Information Sciences 181 (2011) 1306–1324

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

A tool for design pattern detection and software architecture reconstruction

Francesca Arcelli Fontana, Marco Zanoni [*]

Università degli Studi di Milano-Bicocca, DISCo – Dipartimento di Informatica, Sistemistica e Comunicazione, 20126 Milan, Italy

A R T I C L E   I N F O

Article history:
Received 17 December 2008
Received in revised form 1 October 2010
Accepted 2 December 2010
Available online 13 December 2010

Keywords:
Reverse engineering
Software architecture reconstruction
Design pattern detection
Data mining techniques
Metrics computations

A B S T R A C T

It is well known that software maintenance and evolution are expensive activities, both in terms of invested time and money. Reverse engineering activities support the obtainment of abstractions and views from a target system that should help the engineers to maintain, evolve and eventually re-engineer it. Two important tasks pursued by reverse engineering are design pattern detection and software architecture reconstruction, whose main objectives are the identification of the design patterns that have been used in the implementation of a system as well as the generation of views placed at different levels of abstractions, which let the practitioners focus on the overall architecture of the system without worrying about the programming details it has been implemented with.

In this context we propose an Eclipse plug-in called MARPLE (Metrics and Architecture Reconstruction Plug-in for Eclipse), which supports both the detection of design patterns and software architecture reconstruction activities through the use of *basic elements* and metrics that are mechanically extracted from the source code. The development of this platform is mainly based on the exploitation of the Eclipse framework and plug-ins as well as of different Java libraries for data access and graph management and visualization. In this paper we focus our attention on the design pattern detection process.

1. Introduction

A software engineering research area that is getting more and more importance for the maintenance and evolution of software systems is reverse engineering [13,51]. A relevant objective of this discipline is to obtain representations of the system at a higher level of abstraction and to identify the fundamental components of the analyzed system by retrieving its constituent structures. Getting this information should greatly simplify the restructuring and maintenance activities, as we obtain more understandable views of the system and the system can be seen as a set of coordinated components, rather than as a unique monolithic block. Considering these components, particular relevance is given to design patterns [27].

In the context of reverse engineering and program comprehension in particular we focus our attention on the two activities of design patterns detection (DPD) and software architecture reconstruction (SAR).

The main objective of DPD is to gain better comprehension of a software system and the kind of problems which have been addressed during the development of the system itself. The presence of DPs can be considered an indicator of good software design, as design patterns are reusable for their self definition. Moreover, they are very important during the re-documentation process, in particular when the documentation is very poor, incomplete or not up-to-date.

The main objective of SAR is to abstract from the analyzed system's details in order to obtain general views and diagrams with different metrics associated with them. The extraction of such data helps the engineers to have a global understanding of the system and of its architecture.

* Corresponding author.
E-mail addresses: arcelli@disco.unimib.it (F. Arcelli Fontana), marco.zanoni@disco.unimib.it (M. Zanoni).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.12.002

Different tools for DPD have been proposed in the literature (e.g. [31,25,52,60,62]). They usually have problems in finding all the design patterns of the GoF catalogue [27]. Some tools recognize only a small subset of these patterns, but the main problem is that the results found contain many false positive DP instances and moreover they usually do not scale well when trying to analyze medium/large systems. Also for SAR different tools have been proposed (e.g. Codecrawler [47], Doxygen [33], SA4J [35], Codelogic [48], ARMIN [53] and Swagkit [71]), obtaining different views at different levels of abstraction, some exploiting only static analysis and other exploiting both static and dynamic analysis. Tools for SAR usually do not provide any DPD functionality.

The aim of this paper is to describe the project on which we are currently working on, called MARPLE (Metrics and Architecture Reconstruction PLug-in for Eclipse) developed to support both DPD and SAR activities, which constitute the two main modules of the project. MARPLE's architecture has been designed in order to be language independent, even if until now we have performed our analysis on Java systems.

