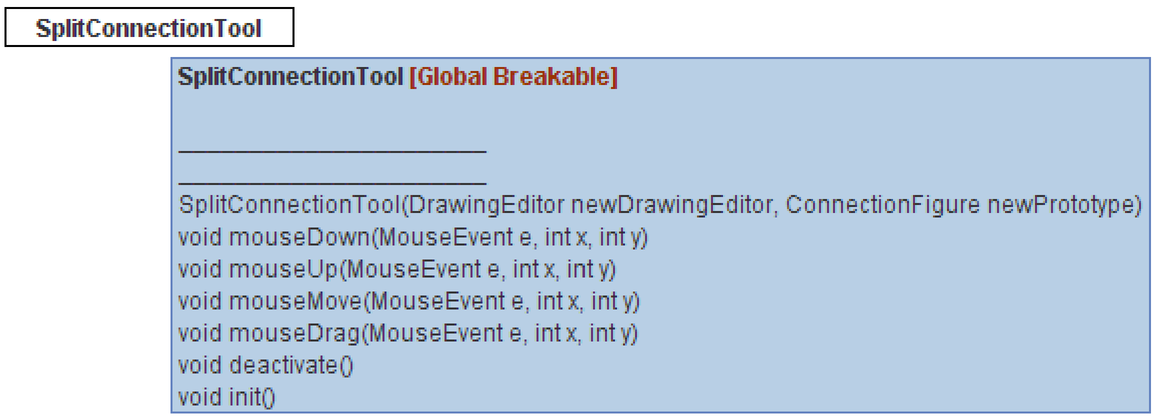
Given the number of local and external dependencies and dependents, six structural antipatterns [SA4J] can be identified:

- *Local breakable*: a local breakable is a class with many local dependencies. Local breakables have excessive responsibility within the system, and can be typically recognized by the presence of many long methods (even if the local dependencies metric can be used as well). The presence of breakables makes the code very difficult to understand, to maintain, and to reuse;
- *Global breakable*: a global breakable is a type that is often affected when any other entity within the system is changed, due to the high number of external dependencies it has. Global breakables are to be avoided, as they indicate fragility and lack of modularity in the system;
- *Local butterfly*: local butterfly is a type that has many local dependents. If a local butterfly is changed, these changes often have an important impact on the rest of the package. Hence, local butterflies are allowed only for either basic system interfaces or utility classes;
- *Global butterfly*: a global butterfly is a type with many global dependents. If a global butterfly is changed, this produces heavy consequences on the rest of the system.

Therefore, as in the local case, global butterflies should only be either basic system interfaces or utility classes;

- *Local hub*: a local hub is a type that has many immediate dependencies and many immediate dependents. Therefore, it is both a local breakable and a local butterfly at the same time. Local hubs have too many responsibilities within the system, and also serve as utility components. Hubs make the code difficult to understand, to maintain, and to reuse, and they also make the code itself unstable;
- *Global hub*: a global hub is a type with many global dependencies and many global dependents. Therefore, it is both a global breakable and a global butterfly. If a modification within a system occurs, a global hub is often consequently affected. Being a global butterfly, it also affects a significant part of the system if it changes. Global hubs indicate fragility and lack of modularity in the system.

A class is considered a breakable (resp. butterfly or hub) if it has at least ten dependencies (resp. dependents or both) with other classes. A further improvement will consider the number of dependencies and dependents with respect to the number of types composing the single package or the global system. In fact, it seems sensible to assert that a class belonging to a package containing, for example, ten classes, which presents ten local dependencies is far more critical than a class having the same ten local dependencies, but spread in a larger package, with for example a hundred or more classes.
Just as an example, Figure 6.8 reports a sample global breakable class, identified in JHotDraw 6.0b1. The detection of these structural antipatterns helps the engineers to identify the components of a system that are critical in terms of their structure, i.e. in terms of their number of outgoing and incoming relationships with the rest of the system.

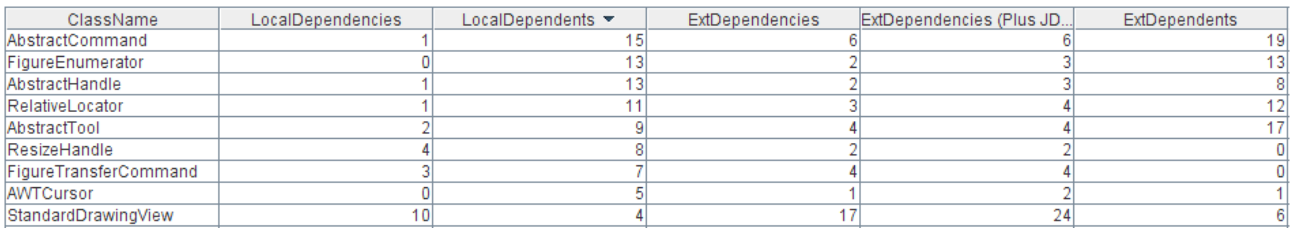


```
SplitConnectionTool [Global Breakable]

SplitConnectionTool(DrawingEditor newDrawingEditor, ConnectionFigure newPrototype)
void mouseDown(MouseEvent e, int x, int y)
void mouseUp(MouseEvent e, int x, int y)
void mouseMove(MouseEvent e, int x, int y)
void mouseDrag(MouseEvent e, int x, int y)
void deactivate()
void init()
```

***Figure 6.8** – A global breakable class, detected in JHotDraw 6.0b1*

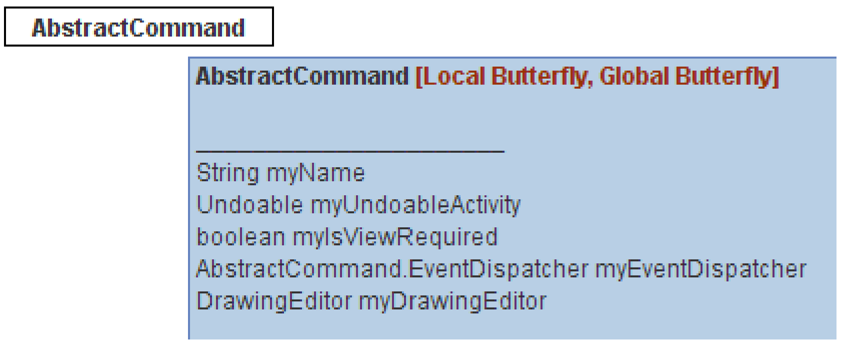
These complex components are to be considered as the first candidates for a structural refactoring. Re-engineering these entities results in having a more stable and self contained system. Currently, the tool doesn't provide a functionality to get all the detected structural antipatterns at a glance. Anyway, the user may refer to the metrics tables related to classes, individuate those classes presenting a high number of dependents and dependencies, and consequently focus on them. As an example, Figure 6.9 report the metrics computed on some classes belonging to the `org.jhotdraw.standard` package.



ClassName	LocalDependencies	LocalDependents ▾	ExtDependencies	ExtDependencies (Plus JD...	ExtDependents
AbstractCommand	1	15	6	6	19
FigureEnumerator	0	13	2	3	13
AbstractHandle	1	13	2	3	8
RelativeLocator	1	11	3	4	12
AbstractTool	2	9	4	4	17
ResizeHandle	4	8	2	2	0
FigureTransferCommand	3	7	4	4	0
AWTCursor	0	5	1	2	1
StandardDrawingView	10	4	17	24	6

Figure 6.9 *– Metrics computed on some classes belonging to the org.jhotdraw.standard package*

Consider the `AbstractCommand` and `StandardDrawingView` classes. According to the antipatterns definitions provided before and to the dependencies and dependents values obtained on them, they respectively should be a local and global butterfly, and a local and global breakable. This can be verified by analyzing the Class extended view, where these classes are correctly tagged with the corresponding antipatterns (Figures 6.10 and 6.11).

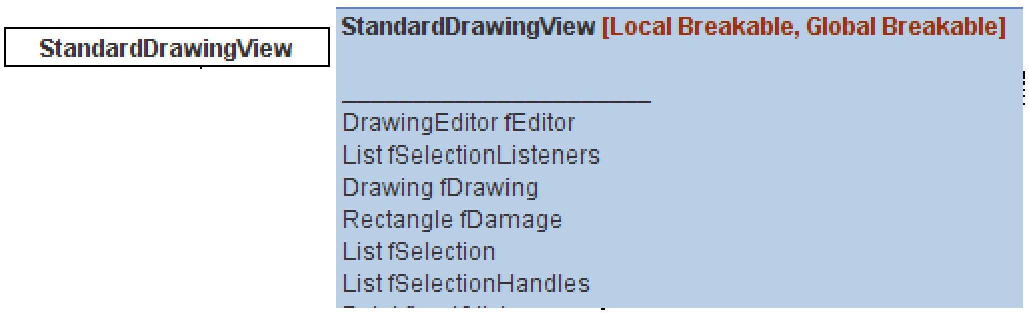


AbstractCommand

AbstractCommand [Local Butterfly, Global Butterfly]

String myName
Undoable myUndoableActivity
boolean myIsViewRequired
AbstractCommand.EventDispatcher myEventDispatcher
DrawingEditor myDrawingEditor

Figure 6.10 *– The AbstractCommand class, instance of the local and global butterfly antipatterns*



StandardDrawingView

StandardDrawingView [Local Breakable, Global Breakable]

DrawingEditor fEditor
List fSelectionListeners
Drawing fDrawing
Rectangle fDamage
List fSelection
List fSelectionHandles

Figure 6.11 *– The StandardDrawingView class, instance of the local and global breakable antipatterns*

6.3. Micro patterns for SAR

In the context of software architecture reconstruction, micro patterns cannot be used to identify structural relationships among the classes or entities composing a system. As we have described in Chapter 3, micro patterns are focused on the characteristics and properties of single classes, with particular emphasis on the properties of their attributes and methods. As breakables, butterflies and hubs identify single classes needing restructuring, in the same way the detection of micro patterns can be exploited to identify classes of particular interest within the system, and classes representing possible class-level antipatterns. In the following sub-sections we will discuss the micro patterns that are considered in the SAR module, motivating for their detection and their importance for SAR activities.

6.3.1. Micro patterns identifying classes of particular interest

Through the use of the Micro-structures detector, we are able to detect the whole set of micro patterns by analyzing the source code of the subject systems. For SAR purposes, we consider only a subset of the micro patterns that let the identification of types of particular interest possible. The considered micro patterns are Function pointer, Function object, Immutable, Canopy, Data manager, Sink and Outline. We now motivate for their consideration and for their importance for the reconstruction of software architectures. For some micro patterns, a direct correspondence with the values assumed by the dependencies and dependents of classes has been noticed. We will indicate and motivate the results on the experimentations pursued on JHotDraw 6.0b1.

Function pointer
Definition: Function pointer classes are those classes presenting only one public instance method, and no fields. They represent the equivalent of a function pointer in a procedural programming language, and can therefore be used to make an indirect polymorphic call to that method.
Relevance for the detection: we can state that these classes play a limited role in the architecture as classes themselves, as they are not characterized by any state, due to the lack of fields. They can be considered as a filter on the set of classes composing a system: as they don't play any particular role within the architecture, the engineers can concentrate on other parts of the system.