Our approach to design pattern detection is based on the detection of design pattern subcomponents [6], which can be considered indicators of the presence of patterns. Currently we perform our analysis only exploiting static source code analysis: the Abstract Syntax Trees (ASTs) of the analyzed projects are parsed in order to obtain the structures we need for our elaboration, which we called *basic elements* (BE); we plan to extend our approach through the exploitation of behavioural analysis, both for DPD and SAR.

DPD activity may be seen as a specialization of the more general SAR activity: DPD provides information that is not directly obtainable by applying SAR techniques, but the results gained with SAR tools can also be useful for the DPD process. In fact, we can detect a design pattern through the DPD module and then check the results through some views offered by the SAR module. For this reason we found it interesting to start developing a tool which can perform both DPD and SAR.

MARPLE is conceived as an Eclipse plug-in. This choice was due to two main reasons: first of all, Eclipse is the most used open source development framework, it is supported by a wide community of developers, and it is strongly based on the concept of extending its functionalities through the implementation of plug-ins. The second reason resides in the fact that this platform encourages strong interaction among the various components that constitute the framework; therefore, the implementation of our plug-in is based on the exploitation of the functionalities of other plug-ins and modules of the Eclipse framework. This reuse of components improves and speeds up the development process.

The main objectives and contributions of this paper are:

- To describe the overall architecture of a tool that integrates the two activities of SAR and DPD: to our knowledge no open source tool for SAR also support design pattern detection;
- To design the software architecture of the tool, that can be easily extended by providing other functionalities useful for reverse engineering and program comprehension, such as anti pattern and code smell detection;
- To focus our attention in particular on DPD, by outlining a promising approach which exploits classification techniques that allows us to refine the results provided in a previous step which are characterized by very high recall values; we extend our previous work [4] and we experiment different classifiers;
- To outline how we try to face the well known variant problems for DPD through the detection of the sub-components, that we call *basic elements* in this paper.

For all these reasons, we are strongly motivated to work on our MARPLE project by extending the current two functionalities of DPD and SAR and by providing other new functionalities like those cited above.

In this paper we focus our attention on describing the results obtained through MARPLE on DPD and in particular on the detection of a specific design pattern, the Abstract Factory, in order to show the full detection process of a given design pattern and the results obtained on it. Moreover, we give examples of architectural views that we can reconstruct through the SAR module.

The paper is organized as follows: Section 2 briefly introduces some tools for design pattern detection and others for software architecture reconstruction; Section 3 introduces the overall MARPLE architecture describing the different modules, and briefly discusses about the opportunity of migrating MARPLE to a distributed environment; Section 4 presents some results about DPD and SAR modules. Finally, Section 5 gathers the conclusions and outlines possible future development.

2. Related works

Probably the only thing all research groups involved in reverse engineering agree on, is that manual inspection is not feasible or is very expensive and tool support is necessary. For example, to manually detect DPs and reconstruct the software architecture of a system without the aid of automatic procedures requires a preliminary study of the whole system, which can take a lot of time for large systems. The time required to understand the system to a sufficient extent and to become confident with it is certainly not linear with its size and might take so long that, when the task is done, the knowledge gained is already outdated. This is due to the relations within the parts, which increase in number and complexity as long as the size of the system increases; other causes are emergency repairs and wild maintenance, which have made things even worse by introducing additional coupling between the system's components.

In the next two subsections we briefly describe different DPD and SAR tools, outlining their principal features.

2.1. Design pattern detection

SPOOL [43] stands for Spreading Desirable Properties into the Design of Object- Oriented, Large-Scale Software Systems. The authors outline three possible ways of detection: manual, automatic, and semi-automatic, the first two of which are supported in SPOOL. Automatic recovery is implemented through queries to a previously generated repository.

The Pat system [55] transforms C++ source code into PROLOG facts and matches them against pattern definitions given as PROLOG statements. The approach is based on first-order logic and constraint solving techniques. The authors claim that this system can detect many pattern instances without missing any and with few false positives. Although we cannot verify the truth of these assertions, we can easily imagine the high computational costs of this approach. In addition, only header files are examined, so no behaviour is available for them.

PTIDEJ tool [29] uses constraint solving with explanation. Explanation consists in first detecting instances matching DP definitions exactly and then, by relaxing some constraints, entities that are less and less similar to DPs.

The MAISA tool [54] uses a library of patterns defined as sets of variables, representing the patterns roles, and unary or binary constraints over them. A solution to the constraint satisfaction problem is a possible instantiation of these variables. To be able to detect instances which do not exactly correspond to the definition, it is possible to relax it by removing some constraints, but the number of candidates tends to increase quickly. A similar approach has been used in the Columbus tool [25], in which patterns are defined by using an XML based language called Design Pattern Markup Language (DPML) and searched for in an Abstract Semantic Graph (ASG) generated by the tool itself.

Web of Pattern [21] uses an approach to the formal definition of design patterns based on the web ontology language (OWL). The authors present their prototype that accesses the pattern definitions and detects patterns in Java software. The tool connects to a pattern server, downloads and scans the patterns, translates them into constraints, and resolves these constraints with respect to the program to be analyzed.

Pinot [60] is a modification of Jikes [36], IBMs Java compiler, developed to detect various design patterns based on static rule-based analysis. The authors present an interesting, but arguable, reverse engineering oriented reclassification of the GoF design patterns into different categories: patterns provided by the programming language, syntax-based patterns, semantic-based patterns, domain-specific patterns and patterns that are mere generic concepts.

Fujaba [52], use fuzzy logic combined with Fujaba Abstract Syntax Graphs to cope with two different types of pattern variations, design variants and implementation variants. They address the former with ASGs, by modeling various design variants with different graphs, and then handle implementation variants by defining a set of fuzzy rules together, giving the degree of belief that a pattern is found at a certain location in the program.

SPQR [62] uses Elemental Design Patterns (EDPs) and a system for logical calculus, the ρ-calculus, to detect DPs. The authors claims that the tool can detect design patterns, but at present no evidence is provided, and it is not possible to verify the results.

DeMIMA [31,41] is an approach to semi-automatically identify micro-architectures that are similar to design patterns in source code and to ensure the traceability of these micro-architectures between implementation and design. DeMIMA consists of three layers: two layers to recover an abstract model of the source code, including binary class relationships, and a third layer to identify design patterns in the abstract model. Through the use of explanation-based constraint programming, DeMIMA ensures 100% recall on an experimentation on five systems.

In [69] a design pattern detection methodology is proposed, based on similarity scoring between graph vertices. The approach has the ability to also recognize patterns that are modified from their standard representation and exploits the fact that patterns reside in one or more inheritance hierarchies, reducing the size of the graphs to which the algorithm is applied. Evaluation on three open-source projects demonstrated the accuracy and the efficiency of the proposed method which has been described in the paper.

Several other tools and approaches have been proposed and described in the literature like those in [66].

We should in any case observe that to undertake a comparison among design patterns detection tools is certainly a difficult task. Benchmark proposals for the evaluation of design pattern detection tools have been presented in [11,26], but a standard benchmark platform is not yet available. While in [32] a general framework for the comparison of design recovery tools (hence not only for the comparison of design motif detection tools) is proposed, illustrating how the framework can be applied for the comparison of two different systems. Moreover in [30] a repository of pattern-like micro-architectures called P-MARt has been defined, serving as a baseline to assess the precision and recall of pattern identification tools. Another interesting approach is described in [44] where the authors compare different design pattern detection tools, and they propose a novel approach based on data fusion, built on the synergy of proven techniques, without requiring any re-implementation of what is already available.