Function object

Definition: Function object classes have a single public instance method, but, differently from the function pointers, they actually have a state, represented by a set of fields. Therefore, instances of the Function object micro pattern can store parameters to the main method of the class.

Relevance for the detection: it has been experimented [GM05] that this micro pattern matches many classes that are event handlers, passed as callback hooks in, for example, the AWT and Swing libraries. The identification of instances of this micro pattern helps the engineer to identify these peculiar classes, allowing to deal with the event-handling classes of the system.

Immutable

Definition: an Immutable class is a class whose fields are only changed by its constructors, therefore only once.

Relevance for the detection: *t*his micro pattern is considered relevant as it establishes a strong condition on the fields assignment: as they can only be changed by the constructors, the declaring classes have a limited impact with the rest of the system. However, no direct correlation with the dependencies and dependents of Immutable classes has been noticed, as also demonstrated by the sample results provided in Table 6.4. Values for dependencies and dependents vary among the Immutable classes detected in the considered package, not differently from what happens while considering another set of classes not implementing the Immutable micro pattern.

Class	Local dependencies	Local dependents	External dependencies	External dependents
GroupFigure	1	1	8	1
ElbowConnection	4	0	10	5
PolyLineHandle	2	2	4	1
InsertImageCommand	1	0	2	2
NumberTextFigure	1	0	1	2
LineFigure	1	0	0	7

Table 6.4 *– Relationships between sample instances of the Immutable micro pattern and the dependencies/dependents values*

Canopy

Definition: a Canopy is a class with one instance field that can be changed only by the class constructors.

Relevance for the detection: the same considerations traced for the Immutable micro pattern can be applied also to the Canopy micro pattern, also as far as dependencies and dependents metrics are concerned.

Data manager

Definition: a Data manager class is a class whose methods are all setters or getters.

Relevance for the detection: the detection of these classes allows for the identification of classes whose objective is exclusively being a repository for data and managing these data. The detected Data manager instances are characterized by a low number of dependencies, as shown in Table 6.5. This indicates that these classes are generally self-contained, without the need of making references to other classes.

Class	Local dependencies	Local dependents	External dependencies	External dependents
FastBufferedUpdateStrategy	0	0	7	1
JHotDrawException	0	0	1	0
JHotDrawRuntimeException	0	1	1	10
Clipboard	0	0	0	3
WindowMenu.ChildMenuItem	0	2	1	0
CTXWindowMenu.ChildMenuItem	0	2	1	0
CommandCheckBox	1	1	2	1
DesktopEvent	0	1	1	1

Table 6.5 *– Relationships between sample instances of the Data manager micro pattern and the dependencies/dependents values*

Sink

Definition: a Sink is a class whose methods do not propagate any call to any other method.

Relevance for the detection: the ambit of these classes is limited, they usually have a low number of dependencies, but may have a large number of dependents.

Class	Local dependencies	Local dependents	External dependencies	External dependents
NullTool	1	0	0	2
AWTCursor	0	5	2	1
NullPainter	0	0	1	0
FigureChangeAdapter	0	0	1	2
ColorEntry	0	1	0	0
PaletteIcon	0	0	1	2
ResourceManagerNotSetException	0	1	1	0

Table 6.6 *– Relationships between sample instances of the Sink micro pattern and the dependencies/dependents values*

The low number of dependencies has also been demonstrated in practice, as it can be noticed by the results provided in Table 6.6. As it can be noticed, the maximum number of local dependencies detected on a Sink class is 1. This occurred only for one class, while for the remaining instances the number of local dependencies is zero.

Outline

Definition: an Outline class is a class for which at least one method invokes an abstract method declared in the same class.

Relevance for the detection: as the name suggests, their aim is to give an outline to a particular algorithm, specifying its main operations without going in the details of their implementation. This is also the aim of the Template method design pattern [GHJV94]: the Outline micro pattern can be considered a hint for its detection. The existence of Outline classes let assume that there exist subclasses extending it, and therefore better specifying the algorithm implementation by overriding the abstract method. This has been practically demonstrated on JHotDraw 6.0b1, as reported in Table 6.7, where we can notice that each Outline instance presents at least one local or external dependent.

Class	Local dependencies	Local dependents	External dependencies	External dependents
AbstractLineDecoration	2	1	6	0
ActionTool	1	0	3	1
AbstractFigure	4	3	9	24
ChangeConnectionHandle	3	2	9	0
ChangeConnectionHandle.UndoActivity	0	3	4	0
AutoscrollHelper	0	0	0	2
DNDHelper	2	2	2	1

Table 6.7 *– Relationships between sample instances of the Outline micro pattern and the dependencies/dependents values*

The existence of dependents for classes implementing the Outline micro pattern allows us to make an interesting consideration. Outline classes are necessarily abstract classes, by the same definition of the Outline micro pattern. Having dependents for such classes grants that an implementation of the abstract method is provided, consequently also obtaining a correct implementation of the template method pattern. The consideration could be extended to abstract classes and interfaces in general. It is desirable that these types present at least one dependent type. This would grant us that abstract classes and interfaces are actually used within the system, providing a mean for their extension.

There are two reasons for which the remaining micro patterns are not considered for SAR. First of all, they may capture information that is already captured (even if in different forms) by EDPs. Or else, they represent types that are of scarce interest from a structural point of view, with no very peculiar characteristics.

6.3.2. Object-oriented antipatterns

Four of the defined micro patterns are devoted to the identification of classes whose implementation is far from the object-oriented paradigm. We can define them as a sort of object-oriented antipatterns, and, even if it has been demonstrated that their presence inside real systems is generally scarce [GM05], their identification lets the engineers focus on these classes in order to solve the issues and problems they present. The four micro patterns representing antipatterns are the following.

Cobol like
Definition: Cobol like classes are classes with a single static method, one or more static variables, but no instance methods or fields.
Relevance for the detection: *t*he programming style represented by this micro pattern is far away from object orientation. This micro pattern can be mainly detected in those main classes developed by beginner programmers, even if also well established systems and libraries may present instances of it.

Pool
Definition: a Pool is a class which declares only static final fields, but no methods.
Relevance for the detection: Pool classes are considered antipatterns as they can be generally implemented as interfaces. In [Blo01] it is known as the "constant interface antipattern".

Pseudo class
Definition: a Pseudo class is a class with no instance fields, and no concrete methods. Within it, only static fields and abstract methods are allowed.
Relevance for the detection: *t*his kind of classes constitutes an antipattern as they can be rewritten as interfaces, therefore they are good candidates for an easy refactoring.

Record
Definition: a Record is a class in which all fields are public, and no methods are declared (other than constructors and those methods inherited from `java.lang.Object`).

Relevance for the detection: *i*nstances of the Record micro pattern look very similar to Pascal record types. Such classes run against the encapsulation principle of object orientation, according to which fields should be declared private (or protected) and accessed by appropriate getter and setter methods (as it happens, for example, with data manager classes).

The detection of these micro patterns has two main advantages. First of all, if detected, they can be refactored according to the object-oriented paradigm, in order to fully comply with the rest of the system. On the other hand, their absence can be considered as an indication of good system quality, as it demonstrates that the system has been designed and implemented correctly following the object-oriented directives.

6.4. Considerations about the detection of software antipatterns and other defects

As we have outlined, in our approach for the reconstruction of software architectures we are also interested in the identification of some antipatterns (namely structural and object-oriented antipatterns) in the analyzed system.
We think that the identification of antipatterns or other kinds of design or programming defects is useful in the context of SAR activities for different reasons. First of all, their detection allows the engineers having an immediate understanding of the critical points of the systems, leading them to a focused and precise intervention on the identified problems in order to solve them in the most effective way. Detecting and consequently solving these issues will therefore improve the quality of the analyzed system and its maintenance.
Besides the structural and object-oriented antipatterns considered in our approach, several other categories of antipatterns have been presented in the literature. The most prominent categorization of antipatterns is proposed by Brown [BMMM98], who distinguishes among development, architecture and management antipatterns. These antipatterns cover various aspects of software development, like issues related to the development and management team to problems more strictly related to the implementation and to the modules composing a software system. To our current knowledge, few tools for the detection of some of these antipatterns currently exist. One of them is Analyst4j [A4J], a commercial tool devoted to the identification of the Blob, Spaghetti code and Swiss army knife antipatterns (refer to [BMMM98] for their definition). With respect to Brown's or other kinds of antipatterns, a set of elements representing simpler design or programming defects is constituted by *code smells* [Fow99]. A code smell is any symptom in the source

code of a system that possibly indicates a deeper issue, and can be seen as ineffective solutions to reccuring implementation problems. Even if they may look similar to software antipatterns, they actually differ from them for different aspects. First of all smells generally have a limited impact in the context of single classes, while antipatterns may involve also groups of classes. They are focused on the identification of bad programming practices, while antipatterns may also consider design or management issues. Smells seem to have a direct correlation to particular values of software metrics, which make them somehow more easily detectable from subject systems with respect to software antipatterns.

Even if antipatterns and smells are actually different elements, strict correlations have been pointed out by some researchers. In [MGD+09a, MGD+09b], Moha *et al.* provide a taxonomy in which some development antipatterns are related to the code smells which are exploited for their identification. They proposed an approach (DECOR) and a related technique (DETEX) for the identification of code smells and related antipatterns, basing on the computation of ad-hoc metrics. Through their approach, they identify the Blob, Functional decomposition, Spaghetti code and Swiss army knife antipatterns [BMMM98] and they provide experimental results about their detection on 11 open source Java systems.

Different other approaches and tools for smell detection have been implemented. We now cite some examples which are not exhaustive of the current literature, but which give an overall idea of the current approaches and available tools for the detection of these elements. Marinescu [Mar04] defined detection strategies for the identification of ten common design flaws. These strategies are based on combinations of metrics whose values can be indicators of the presence of flaws in the analyzed systems. Chatzigeorgiou and Tsantalis implemented JDeodorant [JDeo], an Eclipse plugin which is able to detect (and also solve) the Feature envy [TC09a, TC09c] and the Type checking [TC09b] smells, by respectively applying Move method refactoring and polymorphism exploitation. Other tools concerned with smell identification are for example FindBugs [FindBugs], which is devoted to the detection of bugs related to the correctness and the performances of Java systems, and PMD [PMD], which allows for the identification of bugs (like empty try-catch or switch statements), dead code, complicated expressions as well as duplicate code in Java systems.

These are only a sample of the available approaches. As it can be noticed, research in this field is lively active. Many tools for smell identification have been implemented, but (as we have seen) the same cannot be stated as far as antipatterns are concerned.

We shall observe that none of these tools is currently integrated in any SAR framework. For the future, it is hopefully expected that SAR tools will provide antipatterns and/or smell identification capabilities, as they are of great support for the reengineering and maintenance processes.