2.2. Software architecture reconstruction

CodeCrawler [47] is a tool developed to support reverse engineering, in particular program comprehension through a combination of metrics and architectural views of the analyzed system. The tool is based on the concept of polymetric views [46] that are bi-dimensional visualizations whose nodes represent the system's entities, and the edges identify relations among entities; the visualizations are enriched by the representation on each node of a maximum of five metrics. Codecrawler evolved in X-ray [50] and in Codecity [79].

SHriMP [65] (Simple Hierarchical Multi-Perspective) is an information visualization and navigation system, which can be exploited for viewing information extracted from a system. When it is employed for reconstruction, this tool can assist the user in generating high level architectural views of a system, through the grouping and the manual aggregation of the elements of a graph. SHriMP exists as an autonomous tool, taking in input RSF (Rigi Standard Format), GXL (Graphic eXchange Language) and XMI (XML Metadata Interchange) files, or as an Eclipse plugin, having the name of Creole.

SA4J (Structural Analysis for Java) [35] is a multiplatform instrument developed by IBM for the analysis of the structural dependencies among the packages of a Java project, and the measuring of the stability of both single elements (classes, interfaces, packages) and the whole system. It calculates different dependencies and stability metrics of the system and automatically reconstructs a series of architectural diagrams, other than helping the user in identifying potential design problems and anti-patterns that are potentially unstable or poorly designed pieces of code.

Codelogic [48] is a tool for reverse engineering of Java and C# code. It is a multiplatform software, and it exists as standalone application or as plugin for many IDEs (Eclipse, JBuilder, JDeveloper, IntelliJ IDEA). Codelogic is able to directly analyze the source code, reconstructing the diagrams that describe both static and dynamic aspects of the system.

Understand for Java [59] 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. Understand for Java is mainly focused on the computation of metrics for software quality and complexity evaluation.

Bauhaus [57] provides different methods and tools to analyze and recover the software architecture of legacy systems; in particular it supports the identification of re-usable components and the estimation of change impact.

Barrio [22] is used for cluster dependency analysis, by using the Girvan–Newman clustering algorithm to compute the modular structure of programs. This is useful in assisting software engineers to redraw component boundaries in software, in order to improve the level of reuse and maintainability.

Many other tools supporting software architecture reconstruction and program comprehension are available such as: Coverity [20] is a commercial tool that provides different architecture analysis views, by exploiting both static and dynamic analysis; Doxygen [33] is a documentation system for numerous programming languages, and it is also able to extract the code structure from undocumented source files, by means of graphs and diagrams showing the systems entities and their relationships, using an UML-like visualization. Rational Software Architect [37] and Enterprise Architect [64] are two commercial tools that provide different functionalities useful for program comprehension, such as the reconstruction of static and dynamic UML diagrams; JDepend is used for the generation of design quality metrics, which allow to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability, but it does not provide any view that helps in understanding the architecture of the analyzed systems; ARMIN [42] is a tool for architectural reconstruction and mining which provides different representations that are useful as a starting point for reengineering a system to a new, desired architecture. Other commercial tools are Sotograph [12], which can test the conformance to a certain target architecture, and Lattix [58], which organizes architectural information using the Dependency Structure Matrix form.

2.3. Concluding remarks

We would like to outline that to our knowledge no open source tool is available supporting both DPD and SAR, which is the main objective of the MARPLE project.

As we have outlined in the introduction, we aim to better support SAR through the information gained during DPD. Moreover, we aim to obtain interesting and useful views by exploiting the sub-components, or micro-architectures that we use in MARPLE for DPD purposes.

A problem of having so many proposals and tools for DPD and SAR is that they often report different results, and it is difficult, as we observed before, to make comparisons because there is not a widely used exchange standard for these results; only recently did a group of authors propose an organic model [45] to represent and store DPD information. Experimenting some tools we found many false positive and false negative results, and problems of scalability. Through our approach for DPD we aim to overcome some of these problems. We detect DPs through the detection of subcomponents, called basic elements, which we use to identify all of the potential DP candidates according to a given rule (see Section 3.2), this way trying to face the variant problem which usually occurs during DPD due to the different possible implementation variants of DPs. Then we refine the results through data mining techniques (see Section 3.3).

3. An overview on MARPLE

An overview of the principal activities performed through MARPLE is depicted in Fig. 1 that shows the general process of data extraction, of design pattern detection, of software architecture reconstruction and of results visualization.

The information that is used by MARPLE is obtained by an Abstract Syntax Tree (AST) representation of the analyzed system.

DPD and SAR both receive the same set of basic elements and metrics that have been found inside the system, collected in an XML file. By basic elements we mean a set that is formed by Elemental Design Patterns (EDPs) [61], design pattern clues [49] and micro patterns [28] that are intended as the basic information we exploit as hints for the presence of design patterns

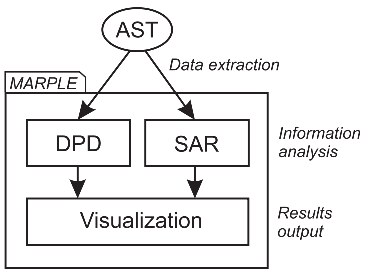


Fig. 1. An overview of the general process.

inside the code and as basic relationships that may connect two or more classes in terms of object creation, method invocation or inheritance. Fig. 2 depicts the overall architecture of the MARPLE project.

The architecture is constituted of five main modules that interact with one another through XML data transfers. The five modules are the following:

- The *Information Detector Engine* which collects both basic elements and metrics starting from an AST representation of the source code of the analyzed project;
- The *Joiner*, that extracts all the potential design pattern candidates which satisfy a given definition, working on the information extracted by the *Information Detector Engine*;
- The *Classifier*, which tries to infer whether the architectures detected by the *Joiner* could effectively be realizations of design patterns or not. This module helps to detect possible false positives identified by the *Joiner* and to evaluate the similarity with correct design pattern implementations, by assigning different confidence values;
- The *Software Architecture Reconstruction* module, which obtains abstractions from the target project basing on the elements and metrics extracted mainly by the *Information Detector Engine*, but also directly from the ASTs of the analyzed system;
- The activity of *Output generation* provides an organic view of the project analysis results. Through this activity, the user will see both the results produced by the detection of design patterns and the views provided by the SAR module.

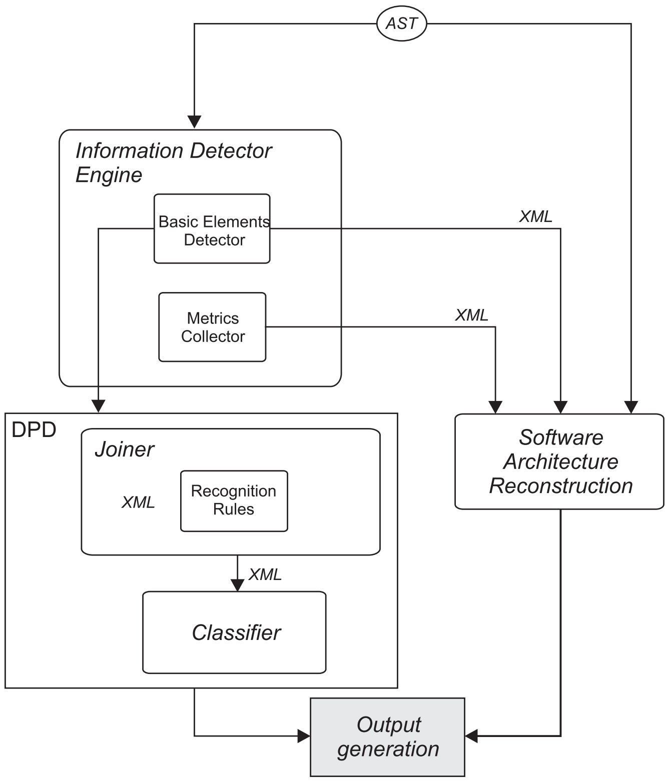


Fig. 2. The architecture of MARPLE.