6.5. Concluding remarks

In this chapter we have described how the micro-structures (and in particular the EDPs and micro patterns) are currently exploited for architecture reconstruction and software analysis capabilities. EDPs have been exploited in order to generate views on the analyzed systems, both on packages (through the Package view) and on classes (through the Class compact and Class extended view). Through the analysis of EDPs, it is possible to compute a set of common object oriented and quality metrics, which help the engineers in analyzing the complexity of the subject systems and consequently focusing on those components and modules that expose criticalities. This process is also supported by the detection of structural antipatterns, which are represented by types with a high number of dependencies and/or dependents with other entities. A set of object-oriented antipatterns is also detectable through micro patterns. In this case, the engineer can analyze those classes whose implementation is not compliant with the object-oriented principles, violating encapsulation and limiting the possibility of extension or reuse of the affected classes. Several other micro patterns can be detected through our SAR module, that depict peculiar classes which the engineers can be interested in. For some of them, correlations with particular metrics values have been underlined and inspected along the chapter.

Chapter 7

A novel interpretation of micro patterns

Abstract

In this chapter we reconsider micro patterns and we suggest a novel approach to their detection aimed to identify classes that are very close and similar to a correct micro pattern implementation, even if some of the methods and/or attributes of the class do not comply with the constraints defined by the micro pattern. The new interpretation is based on two common object oriented metrics, namely the number of attributes (NOA) and the number of methods (NOM) of a class. Among the various advantages of this approach, the identification of classes or interfaces similar to micro patterns allows for example the analysis of software systems along various releases (checking if and how the nature of the attributes and/or methods of a class has changed), as well as the identification of possible critical classes that can't be detected with a precise matching approach.

7.1. Motivation

We now recall some concepts related to micro patterns that have already been presented, but that are useful in order to understand the motivations behind a new interpretation of these micro-structures. We let the reader refer to [GM05] to have a complete description of micro patterns.
Micro patterns have been defined as a set of class-level traceable patterns. Three concepts are related to this definition:

- being *class-level*: each micro pattern stands at the class abstraction level, i.e. it captures characteristics about single classes, which can be derived exclusively from the analysis of their methods and attributes;

- being *traceable*: a code structure is said to be *traceable* if it is mechanically recognizable from the analyzed systems, or, more formally, "it can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components" [GM05];
- being *patterns*: their aim is to capture some programming techniques that are very common in particular among Java developers.

Although the definition of the 27 micro patterns (as proposed by Gil and Maman) followed a rigorous and well established process, in our opinion these elements may encounter some drawbacks.
First of all, the definition of some micro patterns is ambiguous. For example, consider the Data manager micro pattern, which belongs to the Data managers micro pattern category. This elements has been defined as:

- *Data manager*: a class where all methods are either getters or setters;

A question arises here: what kind of methods should be considered as getters or setters? Are setter methods only those methods which contain only one statement that assigns an input parameter to a variable of the class? Or may they contain other statements and operations? In the same way, are getter methods only those methods which contain only one statement returning a field of the class, or may they contain other statements? This kind of ambiguity is not resolved in the original catalogue. We tried to solve this problem in Chapter 3, where getter and setter methods are considered as code atoms, and they are constituted only by one statement, which sets or correspondently returns a field of a certain class.
A second issue related to micro patterns is that all of them are placed at the class abstraction level. Each micro pattern definition begins with "A class/interface that…": this characteristic make it compulsory to analyze each class in general, considering the whole set of attributes and methods, in order to understand if the constraints expressed on each micro pattern are satisfied. As we have seen, other kinds of micro-structures are placed at lower abstraction levels, like single attributes or methods. Hypothetically considering micro patterns defined not on classes, but on the attributes and/or methods of a class, we may have more precise and focused definitions. For example, the Data manager micro pattern could be placed at the method detail level, hence it would not be defined as "a class where all methods are either getters or setters", but could be seen as "a method which is either a getter or a setter".
A third problem of micro patterns is strictly related to the previous one: micro patterns are in our opinion too much restrictive. In general, given a type (i.e. either a class or interface)

T, *T* is an instance of the micro pattern *MP* if and only if the whole set of methods and/or attributes of *T* satisfies the constraints specified for *MP*. This means that, if a type that is a correct realization of a micro pattern *MP* is even slightly modified introducing some code elements that don't comply with the specifications for *MP*, the type won't be considered an instance of *MP* anymore. For example, consider again the Data manager micro pattern. Its definition states that *all methods* are either getters or setters. Given a class implementing the Data manager micro pattern, and adding to it a method that is neither a setter nor a getter, the class is not to be considered a correct instance of the micro pattern. In the same way, there may exist in a system many classes whose largest part of methods is formed by setters and/or getters. These observations can obviously be extended to all the other micro patterns. Hence, we assert that inside a software system there are potentially numerous types presenting micro pattern *flavours*, i.e. types that look very similar to some micro patterns, except for a restricted set of attributes and/or methods that place them at a very little distance from the correct micro pattern implementation [AM09b].

In this context, our aim is finding types that present micro pattern flavours, with the help of the number of attributes (NOA) and number of methods (NOM) metrics [LK94]. These metrics generally take into account both static and non-static members of a given type. It is worth notice that these two metrics considered stand alone don't give really important information about the complexity or quality of a system. Indeed, they are focused on single classes, whose complexity or quality cannot be objectively evaluated only through the use of these (or other) simple metrics. Generally, such basic metrics have to be combined and contextualized, in order to achieve a better general understanding of single classes or of the overall analyzed system as well [KB04].

The detection of micro pattern flavours has two main advantages. First of all, the identification of types presenting micro pattern flavours will obviously lead to the detection of much more instances with respect to the exact ones. It will be noticed, thank to the definition of ad-hoc similarity measures, that many of the newly detected instances are similar to the desired micro patterns with rates often larger than 80%. The detection of these instances helps to identify a larger number of system parts that may need to be changed in order to solve design or programming issues (as for example in the case of the antipatterns already considered in Chapter 6, Section 6.3.2), or simply improved (for example by making them exactly compliant to a micro pattern definition, in the case the micro patterns they implement represent good programming practices). Moreover, the identification of types which present micro pattern flavours allows the analysis of software systems along various releases; the engineers can check if and how the nature of the attributes and/or methods of a class has changed between two different releases of the analyzed systems, making considerations about the eventual improvement of the system stability and quality.

7.2. A new interpretation of micro patterns based on NOA and NOM metrics

In order to allow the detection of types presenting micro pattern flavours through a similarity-based approach, we first propose a different categorization of micro patterns in three groups. If we consider the definitions provided for the micro patterns, we notice that each of them is based on one of the following three aspects:

- The analysis of the attributes belonging to a type;
- The analysis of the methods declared within a type;
- The analysis of both attributes and methods that characterize a type;

We define the set *A* as the set of micro patterns that can be identified by only analyzing the attributes of a type. Starting from [GM05] and from the definitions provided in Chapter 3, this set is clearly defined as *A* = {Stateless, Common state, Immutable, Box, Compound box, Canopy, Trait}.
We define the set *M* as the set of micro patterns that can be identified by analyzing only the methods declared within a type. Therefore, *M* = {Data manager, Sink, Outline, State machine, Implementor, Overrider, Extender}.
Finally, the set *AM* is the set containing those micro patterns that are identified by analyzing both the attributes and methods belonging to a certain type. Hence, *AM* = {Designator, Taxonomy, Joiner, Pool, Function pointer, Function object, Cobol like, Restricted creation, Sampler, Record, Pure type, Augmented type, Pseudo class}.
These categories contain all the micro patterns, as $|M| + |A| + |AM| = 7 + 7 + 13 = 27$.
Table 7.1 reports the number of elements of each of the eight categories defined by Gil and Maman which belong to each of the three new categories we have just defined.
The distribution of the micro patterns inside the new three categories is not surprising, and is a direct consequence of the definition of each single micro pattern.
The Degenerate state and behavior, Degenerate behavior and Controlled creation categories are all completely mapped in the *AM* set. Degenerate state and Wrappers are mapped in the *A* category, as they deal completely with a class state, which is represented by its attributes. Only the Inheritors set of micro patterns is completely mapped in the *M* category, the other categories whose elements belong to *M* have also elements that belong to at least another category out of *A* and *AM.*

Micro pattern category	*A*	*M*	*AM*
Degenerate state and behavior	0	0	4
Degenerate behavior	0	0	3
Degenerate state	3	0	0
Controlled creation	0	0	2
Wrappers	3	0	0
Data managers	0	2	1
Base classes	1	2	3
Inheritors	0	3	0
Total	7	7	13

Table 7.1 *– Relationships between the orginal micro pattern categories defined by Gil and Maman, and those based on attributes and methods*

As we have outlined, we want to revisit the micro patterns according to the NOA and NOM metrics, in order to support the detection of types presenting micro pattern flavours. To do so, we introduce three values: the *attributes similarity ratio* (*ASR*), the *methods similarity ratio* (*MSR*), and the *global similarity ratio* (*GSR*).
For each micro pattern belonging to the *A* category, ASR measures the amount of attributes of a given type which satisfy the attributes conditions specified for that micro pattern, with respect to the total number of attributes declared within the type.
For each micro pattern in *M*, MSR calculates the amount of methods of a given type which satisfy the methods conditions specified for that micro pattern, with respect to the total number of methods declared within the type.
Finally, for each micro pattern in *AM*, GSR considers both attributes and methods as being homogeneous entities of a type. Therefore, GSR measures the amount of attributes and methods (considered altogether) of a given type which satisfy the attributes and methods conditions specified for the micro pattern, with respect to the total number of attributes and methods declared within the type.
ASR, MSR and GSR are calculated considering the NOA and NOM of each given type, hence taking into account the whole set of attributes and methods that characterize each of them. These similarity ratios are percentage rates, and are calculated in a different way depending on the micro pattern of interest. Moreover, they are to be intended as an indication of how much a given type is similar to a certain micro pattern. The higher the value of these measures, the more the type is close to a correct and complete micro pattern realization. If a type has a similarity ratio of 100% to a certain micro pattern, the whole set of its attributes and/or methods satisfy the constraints specified by the micro pattern, and hence it is a precise instance of it. Instances with a 100% similarity ratio are therefore those

that can also be identified by the precise matching approach proposed by Gil and Maman [GM05]. For some micro patterns (Designator, Taxonomy, Joiner, Trait, Pure type, Pool and Record) we specify an upper bound of 3 or 5 methods and/or attributes defined by a type. This because we think it is too restrictive to consider only those types that do not define any attributes and/or methods at all. Moreover, we verified that specifying a higher upper bound would result in detecting a larger number of instances presenting flavours of these patterns, but that are of scarce interest if we consider the purpose and the specifications of these micro patterns.

Micro pattern	**Conditions on attributes**	**ASR**
Stateless	NOA == 0	1
	NOA > 0	(static fileds + final fields) / NOA
Common state	NOA == 0	0
	NOA > 0	static fields / NOA
Immutable	NOA > 1	Number of fields modified by constructor / NOA
	Else	0
Box	NOA = 0	0
	NOA = 1 Non-final fields == 1	1
	NOA > 1	0
Compound box	NOA >= 1 Non-primitive fields == 1	1
	NOA > 1 Non-primitive fields > 1	1 – (Non-primitive fileds / NOA)
Canopy	NOA == 1	Number of fields modified by constructor / NOA
	Else	0
Trait	NOA > 5	0
	Else	1 – NOA / 5

Table 7.2 *– Attribute similarity ratios for the micro patterns based on attributes analysis*

Table 7.2 reports the 7 micro patterns based on attributes, and indicates how the correspondent ASR must be calculated in order to identify types presenting flavours of these micro patterns. Given the definitions of these micro patterns in [GM05], the new interpretation and the meaning of the similarity ratios should be straightforward.