Each of these modules takes part in different steps of computation (see Fig. 2): modules are not necessarily used for both DPD and SAR, but on the contrary some modules are specialized and used only in one of these activities. Table 1 reports the modules involved in each step of computation both for DPD and SAR, and the type of information produced as output from these two activities.

As we have outlined, the MARPLE project leans on the Eclipse framework and hence many functions did not need to be rewritten, but have been implemented by extending the core concepts provided by the platform. Obviously, many functionalities have also been implemented by using third party libraries, like XML data access or graph representations. As it appears clear from Fig. 2, all the modules work on XML files that come from some previous modules. Each of these modules works on these files, both for reading and writing, using the Apache XMLBeans library [3].

In the following, we will discuss the components we exploited in the implementation of each module constituting MARPLE, discussing the reasons about the choices we made and the effectiveness of these components in pursuing our objectives.

3.1. The information detector engine module

Currently, the *Basic Elements Detector* (BED) sub-module has been completely developed and tested. The basic elements are extracted by visitors that parse an AST representation of the source code, each of them returning instances of the basic elements if the analyzed classes or interfaces actually implement them. The information is acquired statically and is characterized by 100% rate of precision and recall. This value is due to the fact that these kinds of structures are meant to be *mechanically recognizable*, i.e. there is always a 1-to-1 correspondence between a basic element and a piece or a set of pieces of code. In other words, the basic elements are not ambiguous (as on the contrary design patterns may be), and once a basic element has been specified in terms of the source code details that are used to implement it, the basic element can be correctly detected.

By their definition, basic elements can be recognized in a software system using different techniques, and the only constraint is that they have to refer to one or two types in the system, as detailed in Section 3.2. Examples of exploitable techniques are data flow analysis, call graph analysis, dynamic analysis or simulation, symbolic execution, points to analysis, etc. Currently we consider basic elements that we are able to detect only exploiting static analysis.

We think that the definition of the elements we use to abstract the representation of the system is our contribution to mitigate the variants problem: each basic element is recognized in an exact manner, but it possibly synthesizes more than one low level structure bringing the same concept. By exploiting this kind of information we can represent the system in a way that explicitly models the concerns contained in the source code.

Our actual implementation of the BED module is based on AST analysis, and exploits the JDT library that provides all the classes and interfaces, which can be used to access the ASTs contained in a set of projects of the same workspace. The BED module runs the analysis on a set of selected projects. The basic elements are collected by a set of visitors, invoked sequentially on the ASTs of the classes constituting the project and the visitor work only on those nodes that may contain the information they are able to detect. For example, the visitors that look for method call EDPs only analyze nodes that represent a method invocation, i.e. instances of the *MethodInvocation* class.

The results coming from the visitors, i.e. the instances of basic elements that have been found inside the project, are then stored in an XML file. The BED module has been developed also for the .NET environment [5].

As far as the *Metrics Collector* sub-module is concerned, the evaluation of some object-oriented metrics that are useful for SAR has been implemented. These metrics are exploited in the generation of some of the architectural views described in the *Software Architecture Reconstruction Module* sub-section.

3.2. The Joiner module

In the *Joiner* module no particular third party technologies have been used. Nonetheless, this module does not handle the system through its AST representation, but it manages it as an *Attributed Relational Graph* (ARG), where the set of vertex corresponds to the set of types (i.e. classes and interfaces) the project is constituted of, while the edges are the set of basic elements that connect the types with one another. In fact, each basic element can be seen as a relationship between a type and another one (therefore depicted as an edge between two nodes of the graph), or as a relationship between a class and itself (depicted as a self loop on a graph node).

Table 1
The modules involved in the various steps.

	Data extraction	Information analysis	Output generation
DPD	Inf. detector engine	Joiner Classifier	XML Design pattern diagrams
SAR	Inf. detector engine AST	SAR module	Abstracted views Metrics

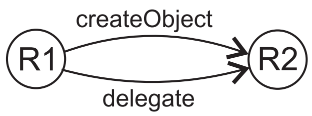


Fig. 3. An example of Joiner rule.