Table 7.3 reports the 7 micro patterns based on methods, and indicates how the MSR is calculated in order to identify types presenting flavours of these micro patterns.

Micro pattern	Conditions on methods	MSR
Data manager	NOM = 0	0
	NOM > 0	(Getter methods + setter methods) / NOM
Sink	NOM = 0	0
	NOM > 0	Propagating methods[1] / NOM
Outline	Methods invoking an abstract method of the same class >= 1	1
	Else	0
State machine	NOM = 0	0
	NOM > 0	1 – parameterized methods [2]/ NOM
Implementor	NOM = 0	0
	NOM > 0	Implementing methods[3] / NOM
Overrider	NOM = 0	0
	NOM > 0	Overriding methods[4] / NOM
Extender	NOM = 0	1
	NOM > 0	1 – overriding methods / NOM

Table 7.3 *– Method similarity ratios for the micro patterns based on methods analysis*

[1] A *propagating method* is a method which invokes at least another method within its body, either defined in the same class or in another type. In Table 7.3, *Propagating methods* represents therefore the total number of propagating methods detected in a class.

[2] A *parameterized method* is a method which defines at least one formal parameter. In Table 7.3, *Parameterized methods* is therefore the number of methods with parameters identified in a type.

[3] An *implementing method* is a method which overrides an inherited abstract method. In Table 7.3, *Implementing methods* represents the number of implementing methods of a class.

[4] An *overriding method* is a method which overrides an inherited non-abstract method. In Table 7.3, *Overriding methods* represents the number of overriding methods detected in a class.

Table 7.4 reports the 13 micro patterns based on both attributes and methods, and reports how the GSR similarity value is to be calculated for each of them.

Micro pattern	Conditions on attributes	Conditions on methods	GSR
Designator	NOA > 3	In any case	0
	In any case	NOM > 3	0
	Else	Else	((1 - NOA / 3) + (1 – NOM / 3)) / 2
Taxonomy	NOA > 3	In any case	0
	In any case	NOM > 3	0
	Else	Else	((1 - NOA / 3) + (1 – NOM / 3)) / 2
Joiner	NOA > 3	In any case	0
	In any case	NOM > 3	0
	Else	Else	((1 - NOA / 3) + (1 – NOM / 3)) / 2
Pool	In any case	NOM > 5	0
		Else	((1 – NOM / 5) + (static final fields / NOA)) / 2
Function pointer	NOA == 0	NOM >= 1 Public methods == 1	1
	Else	Else	0
Function object	NOA >= 1	NOM >= 1 Public methods == 1	1
	Else	Else	0
Cobol like	In any case	Static methods == 1 NOM == 1	(static methods + static fields) / (NOM + NOA)
		Static methods == 1	0
		NOM > 1 Else	(static methods + static fields) / (NOM + NOA)
Restricted creation	Static same class fields >= 1	In any case	Private constructors / constructors
	Else		0
Sampler	Static same class fields >= 1	Public constructors >= 1	1
	Else	Else	0

Record	In any case	NOM > 5	0
		Else	((1 – NOM / 5) + (public fields /NOA)) / 2
Pure type	NOA > 5	In any case	0
	NOA <= 5 Non static fields == NOA		((1 – Non static fields /5) + (abstract methods / NOM)) / 2
	Else		0
Augmented type	Static final same class fields >= 3	In any case	(1 + abstract methods / NOM) / 2
	Else		((Static final same class fields / 3) + (abstract methods / NOM)) / 2
Pseudo class	In any case	In any case	(abstract methods + static methods + static fields) / (NOM + NOA)

Table 7.4 *– Global similarity ratios for the micro patterns based on both attributes and methods analysis*

7.3. Experimental results

To prove the importance of detecting not only classes that exactly match the micro patterns definitions, but also those containing relevant micro pattern flavours, we identified instances of micro patterns and micro pattern flavours on the Java systems reported in Table 7.5.

System and version	**Description**	**Number of packages**	**Number of types**
Ant 1.5.2	Java-based build tool	56	724
Ant 1.6.2		67	951
Ant 1.7.1		72	1130
JHotDraw 5.1	GUI framework	11	172
JHotDraw 6.0b1		30	544
JHotDraw 7.1		44	718
Apache Lucene 1.4.3	Text search engine library	24	294
Apache Lucene 1.9		25	459
Apache Lucene 2.0		24	399
Total		437	5391

Table 7.5 *– An overview of the analyzed systems*

For each system, we considered three different releases in order to let a comparison about the evolution of micro patterns throughout different releases be possible.
The following tables report the results of the micro patterns detection on the analyzed systems. Besides reporting the name of each micro pattern, the tables are divided into two main sections. The first half indicates the precise matching results: i.e. for each micro pattern and for each system release, it reports the number of exact detected instances, as well as the percentage of types implementing the micro pattern with respect to the total number of types constituting the subject system release. The second part reports the number of types (and their percentage with respect to the overall system) whose similarity ratio (i.e. ASR, MSR or GSR, depending on the considered kind of micro pattern) with the correspondent micro pattern is at least 80%. We consider 80% as an acceptable lower-bound threshold, as it allows the detection of those instances that are mostly closed to the correct micro patterns implementation.
Table 7.6 to 7.8 report the results obtained on the analysis of Ant, respectively about the micro patterns based on attributes, those based on methods, and those based on both attributes and methods.
As far as the micro patterns based on attributes are concerned, their distribution in the three releases remains quite constant, both for the precise matching instances and for the classes presenting micro pattern flavours. A considerable number of classes (around 30%) are Stateless classes, i.e. their attributes are uniquely both static and final.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*Ant 1.5.2*		*Ant 1.6.2*		*Ant 1.7.1*		*Ant 1.5.2*		*Ant 1.6.2*		*Ant 1.7.1*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Box	130	18,0%	157	16,5%	177	15,7%	5	0,7%	4	0,4%	7	0,7%
Canopy	13	1,8%	12	1,3%	16	1,4%	0	0,0%	0	0,0%	0	0,0%
Common state	39	5,4%	57	6,0%	104	9,2%	9	1,2%	11	1,2%	16	1,7%
Compound box	140	19,3%	162	17,0%	195	17,3%	7	1,0%	7	0,7%	11	1,2%
Immutable	20	2,8%	18	1,9%	16	1,4%	1	0,1%	2	0,2%	1	0,1%
Stateless	206	28,5%	285	30,0%	356	31,5%	8	1,1%	9	0,9%	16	1,7%
Trait	2	0,3%	5	0,5%	6	0,5%	9	1,2%	13	1,4%	18	1,9%

Table 7.6 *– Micro patterns based on attributes: detection results on three releases of Ant*

Also the micro patterns based on methods (Table 7.7) don't undergo any particular evolution. In this case, it is relevant the presence of many classes presenting flavours of the Extender micro pattern, while fewer are precise implementations of it. This means that there is a considerable number of classes (around 20%) within Ant which override only a small part of the inherited methods, while an exact implementation of the Extender micro pattern would not override any inherited method.

	Precise matching						Similarity matching (at least 80%)					
Micro pattern	*Ant 1.5.2*		*Ant 1.6.2*		*Ant 1.7.1*		*Ant 1.5.2*		*Ant 1.6.2*		*Ant 1.7.1*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Data manager	34	4,7%	45	4,7%	49	4,0%	16	2,2%	21	2,2%	22	1,9%
Extender	60	8,3%	85	8,9%	108	7,5%	143	19,8%	188	19,8%	211	18,7%
Implementor	46	6,4%	59	6,2%	50	5,2%	0	0,0%	0	0,0%	1	0,1%
Outline	10	1,4%	14	1,5%	20	1,2%	0	0,0%	0	0,0%	0	0,0%
Overrider	17	2,3%	28	2,9%	41	2,5%	1	0,1%	3	0,3%	5	0,4%
Sink	39	5,4%	53	5,6%	55	4,7%	9	1,2%	10	1,1%	9	0,8%
State machine	3	0,4%	3	0,3%	7	0,3%	1	0,1%	1	0,1%	1	0,1%

Table 7.7 *– Micro patterns based on methods: detection results on three releases of Ant*

As far as the third category of micro patterns is concerned (those based on both attributes and methods, reported in Table 7.8), we can notice how the number of Function objects increases as well as the Function pointer instances decrease.
No other particular evolutions are to be noticed. There are very few instances of the four micro patterns devising antipatterns (i.e. Cobol like, Pool, Pseudo class and Record), hence the system seems to be well implemented according to the object-oriented paradigm. Anyway, there are also some classes presenting considerable flavours of the Pool and Record micro patterns, that will probably need to be further inspected.

	Precise matching						Similarity matching (at least 80%)					
Micro pattern	*Ant 1.5.2*		*Ant 1.6.2*		*Ant 1.7.1*		*Ant 1.5.2*		*Ant 1.6.2*		*Ant 1.7.1*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Augmented type	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%
Cobol like	10	1,4%	13	1,4%	17	1,5%	2	0,3%	3	0,3%	9	0,8%
Designator	1	0,1%	4	0,4%	4	0,4%	17	2,3%	23	2,4%	31	2,7%
Function object	40	5,5%	54	5,7%	86	7,6%	0	0,0%	0	0,0%	0	0,0%
Function pointer	76	10,5%	93	9,8%	74	6,5%	0	0,0%	0	0,0%	0	0,0%
Joiner	1	0,1%	3	0,3%	3	0,3%	0	0,0%	0	0,0%	0	0,0%
Pool	1	0,1%	2	0,2%	5	0,4%	15	2,1%	25	2,6%	50	4,4%
Pseudo class	0	0,0%	1	0,1%	1	0,1%	0	0,0%	0	0,0%	0	0,0%
Pure type	34	4,7%	46	4,8%	59	5,2%	2	0,3%	2	0,2%	3	0,3%
Record	8	1,1%	10	1,1%	10	0,9%	9	1,2%	16	1,7%	25	2,2%
Restricted creation	1	0,1%	1	0,1%	2	0,2%	0	0,0%	0	0,0%	0	0,0%
Sampler	0	0,0%	0	0,0%	5	0,4%	0	0,0%	0	0,0%	0	0,0%
Taxonomy	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%

Table 7.8 *– Micro patterns based on both attributes and methods: detection results on three releases of Ant*

Table 7.9 to 7.11 report the results obtained on the analysis of three different releases of JHotDraw.

A considerable amount of Box, Compound box and Stateless instances are found in all the releases. It should be noticed that there are very few classes which are similar to micro patterns, most of the found instances precisely match with the specifications. On one hand, this can be an indication of a well-structured and implemented system, where each single class seems to be designed to fully comply with the micro pattern specifications. On the other hand, we can assert that the definition of micro patterns is actually able to well capture programming practices, codifying classes whose structure is commonly present in well developed systems.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*JHotDraw 5.1*		*JHotDraw 6.0b1*		*JHotDraw 7.1*		*JHotDraw 5.1*		*JHotDraw 6.0b1*		*JHotDraw 7.1*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Box	35	20,3%	70	12,9%	91	12,7%	0	0,0%	1	0,2%	0	0,0%
Canopy	13	7,6%	22	4,0%	6	0,8%	0	0,0%	0	0,0%	0	0,0%
Common state	11	6,4%	22	4,0%	92	12,8%	0	0,0%	2	0,4%	1	0,1%
Compound box	51	29,7%	258	47,4%	100	13,9%	0	0,0%	0	0,0%	0	0,0%
Immutable	15	8,7%	24	4,4%	20	2,8%	0	0,0%	1	0,2%	1	0,1%
Stateless	57	33,1%	160	29,4%	235	32,7%	0	0,0%	2	0,4%	1	0,1%
Trait	2	1,2%	6	1,1%	8	1,1%	3	1,7%	3	0,6%	8	1,1%

Table 7.9 *– Micro patterns based on attributes: detection results on three releases of JHotDraw*

Differently from what happened with Ant, the three releases actually underwent important evolutions and modifications, as proved by the changes in the micro pattern percentages along the releases. This is also demonstrated by the heavily different number of types that constitute each release, and by the different structure of packages and kind of types each of them is composed of.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*JHotDraw 5.1*		*JHotDraw 6.0b1*		*JHotDraw 7.1*		*JHotDraw 5.1*		*JHotDraw 6.0b1*		*JHotDraw 7.1*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Data manager	3	1,7%	8	1,5%	3	0,4%	0	0,0%	0	0,0%	3	0,4%
Extender	7	4,1%	22	4,0%	49	6,8%	1	0,6%	20	3,7%	12	1,7%
Implementor	0	0,0%	0	0,0%	0	0,0%	0	0,0%	3	0,6%	2	0,3%
Outline	5	2,9%	7	1,3%	15	2,1%	0	0,0%	0	0,0%	0	0,0%
Overrider	10	5,8%	32	5,9%	39	5,4%	5	2,9%	6	1,1%	12	1,7%
Sink	25	14,5%	34	6,3%	61	8,5%	1	0,6%	3	0,6%	17	2,4%
State machine	2	1,2%	7	1,3%	4	0,6%	0	0,0%	1	0,2%	1	0,1%

Table 7.10 *– Micro patterns based on methods: detection results on three releases of JHotDraw*

Similar considerations can be made while analyzing the other two categories of micro patterns. Once again, we can notice how very few instances present micro pattern flavours, being a hint for a well-designed system. This is also evident by the scarce presence of the Cobol like, Pool, Pseudo class and Record micro patterns: their absence justify for a system which is strictly developed according to the object-oriented principles.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*JHotDraw 5.1*		*JHotDraw 6.0b1*		*JHotDraw 7.1*		*JHotDraw 5.1*		*JHotDraw 6.0b1*		*JHotDraw 7.1*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Augmented type	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%
Cobol like	2	1,2%	4	0,7%	12	1,7%	0	0,0%	1	0,2%	1	0,1%
Designator	0	0,0%	3	0,6%	1	0,1%	5	2,9%	8	1,5%	6	0,8%
Function object	10	5,8%	16	2,9%	64	8,9%	0	0,0%	0	0,0%	0	0,0%
Function pointer	15	8,7%	29	5,3%	21	2,9%	0	0,0%	0	0,0%	0	0,0%
Joiner	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%
Pool	0	0,0%	0	0,0%	17	2,4%	6	3,5%	9	1,7%	20	2,8%
Pseudo class	0	0,0%	2	0,4%	1	0,1%	0	0,0%	0	0,0%	0	0,0%
Pure type	18	10,5%	41	7,5%	45	6,3%	0	0,0%	0	0,0%	0	0,0%
Record	1	0,6%	1	0,2%	15	2,1%	0	0,0%	1	0,2%	22	3,1%
Restricted creation	1	0,6%	2	0,4%	2	0,3%	0	0,0%	0	0,0%	0	0,0%
Sampler	1	0,6%	5	0,9%	4	0,6%	0	0,0%	0	0,0%	0	0,0%
Taxonomy	0	0,0%	1	0,2%	0	0,0%	0	0,0%	0	0,0%	0	0,0%

***Table 7.11** – Micro patterns based on both attributes and methods: detection results on three releases of JHotDraw*

Finally, Tables 7.12 to 7.14 indicate the results obtained on the analysis of Lucene. As it happened with Ant and JHotDraw, the majority of the micro patterns based on attributes are instances of the Box, Compound box or Stateless micro pattern.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*Lucene 1.4.3*		*Lucene 1.9*		*Lucene 2.0*		*Lucene 1.4.3*		*Lucene 1.9*		*Lucene 2.0*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Box	88	29,9%	142	30,9%	142	35,6%	2	0,7%	0	0,0%	0	0,0%
Canopy	3	1,0%	2	0,4%	2	0,5%	0	0,0%	0	0,0%	0	0,0%
Common state	18	6,1%	20	4,4%	20	5,0%	6	2,0%	6	1,3%	6	1,5%
Compound box	60	20,4%	83	18,1%	82	20,6%	3	1,0%	3	0,7%	3	0,8%
Immutable	8	2,7%	6	1,3%	6	1,5%	1	0,3%	1	0,2%	1	0,3%
Stateless	61	20,7%	77	16,8%	77	19,3%	5	1,7%	6	1,3%	6	1,5%
Trait	11	3,7%	13	2,8%	12	3,0%	8	2,7%	10	2,2%	9	2,3%

***Table 7.12** – Micro patterns based on attributes: detection results on three releases of Lucene*

No particular evolution in terms of micro patterns has been registered along the three releases. For some micro patterns (like Trait, Implementor and State machine), the number of classes presenting flavours is considerable if compared with the exact instances. This suggests us to check for those classes and see if it is possible to make them fully compliant with the micro pattern specifications.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*Lucene 1.4.3*		*Lucene 1.9*		*Lucene 2.0*		*Lucene 1.4.3*		*Lucene 1.9*		*Lucene 2.0*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Data manager	9	3,1%	10	2,2%	9	2,3%	2	0,7%	1	0,2%	1	0,3%
Extender	14	4,8%	17	3,7%	16	4,0%	0	0,0%	3	0,7%	2	0,5%
Implementor	1	0,3%	1	0,2%	4	1,0%	4	1,4%	10	2,2%	9	2,3%
Outline	13	4,4%	16	3,5%	16	4,0%	0	0,0%	0	0,0%	0	0,0%
Overrider	4	1,4%	6	1,3%	3	0,8%	2	0,7%	2	0,4%	2	0,5%
Sink	4	1,4%	8	1,7%	8	2,0%	0	0,0%	0	0,0%	0	0,0%
State machine	1	0,3%	1	0,2%	1	0,3%	3	1,0%	3	0,7%	3	0,8%

***Table 7.13** – Micro patterns based on methods: detection results on three releases of Lucene*

This is even more relevant with three of the micro patterns codifying antipatterns, namely Pool, Pseudo class, and Record.

	Precise matching						**Similarity matching (at least 80%)**					
Micro pattern	*Lucene 1.4.3*		*Lucene 1.9*		*Lucene 2.0*		*Lucene 1.4.3*		*Lucene 1.9*		*Lucene 2.0*	
	No.	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*	*No.*	*%*
Augmented type	0	0,0%	0	0,0%	0	0,0%	1	0,3%	1	0,2%	1	0,3%
Cobol like	7	2,4%	9	2,0%	9	2,3%	1	0,3%	1	0,2%	1	0,3%
Designator	1	0,3%	2	0,4%	2	0,5%	2	0,7%	3	0,7%	3	0,8%
Function object	48	16,3%	60	13,1%	60	15,0%	0	0,0%	0	0,0%	0	0,0%
Function pointer	12	4,1%	15	3,3%	15	3,8%	0	0,0%	0	0,0%	0	0,0%
Joiner	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%	0	0,0%
Pool	3	1,0%	3	0,7%	3	0,8%	3	1,0%	13	2,8%	12	3,0%
Pseudo class	4	1,4%	3	0,7%	4	1,0%	3	1,0%	3	0,7%	3	0,8%
Pure type	12	4,1%	13	2,8%	14	3,5%	2	0,7%	3	0,7%	2	0,5%
Record	16	5,4%	15	3,3%	15	3,8%	3	1,0%	8	1,7%	7	1,8%
Restricted creation	0	0,0%	7	1,5%	7	1,8%	0	0,0%	0	0,0%	0	0,0%
Sampler	1	0,3%	1	0,2%	1	0,3%	0	0,0%	0	0,0%	0	0,0%
Taxonomy	1	0,3%	1	0,2%	1	0,3%	0	0,0%	0	0,0%	0	0,0%

***Table 7.14** – Micro patterns based on both attributes and methods: detection results on three releases of Lucene*

As these elements represent critical classes inside a system, those classes presenting flavours of these patterns are possible candidates for refactoring. Anyway, the number of instances of these micro patterns is low if compared with the size of the system. Like for

JHotDraw, this is an indicator of a system that is well developed according to the principles of object-orientation.

7.4. Concluding remarks

From the above experimentations, some interesting conclusions can be drawn. First of all, Box, Compound box and Stateless are the more frequent micro patterns in all the systems and in all the considered releases. These results can be also compared with [GM05], Table 4, which demonstrates how these micro patterns are quite frequent, independently from the analyzed system. They seem therefore to codify types that are very recurrent in the programming practices.

Another observation comes from the occurrences of the Trait micro pattern. Trait identifies abstract classes which have no state. To some extent, this micro pattern can therefore be considered as an indication of the abstractness of the considered system. The relaxation of the constraint of this micro pattern to allow the existence of some state succeeded in identifying a good number of classes presenting Trait flavours. In some cases, classes closely similar to Trait are more than those that exactly match the micro pattern. These classes could be further inspected in order to make them fully compliant with the specifications.

Instances of the Extender, Implementor and Overrider micro patterns give indications about how the systems organize class hierarchies. JHotDraw mainly overrides non-abstract methods, hence redefining already established operations. Many classes are similar to the Extender micro pattern, hence they extend a class overriding only a little set of methods. Many precise instances of the Extender micro pattern have also been detected on Lucene and Ant (the latter providing also an interesting rate of Extender flavours). In these systems, a good percentage of classes overriding abstract methods of the superclasses can also be found. The detection of classes presenting micro pattern flavours allowed for the identification of a good number of inheritors micro patterns, that surely add more information to that provided uniquely by the precise instances, and gives an overview of the inheritance mechanisms adopted by the various systems.

Moreover, the identification of micro pattern flavours is especially useful in two cases. First of all, identifying flavours of micro patterns devising good programming practices in some classes can suggest to modify those classes in order to make them fully compliant with the specifications. On the other side, it is useful to identify flavours of micro patterns codifying bad programming practices, like Cobol like, Pool, Pseudo class and Record, that would not be detected by any precise matching approach. The detection of these instances suggests to refactor them in order to solve the issues they present.