We think that this kind of basic element modeling can be also useful for representing some kind of behavioural information, such as the resolution of polymorphic method calls.

The system graph representation is directly derived from the output generated by the *Information Detector Engine*. As we have briefly anticipated, this module tries to extract architectures that match a target structure, defined in terms of *Joiner rules*. A *Joiner rule* is a graph that collects roles and basic elements (edges) that must be present among the roles in order to satisfy the rule. In particular, we have to define rules to extract candidate design pattern instances. In this way the roles in the rule are the roles of the target design pattern. For example, if we want to extract the couple of roles ($R1$,$R2$), where $R1$ has a *create object* and a *delegate method call* to $R2$, we may represent this rule as shown in Fig. 3.

The *Joiner* module tries to match and extract this kind of architecture from the graph representing the system through an ad hoc graph matching algorithm. The algorithm has been demonstrated to have linear complexity in the number of the classes of the system. All the details of the algorithm and the complexity demonstration can be found in [80].

The matches in the graph are also grouped in order to identify single pattern instances having more than one class for the same role. For example, if we want a couple of roles ($R1$,$R2$) where $R1$ must extend $R2$, we can impose that the results will be grouped by the assignment of $R2$, obtaining tree-wise instances, each one formed by an assignment of $R2$ and a set of related $R1$ assignments. The model that fully implements these basic concepts has been also proposed for generic pattern instance modeling in [7].

The architectures extracted by the Joiner are then inspected by the *Classifier* module, which tries to infer whether they can represent instances of design patterns or not.

3.3. The classifier module

The *Classifier* module takes all the candidate design pattern instances and tries to evaluate their grade of similarity with the searched design pattern, in order to be able to rank them. Fig. 4 shows an example of the classification process.

The rational behind the approach is that the Joiner module finds all the instances matching an exact rule; this rule is written trying to keep it very general, so the matching tends to produce a large number of instances (having very high recall), and many of them are false positives. However, the returned instances also carry a lot of information that the recognition rule does not use: all the basic elements found in the classes belonging to the instances. A classifier has the possibility to choose the right instances among the ones extracted by the Joiner module, exploiting the basic elements not used by the Joiner. This approach has the benefit of submitting a smaller number of candidates to the classifier, but with a high percentage of true instances, instead of calculating all the possible pattern instances in the analyzed system. In this way the performance estimation is more accurate and the dataset contains a smaller quantity of noise, because it is focused only on a specific subset of the domain. This is one of the principal features characterizing our approach, and differentiating it from other approaches in the literature.

In some cases the same pattern can have different well known structural alternatives; in these cases through our approach we consider different alternatives as different patterns, using a Joiner rule and a classifier training for each of them. When each rule defines the same set of roles, the user interface should group different alternatives to the user in order to simplify the user interaction; this last aspect does not belong to the core of the detection process and it is currently under development.

Through our current approach, depicted in Fig. 4, we generate every possible valid mapping $\{(R1,C1),(R2,C2),\ldots,(Rn,Cn)\}$ for each pattern instance, where each Ci is the class that is supposed to play the role Ri inside the pattern. These mappings are all of fixed size (an element for each pattern role) and each class has a fixed number of features, where the features are the basic element retrieved in the class. In this way each mapping can be represented as a vector of features whose length is given by ($num\_features * num\_roles$). These vectors are grouped by a clustering algorithm, producing k clusters; each pattern instance is represented as a k-long vector, having in each position i the absence/presence of the ith mapping. Since we know that an instance is a DP or not directly from the training set, we can enqueue to each vector the class attribute and use the resulting dataset for the training of a supervised classifier.

In the *Classification* Module we used the clustering and classification algorithms provided in Weka [70]. All the details of classification process can be found in [68].