Chapter 8

Conclusions and future works

8.1. Conclusions

The main contributions and results of this thesis are placed in the vast field of reverse engineering. In particular, the focus was concentrated on exploring and analyzing three categories of software micro-structures (elemental design patterns, design pattern clues and micro patterns), inspecting their relevance and role in design pattern detection and software architecture reconstruction activities.
From a first in-depth analysis of micro-structures, we provided a comparison aiming at underlining the characteristics and peculiarities of each category, and at tracing the possible similarities and differences among them. An important outcome of this comparative evaluation is related to the disadvantages exposed by the considered micro-structures. Two main drawbacks can be pointed out. First of all, the definition of micro-structures is generally not formal, and may result ambiguous in some cases. Secondly, each category has always been considered as a stand-alone, and never compared with other micro-structures categories. Indeed, we think that micro-structures, independently from their purposes and definitions, can actually be considered as similar elements, in that they can all be automatically identified from source code and exploited for DPD and/or SAR purposes, as discussed in this thesis. For these reasons, we provided a novel definition of the considered micro-structures that is based on common core concepts. The redefinition process aimed at solving the possible ambiguities presented by the micro-structures, and at giving them a common categorization and meaning. We implemented a module for the identification of all the considered micro-structures inside Java systems, which has been developed as an Eclipse plugin.
The first introduction of micro-structures was supported by a practical task. Elemental design patterns were exploited for design pattern detection purposes, supported by the SPQR approach. Design pattern clues as well have been introduced and exploited for

design pattern detection, while micro patterns have been defined to identify common programming techniques, and their relevance and presence in existing systems has been analyzed. Anyhow, each micro-structures category has never been considered with respect to the already defined ones, in order to verify if the joint exploitation of different kinds of micro-structures would have provided advantages in the activities of interest. We faced this issue by considering the whole set of EDPs, clues and micro patterns and inspect their global usefulness for both DPD and SAR purposes.

As far as DPD is concerned, we inspected the role and relevance of micro-structures in the identification of pattern roles and pattern structures. From our experimentations on different sets of pattern instances, we verified how EDPs reveal useful for the extraction of structural information related to the pattern instances, while clues are more suited to identify pattern roles. On the other hand, in general micro patterns didn't provide useful information, neither for the recovery of structural information about a pattern, nor for the identification of their roles. Some exceptions shall be made, like for the Template method pattern, whose abstract class role can be correctly identified by the Outline micro pattern. Starting from our experimentations, for each pattern we defined the set of micro-structures that are useful in the detection process. Through our work we pointed out how considering only one single category of micro-structures would not be sufficient in order to detect all the peculiarities related to a design patterns. On the contrary, this seems possible while extending the considerations to at least two micro-structures categories.

An innovative research task we have introduced is related to the refinement of the design pattern detection results provided by various detection tools. It is well known and experimented that design pattern detection tools generally identify different design patterns and detect different instances, even while analyzing the same systems. This is mainly due to the heterogeneous strategies adopted in the detection process, and also to the lack of formalization for design patterns, which is one of the causes of the well known variants problem. As the detection results vary among tools, this implies that the tools necessarily identify many false positive instances, which badly affect the precision of the results. The refinement process we proposed is aimed to improve the precision of the detection tools by trying to eliminate false positive instances. The approach is based on the application of micro-structure-based rules on the detected instances, which allow to discard those instances that do not implement the necessary micro-structures which characterize the corresponding pattern. From our experimentations, it emerged that, out of the considered patterns, the refinement rules behave well for the Factory method, Singleton, Template method, Visitor, Composite and Decorator design patterns, while some problems were encountered in the refinement of the Adapter pattern, due to its generality.

Focusing on micro patterns, we gave a novel interpretation of them, basing on the NOM and NOA metrics, which allowed us to identify classes that are very close and similar to a micro pattern implementation, but couldn't be detected by the original precise matching approach. The adopted similarity-based approach allowed us to compare different releases of a same subject system, inspecting and analyzing the evolution of the implemented micro patterns throughout the various releases. In the case the considered micro pattern represents a well established and valid programming practice, identifying classes that are very similar to this micro pattern lets the engineers concentrate on it in order to make it fully compliant with the micro pattern specifications. On the other hand, in the case of micro patterns representing design or programming issues, it allows to solve the issues they present, and that would have been not considered by the precise matching detection approach.

As far as software architecture reconstruction is concerned, we shall remind that, to our current knowledge, the exploitation of micro-structures for this activity has never been inspected. We moved the first steps towards this research direction, developing a module for software architecture reconstruction based on micro-structures detection which is devoted to the generation of package and class views on the subject systems, as well as to the computation of object-oriented and quality metrics, and to the detection of structural and object-oriented antipatterns, or of classes of particular interest. In this context, we experimented that EDPs can be exploited in order to recover the relationships among the packages and classes composing a system. Consequently, they are also suitable to compute the dependencies and dependents of each single software entity. Dependencies and dependents are the core values which other quality metrics can be calculated from, as we discussed in chapter 6. We also provided considerations about classes with a high number of dependencies and/or dependents, which are to be considered as structural antipatterns and hence need particular consideration by the software engineers. In fact, these components are the most critical, as they negatively influence the maintainability and evolution of the system itself. In the context of SAR, we also considered micro patterns as means to identify classes of particular interest and some object-oriented antipatterns. For some micro patterns, we noticed possible correlations with the dependencies and dependents metrics, and discussed about them and their relevance. Micro patterns were also exploited to detect four antipatterns representing classes whose implementation is far from the object-oriented best practices. For SAR purposes, we didn't consider design pattern clues, as they didn't reveal useful for the identification of structural relationships among system components. In fact, for their own definition, they are focused on the identification of hints for the presence of design patterns inside the system, which are generally intra-class peculiarities and don't devise any particular constraint about the relationships among different classes.

The exploitation of micro-structures for SAR as well as for DPD allowed us to have a common source of information for both activities, without the need to further inspect and analyze the subject systems in order to achieve the desired functionalities. At now, only few tools support both functionalities, and in general they are able to detect a little set of patterns, as far as DPD is concerned. We think that SAR and DPD are strictly related activities. In fact, DPD can be exploited to assert the quality of a system and the presence of well designed and reusable system modules, whose presence is important in order to have an easily maintainable and evolvable system. Having a common source of information for both DPD and SAR is the first step towards the complete development of an integrated approach and tool supporting both activities.

8.2. Future works

As the exploitation of micro-structures for both DPD and SAR purposes is an innovative research field, many future activities can be devised. Focusing on our work, the refinement process described in Chapter 5 seems very promising. We will extend our experimentations by considering the results provided by more design pattern detection tools on a larger set of subject systems. Moreover, we are planning to extend the set of patterns we will be able to refine, by defining appropriate validation rules for them. We have also planned to integrate the refinement approach within the benchmark platform for design pattern detection proposed in [ATZ08]. The integration will hopefully result in an extended application of the refinement process on the results provided by more detection tools on the analysis of a wide set of systems. The exploitation of the approach will reveal useful also to improve the comparisons among the instances detected by the various tools.
As far as SAR activities are concerned, we have implemented a first module for architecture reconstruction, metrics computation and antipattern detection. We are currently planning to extend our work with new views and with the computation of other metrics. Furthermore, we are working on the detection of the antipatterns defined in [BMMM98], as at our knowledge no tool for the detection of these entities currently exists (except for Analyst4J [A4J], a commercial tool which is able to detect three of the defined antipatterns, namely Blob, Spaghetti code and Swiss army knife). We think that the detection of antipatterns is crucial in order to assess the quality of a system and to possibly identify its critical components.
It will also be interesting to analyze the smells presented in [Fow99] and the approaches for their detection, in order to understand if our SAR approach could actually benefit from the identification of these elements. In this context it will be of particular interest trying to integrate or exploit the approaches presented for example in [MGD+09a, MGD+09b,

TC09a, TC09b, TC09c] with our tool, as well as start possible collaborations with researchers working in this field.
In the last years, the migration of legacy systems towards SOA architectures is achieving more and more importance [BSL05, LMSB05]. We think that the detection of design patterns and antipatterns, supported by an integrated tool, can be of great usefulness for this process. The core concept behind the migration process resides in the identification of system components that can be externally exposed as services. Some design patterns, like the Façade [GHJV94], are well suited for the identification of candidate classes to be migrated to services, as they completely hide the real implementation of a system, providing a single integrated interface to access it. Some preliminary work in this direction and discussion on these topics can be found in [ATZ08b].
On the other hand, the identification of classes implementing other kinds of design patterns or antipatterns as described in this thesis will reveal the impossibility to migrate them to services. For example, classes or modules that are highly globally coupled with the overall system are not good candidates for this migration, as they do not expose a uniform interface that can be effectively exploited and accessed from outside the system.
Another research direction we would like to investigate is related to the dynamic analysis of software systems. In this context, a survey of dynamic analysis techniques to support program comprehension can be found in [CZD+09]. At now, we only focused on static analysis because the considered micro-structures, as they are directly extracted from the source code of the analyzed systems, currently codify static information. Moving towards dynamic analysis we will be able to generate behavioural views and reports about the subject systems, as well as to provide hints for the detection of behavioural design patterns, whose identification through micro-structures (and through standard static analysis approaches) revealed troublesome.
All these considerations suggest us that having an integrated approach for DPD and SAR based on software micro-structures will allow us to support software evolution through design quality evaluation. As it is well established, design quality is strictly related to the computation of quality metrics, as well as to the presence or absence of design patterns or antipatterns. The integration of DPD and SAR functionalities in a single tool will ease and improve these kinds of evaluations.

List of publications

S. Maggioni, Design Pattern Clues for Creational Design Patterns, Proceedings of the 1st International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE 2006), Benevento, Italy, October 2006.

F. Arcelli et al., MARPLE: Metrics and Architecture Recovery Plug-in for Eclipse, Technical Report, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)-12-02-06, University of Milano-Bicocca, 2006.

S. Maggioni, Exploiting Application Portfolio Management Techniques to Understand Reverse Engineering Activities, 1st International Working Session on Reverse Engineering techniques for Application Portfolio Management (RE4APM 2007), co-located event with ICSM 2007, Paris, France, October 2007.

F. Arcelli, C. Tosi, M. Zanoni, and S. Maggioni, The MARPLE project - A Tool for Design Pattern Detection and Software Architecture Reconstruction. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008), co-located event with ECOOP 2008, Paphos, Cyprus, July 2008.

F. Arcelli, S. Maggioni, C. Tosi, M. Zanoni, Refining Design Pattern Detection through Design Pattern Clues, submitted to the 15th Working Conference on Reverse Engineering (WCRE 2008), Antwerp, Belgium, October 2008.

S. Maggioni, Redefining Micro Patterns to Support the Analysis of Software Evolution, submitted to LNCS Transactions on Pattern Languages of Programming (TPLoP), January 2009.

F. Arcelli, C. Tosi, M. Zanoni, R. Porrini, M. Vivanti, S. Maggioni, A Model Driven Approach for Program Comprehension to Support Software Evolution, Technical

Report, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)-20-03-09, University of Milano-Bicocca, 2009.

S. Maggioni, F. Arcelli, Metrics-Based Detection of Micro Patterns to Improve the Assessment of Software Quality, 1st International Symposium on Emerging Trends in Software Metrics (ETSM 2009), Pula, Italy, May 2009.

S. Maggioni, F. Arcelli, C. Tosi, M. Zanoni, Refining Design Pattern Detection through Design Pattern Clues, submitted to the Journal of Systems and Software, July 2009.

S. Maggioni, C. Tosi, M. Zanoni, A Design Pattern Detection Plugin for Eclipse, 4th Italian Workshop on Eclipse Technologies (Eclipse-IT 2009), Bergamo, Italy, September 2009.

F. Arcelli, S. Maggioni, C. Raibulet, Towards the Detection of Design Patterns through Micro-Structures: an Achievable Task?, submitted to the ACM Transactions on Software Engineering and Methodologies (TOSEM), September 2009.

F. Arcelli, S. Maggioni, C. Raibulet, Tasting Design Patterns: a Survey on Their Ingredients, submitted to the Journal of Software Maintenance and Evolution (JSME), October 2009.

S. Maggioni, F. Arcelli, An Experience Report on Using Software Analysis Tools for Java Systems, submitted to the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), October 2009.

S. Maggioni, F. Arcelli, Supporting Software Evolution through Design Quality Evaluation, submitted to IEEE EVOOL 2010, October 2009.

References

[AB05] C. Artho, A. Biere, Combined Static and Dynamic Analysis, Technical Report 466, AIOOL, 2005.

[AM09] F. Arcelli, S. Maggioni, An Experience Report on Using Software Analysis Tools for Java Systems, submitted to the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), October 2009.

[AM09b] S. Maggioni, F. Arcelli, Metrics-Based Detection of Micro Patterns to Improve the Assessment of Software Quality, 1st International Symposium on Emerging Trends in Software Metrics (ETSM 2009), Pula, Italy, May 2009.

[AMR09a] F. Arcelli, S. Maggioni, C. Raibulet, Tasting Design Patterns: a Survey on Their Ingredients, submitted to the Journal of Software Maintenance and Evolution (JSME), October 2009.

[AMR09b] F. Arcelli, S. Maggioni, C. Raibulet, Towards the Detection of Design Patterns through Micro-Structures: an Achievable Task?, submitted to the ACM Transactions on Software Engineering and Methodologies (TOSEM), September 2009.

[AMRT05] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato, A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition, in Proceedings of the Australian Software Engineering Conference, Brisbane, Australia, March 28th-31st 2005, pp. 262-269.

[APRR09] F. Arcelli, F. Perin, C. Raibulet, S. Ravani, JAdept: Behavioural Design Pattern Detection through Dynamic Analysis, in Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE'09), Milan, Italy, 2009, pp. 95-106.

[Arc06] F. Arcelli et al., MARPLE: Metrics and Architecture Recovery Plug-in for Eclipse, Technical Report, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)-12-02-06, University of Milano-Bicocca, 2006.

[ATZ08] F. Arcelli, C. Tosi, M. Zanoni, A Benchmark Proposal for Design Pattern Detection Tools, in Proceedings of the FAMOOS 2008 Workshop, co-located event with IEEE WCRE 2008, Antwerp, Belgium, Ocotber 2008.

[ATZ08b] F. Arcelli, C. Tosi, M. Zanoni, Can Design Pattern Detection be Useful for Legacy Systems Migration towards SOA?, in Proceedings of the 2nd international workshop on Systems development in SOA environments (SDSOA 2008), Leipzig, Germany, May 2008.

[ATZM08] F. Arcelli, C. Tosi, M. Zanoni, and S. Maggioni, The MARPLE project - A Tool for Design Pattern Detection and Software Architecture Reconstruction. International Workshop on Advanced Software Development Tools and Techniques (WASDeTT 2008), co-located event with ECOOP 2008, Cyprus, 2008.

[ATZ+09] F. Arcelli, C. Tosi, M. Zanoni, R. Porrini, M. Vivanti, S. Maggioni, A Model Driven Approach for Program Comprehension to Support Software Evolution, Technical Report, Dipartimento di Informatica, Sistemistica e Comunicazione (DISCo)-20-03-09, University of Milano-Bicocca, 2009.

[A4J] Codeswat Analyst4j, http://www.codeswat.com/cswat/index.php?option=com_content&task=view&id=43&Itemid=63

[Bau] Bauhaus, http://www.*Bauhaus*-stuttgart.de/demo/index.html

[BCK03] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 2nd edition, Addison-Wesley, 2003.

[BF03] Z. Balanyi, R. Ferenc, Mining Design Patterns from C++ Source Code, in Proceedings of the International Conference on Software Maintenance (ICSM'03), Amsterdam, The Netherlands, 2003, pp. 305-314.

[BG97] B. Bellay, H. Gall, A Comparison of four Reverse Engineering Tools, in Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE'97), 1997.

[Bir07] M. Birkner, Object-Oriented Design Pattern Detection using Static and Dynamic Analysis in Java Software, M.Sc. Thesis, University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin, Germany, August 2007.

[BL03] D. Beyer, C. Lewrentz, CrocoPat: Efficient Pattern Analysis in Object-Oriented Programs, in Proceedings of the 11th IEEE International Workshop on Program Comprehension, Los Alamitos, USA, 2003, pp. 294-295.

[Blo01] J. Bloch, Effective Java Programming Language Guide. Addison-Wesley, 1st edition, June 2001.

[BMMM98] W. J. Brown, R. C. Malveau, H. W. McCormick, T. J. Mowbray, AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis, 1st edition, Wiley, 1998.

[BNL05] D. Beyer, A. Noack, C. Lewrentz, Efficient Relational Calculation for Software Analysis, Transactions on Software Engineering, 31(2), 2005, pp. 137-149.

[BR99] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, 1999.

[Bro97] K. Brown, Design Reverse Engineering and Automated Design Pattern Detection in Smalltalk, M. Sc. Thesis, University of Illinois at Urbana-Campaign, 1997.

[BSL05] L. O'Brien, D. Smith, G. Lewis, Supporting Migration to Services using Software Architecture Reconstruction, in Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering in Practice (STEP'05), Washington, DC, USA, September 2005, pp. 222-229.

[BSV02] L. O'Brien, C. Stoermer, C. Verhoef, Software Architecture Reconstruction: Practice, Needs and Current Approaches, Technical Report CMU/SEI-2002-TR-024, Carnegie Mellon University, 2002.

[CDD+05] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Design Pattern Recovery by Visual Language Parsing, 9th European Conference on Software Maintenance and Reengineering (CSMR'05), 2005.

[Chi90] E. J. Chikofsky, J. H. Cross II, Reverse Engineering and Design Recovery: a Taxonomy, IEEE Software, 7(1), January 1990, pp. 13-17.

[CL] CodeLogic for Java, http://www.logicexplorers.com/CodeLogicJava.html

[Coo98] J. W. Cooper, The Design Patterns Java Companion, Addison-Wesley Design Patterns Series, October 1998.

[CZD+09] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, R. Koschke, A Systematic Survey of Program Comprehension through Dynamic Analysis, IEEE Transactions on Software Engineering, 35(5), September 2009, pp. 684-702.

[DDGR09] A. De Lucia, V. Deufemia, C. Gravino, M. Risi, Design Pattern Recovery through Visual Language Parsing and Source Code Analysis, The Journal of Systems and Software, 82, Elsevier, February 2009, pp. 1177-1193.

[DE07] J. Dietrich, C. Elgar, Towards a Web of Patterns, in Web Semantics: Science, Services and Agents on the World Wide Web, 5(2), Elsevier Science Publishers B. V., June 2007, pp. 108-116.

[DHK+04] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, C. Riva, Viewpoints in Software Architecture Reconstruction, in Proceedings of the 6th Workshop on Software Reengineering (WSR04), 2004.

[DHK+04b] A. van Deursen, C. Hofmeister, R. Koschke, L. Moonen, C. Riva, Symphony: View-driven Software Architecture Reconstruction, in Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), 2004.

[Doxy] Doxygen, http://www.stack.nl/~dimitri/doxygen/

[DR04] A. van Deursen, C. Riva, Software Architecture Reconstruction, in Proceedings of the 26th International Conference on Software Engineering (ICSE), 2004.

[DTD01] S. Demeyer, S. Tichelaar, and S. Ducasse, FAMIX 2.1— The FAMOOS Information Exchange Model, Technical report, University of Bern, 2001.

[Eclipse] Eclipse framework, http://www.eclipse.org/

[Ent] Sparx Systems Enterprise Architect, http://www.sparxsystems.com.au/

[FATM99] R. Fiutem, G. Antoniol, P. Tonella, E. Merlo, ART: an Architectural Reverse Engineering Environment, Journal of Software Maintenance: Research and Practice, 11(5), 1999.

[FGMP02] R. Ferenc, J. Gustafsson, L. Mueller, J. Paakki, Recognizing Design Patterns in C++ Programs with the Integration of Columbus and MAISA, in Acta Cybernetica, 15(4), 2002, pp. 669-682.

[FHFG08] L. J. Fülöp, P. Hegedűs, R. Ferenc, T. Gyimóthy, Towards a Benchmark for Evaluating Reverse Engineering Tools, in Proceedings of the 15th IEEE Working Conference on Reverse Engineering (WCRE 2008), Antwerp, Belgium, October 2008.

[FindBugs] FindBugs, http://findbugs.sourceforge.net/

[FM04] J. Fabry, T. Mens, Language-Independent Detection of Object-Oriented Design Patterns, Elsevier International Journal on Computer Languages, Systems & Structures - Proceedings of the ESUG 2004 Conference, 30(1-2), 2004, pp. 21-33.

[Fow99] M. Fowler, Refactoring – Improving the Design of Existing Code, 1st edition, Addison-Wesley, June 1999.

[GA08] Y. G. Guéhéneuc, G. Antoniol, DeMIMA: A Multilayered Approach for Design Pattern Identification, IEEE Transactions on Software Engineering, 34(5), September/October 2008, pp. 667-684.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994.

[GM05] Y. Gil, I. Maman, Micro Patterns in Java Code, in Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA '05), October 2005, pp. 97-116.

[GMW06] Y. G. Guéhéneuc, K. Mens, R. Wuyts, A Comparative Framework for Design Recovery Tools, in Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2006), 2006, pp. 123 – 134.

[GSZ04] Y. G. Guéhéneuc, H. Sahraoui, F. Zaidi, Fingerprinting Design Patterns, in Proceedings of the 11th Working Conference on Reverse Engineering, IEEE CS Press, November 2004, pp. 172-181.

[Gue05] Y. G. Guéhéneuc, PTIDEJ: Promoting Patterns with Patterns, in Proceedings of the 1st ECOOP Workshop on Building Systems using Patterns, Springer-Verlag, July 2005.

[GZ05] I. Gorton, L. Zhu, Tool Support for Just-in-Time Architecture Reconstruction and Evaluation: an Experience Report, in Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA, May 2005.

[HHHL03] D. Heuzeroth, T. Holl, G. Högström, W. Löwe, Automatic Design Pattern Detection, 11th IEEE International Workshop on Program Comprehension, May 2003, pp. 94 – 103.

[IEEE] IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems - Description, http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html

[JDeo] JDeodorant, http://java.uom.gr/~jdeodorant/

[JDepend] JDepend, http://clarkware.com/software/JDepend.html

[JDY07] D. Jing, L. Dushyant, Z. Yajing, DP-Miner: Design Pattern Discovery Using Matrix, 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, 2007 (ECBS '07), March 2007.

[JGraph] JGraph, http://www.jgraph.com/

[JHD] JHotDraw, http://www.jhotdraw.org/

[Jikes] Jikes Java compiler, http://jikes.sourceforge.net/

[JMP04] M. Jha, P. Maheshwari, T. K. A. Phan, A Comparison of Four Software Architecture Reconstruction Toolkits, Technical Report UNSW-CSE-TR-0435, The University of New South Wales, Sydney, Australia, 2004.

[Jus01] N. Jussien, e-Constraints: Explanation-based constraint programming, In Barry O'Sullivan and Eugene Freuder (Eds.), 1st CP workshop on User-Interaction in Constraint Satisfaction, Paphos, Cyprus, December 2001.

[KB04] C. Kaner, P. Bond, Software Engineering Metrics: What Do They Measure and How Do We Know?, 10th International Software Metrics Symposium (METRICS 2004), 2004.

[KBV03] R. Kazman, L. O'Brien, C. Verhoef, Architecture reconstruction guidelines, third edition, CMU/SEI-2002-TR-034, Carnegie Mellon University, Software Engineering Institute, 2003.

[KC99] R. Kazman, S. J. Carriere, Playing detective: Reconstructing Software Architecture from Available Evidence, Automated Software Engineering, 1999.

[KP96] C. Kramer, L. Prechelt, Design Recovery by Automated Search for Structural Design Patterns in Object Oriented Software, in Proceedings of Working Conference on Reverse Engineering (WCRE'96), Monterey, USA, 1996, pp. 208-215.

[Kru99] P. Kruchten, The Rational Unified Process: an Introduction, Addison-Wesley, 1999.

[KSRP99] R. K. Keller, R. Schaur, S. Robitaille, P. Paige Pattern-Based Reverse Engineering of Design Components, in Proceedings of the 21st IEEE International Conference on Software Engineering (ICSE), Los Angeles, USA, 226-235, 1999.

[Lan03] M. Lanza, Polymetric Views – A Lightweight Visual Approach to Reverse Engineering, IEEE Transactions on Software Engineering, September 2003.

[Lar04] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), October 2004.

[Licor] LiCoR, http://prog.vub.ac.be/research/DMP/soul/soul2.html

[LK94] M. Lorenz, J. Kidd, Object-oriented software metrics: a practical guide. Prentice-Hall, Inc. 1994.

[LMSB05] G. Lewis, E. Morris, D. Smith, L. O'Brien, Service-Oriented Migration and Reuse Technique (SMART), in Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice, (STEP'05), Washington, DC, USA, Spetember 2005, pp. 222-239.

[Lucene] Apache Lucene, http://lucene.apache.org/

[Mag06a] S. Maggioni, A New Approach to Design Pattern through the Use of Design Pattern Clues, M. Sc. Thesis, University of Milano-Bicocca, Milan, Italy, February 2006.

[Mag06b] S. Maggioni, Design Pattern Clues for Creational Design Patterns, Proceedings of the 1st International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE 2006), Benevento, Italy, October 2006.

[Mar95] R. C. Martin, OO Design Quality Metrics, An Analysis of Dependencies, ROAD Vol. 2 No. 3, 1995.

[Mar04] R. Marinescu, Detection Strategies: Metrics-based Rules for Detecting Design Flaws, in Proceedings of the 20th International Conference on Software Maintenance (ICSM '04), IEEE Computer Society Press, 2004, pp. 350-359.

[MATZ09] S. Maggioni, F. Arcelli, C. Tosi, M. Zanoni, Refining Design Pattern Detection through Design Pattern Clues, submitted to the Journal of Systems and Software, July 2009.

[McC90] W. McCune, Otter 2.0 (theorem prover), in Proceedings of the 10th International Conference on Automated Deduction, Kaiserslautern, July 1990, pp. 663-664.

[MGD+09a] N. Moha, Y. G. Guéhéneuc, L. Duchien, A. F. Le Meur, DECOR: A Method for the Specification and Detection of Code and Design Smells, IEEE Transactions on Software Engineering (TSE), 2009.

[MGD+09b] N. Moha, Y. G. Guéhéneuc, L. Duchien, A. F. Le Meur, A. Tiberghien, From a Domain Analysis to the Specification and Detection of Code and Design Smells, in Formal Aspects of Computing (FAC), 2009.

[MJS+00] H. A. Mueller, J. H. Jahnke, M. A. Storey, D. B. Smith, S. R. Tilley, K. Wong, Reverse Engineering: a Roadmap, in Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, ACM Press, 2000, pp. 47-60.

[Mül93] H. A. Müller, O. A. Mehmet, S. R. Tilley, J. S. Uhl, A Reverse Engineering Approach to System Identification, in Journal of Software Maintenance: Research and Practice 5, 4 December 1993, pp. 181-204.

[MWT95] H. Müller, K. Wong, and S. Tilley, Understanding Software Systems Using Reverse Engineering Technology, Object-Oriented Technology for Database and Software Systems, World Scientific, 1995.

[MXY01] H. Mei, T. Xie, F. Yang, JBOORET: an Automated Tool to Recover OO Design and Source Models, 25th Annual International Computer Software and Applications Conference (COMPSAC'01), 2001.

[Nie02] J. Niere, Fuzzy Logic Based Interactive Recovery of Software Design, in Proceedings of the 24th IEEE International Conference on Software Engineering (ICSE), Florida, USA, 2002, pp. 726-728.

[NNZ00] U. Nickel, J. Niere, and A. Zündorf, The FUJABA Environment, in Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland, 2000, pp. 742-745.

[NSW+02] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, Towards Pattern-Based Design Recovery, in Proceedings of the 24th International Conference on Software Engineering, Orlando, Florida, USA, 2002, pp. 338-348.

[PDP+07] D. Pollet, S. Ducasse, L. Poyet, I. Allaoui, S. Cimpan, H. Verjus, Towards a Process-Oriented Software Architecture Reconstruction Taxonomy, in Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR07), Amsterdam, The Netherlands, March 2007.

[PL06] N. Pettersson, W. Loewe, Efficient and Accurate Software Pattern Detection, 13th Asia-Pacific Software Engineering Conference (APSEC 2006), December 2006.

[PMD] PMD, http://pmd.sourceforge.net/

[Pre94] Wolfgang Pree, Meta Patterns - A Means For Capturing the Essentials of Reusable Object-Oriented Design, in Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag, 1994, pp. 150-162.

[Pre97] Wolfgang Pree, Essential Framework Design Patterns, Object Magazine, 1997.

[PSRN05] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, An Approach for Reverse Engineering of Design Patterns, Software and Systems Modeling, 4(1), Springer Verlag, April 2005, pp. 55-70.

[RSA] Rational Software Architect, http://www-01.ibm.com/software/awdtools/architect/swarchitect/

[SA4J] Structural Analysis for Java, http://www.alphaworks.ibm.com/tech/sa4j

[Sei] Software Engineering Institute (SEI), Community Software Architecture Definitions, http://www.sei.cmu.edu/architecture/start/community.cfm

[SJ98] J. Seemann, J.W. von Gudenberg, Pattern-Based Design Recovery of Java Software, ACM SIGSOFT Software Engineering Notes, 23(6), ACM Press, November 1998, pp. 10-16.

[SM95] M. D. Storey, H. Müller, Manipulating and Documenting Software Structures using SHriMP Views, in Proceedings of the 11th International Conference on Software Maintenance (ICSM'95), 1995.

[Smi02] J. McC. Smith, An Elemental Design Patterns Catalog, Tech. Rep. 02-040, Computer Science Department, University of North Carolina at Chapel Hill, December 2002.

[SO06] N. Shi, R. A. Olsson, Reverse Engineering of Design Patterns from Java Source Code, 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan, September 2006, pp. 123-132.

[SP03] R. Seacord, D. Plakosh, Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices (SEI Series in Software Engineering), 2003.

[SRBV06] C. Stoermer, A. Rowe, L. O'Brien, C. Verhoef, Model-centric Software Architecture Reconstruction, Software – Practice and Experience, 36(4), 2006.

[SS02] J. McC. Smith, D. Stotts, Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture, in Proceedings of the 27th Annual IEEE/NASA Software Engineering Laboratory Workshop, Greenbelt, MD, 2002, pp. 183-190.

[SS03] J. McC. Smith, D. Stotts, SPQR: Flexible Automated Design Pattern Extraction From Source Code, in Proceedings of the 2003 IEEE International Conference on Automated Software Engineering, Montreal QC, Canada, October, 2003, pp. 215-224.

[SSys02] E. Stroulia, T. Systä, Dynamic Analysis for Reverse Engineering and Program Understanding, in Applied Computing Review, 10(1), pp. 8-17, 2002.

[SW08] K. Stencel, P. Wegrzynowicz, Detection of Diverse Design Pattern Variants, 15th Asia-Pacific Software Engineering Conference (APSEC 2008), Dcember 2008.

[Swag] Swagkit, http://www.swag.uwaterloo.ca/swagkit/

[SYS06] K. Sartipi, L. Ye, H. Safyallah, Alborz: an Interactive Toolkit to Extract Static and Dynamic Views of a Software System, in Proceedings of the IEEE International Conference on Program Comprehension (ICPC06), Athens, Greece, 2006, pp. 256-259.

[TC06] N. Tsantalis, A. Chatzigeorgiou, Design Pattern Detection using Similarity Scoring, in IEEE Transactions on Software Engineering, 32(11), November 2006, pp. 896-909.

[TC09a] N. Tsantalis, A. Chatzigeorgiou, Identification of Move Methods Refactoring Opportunities, in IEEE transactions on Software Engineering 35(3), May/June 2009, pp. 347-367.

[TC09b] N. Tsantalis, A. Chatzigeorgiou, Identification of Refactoring Opportunities Introducing Polymorphism, accepted in the Journal of Systems and Software (JSS), September 2009.

[TC09c] N. Tsantalis, A. Chatzigeorgiou, Identification of Extract Method Refactoring Opportunities, in Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009), Kaiserslautern, Germany, March 2009.

[TPS96] S. R. Tilley, S. Paul, D. B. Smith. Towards a framework for program understanding. In IWPC, 1996.

[U4J] Understand for Java, http://www.softpedia.com/get/Programming/File-Editors/Understand-for-Java.shtml

[Wen03] L. Wendehals, Improving Design Pattern Instance Recognition by Dynamic Analysis, in Proceedings of the ICSE Workshop on Dynamic Analysis, Portland, USA, 2003, pp. 29-32.

[WT04] W. Wang, V. Tzerpos, DPVK - An Eclipse Plug-in to Detect Design Patterns in Eiffel Systems, in Proceedings of the Second Eclipse Technology Exchange: eTX and the Eclipse Phenomenon (eTX 2004).

[XML] Apache XMLBeans, http://xmlbeans.apache.org/

[XRay] X-Ray Eclipse plugin, http://www.eclipseplugincentral.com/Web_Links-index-req-viewlink-cid-1147.html

[YGS+04] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, R. Kazman, DiscoTect: a System for Discovering Architectures from Running Systems, in Proceedings of the 26th International Conference on Software Engineering (ICSE'04), Edinburgh, United Kingdom, May 2004.