3.4. The software architecture reconstruction module

One of the objectives of MARPLE is to support the user with the visualization of abstractions about the analyzed systems. Currently, the SAR module generates six kinds of views on a system:

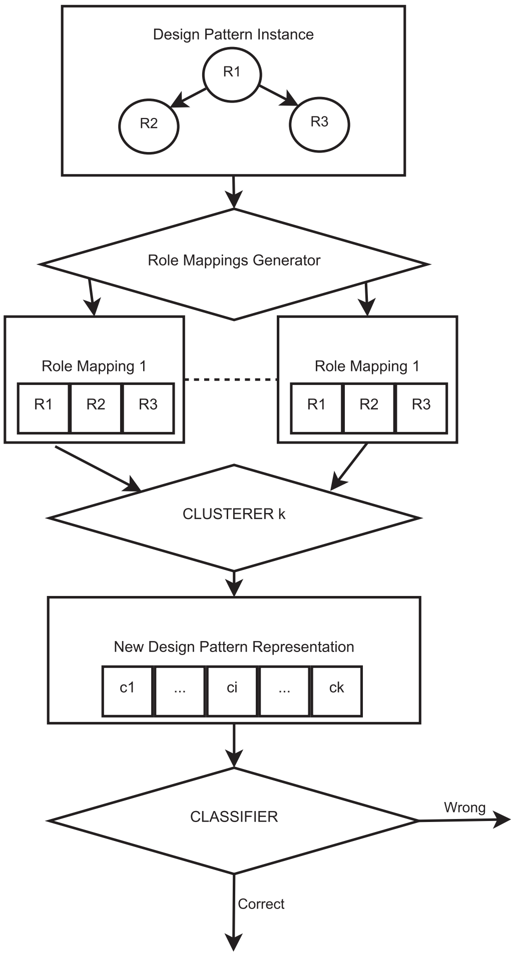


Fig. 4. Classification process.

- The *package diagram* of all the packages that form the analyzed system;
- The *class compact diagrams* of each package constituting the system. In this view, all the classes and interfaces belonging to the package are shown as a single graph, where the nodes correspond to the classes and interfaces, while the edges are the relationships connecting them;
- The *class extended diagrams* of each package constituting the system. This view is characterized by many graphs, one for each class or interface belonging to the package. Each graph reports only the relationships its subject class or interface has with the other classes or interfaces that constitute the system. In this way, the graphs will not be overwhelmed with a huge number of edges, letting the users focus on single classes without minding to the rest of the system;
- A *system complexity* view, giving an overview of the size and the structure of the classes of the system; it is similar to the one provided by CodeCrawler [47];
- A *type graph* of the entities constituting the project, similar to the *class graph* of Doxygen [33]; With the term *type* we mean both classes and interfaces, according to the JDT API specifications;
- A *class blueprint* diagram showing methods, attributes and relationships belonging to each class, like the one available in CodeCrawler.

These views are obtained by different kinds of information: the package and the class diagrams exploit the output coming from the *Basic Elements Detector*. More specifically, the SAR functionalities related to the generation of the *class compact* and of the *class extended* diagrams are achieved only through the analysis of the *elemental design patterns* detected by the *Basic Elements Detector* module. These elements revealed themselves very useful for the identification and definition of the relationships that are typical of class diagrams and which link the various classes constituting the analyzed project. These relationships, underlining the architectural constraints, let the users have a general overview of the classes structures and aggregations. The remaining views also exploit the *Metrics Collector* output, which gathers some common object oriented metrics starting from a further analysis of the ASTs. Namely, the *system complexity view* is based on the inheritance relationships among the classes constituting the system, while the complexity of each single class is measured in terms of their number of attributes (NOA), number of methods (NOM) and the class lines of code (WLOC); the *type graph* view is built by analyzing the inheritance, implementation, association and containment relationships of each class with the rest of the system; finally, the *class blueprint* diagram reports for each method the number of invoked methods (NI) and the method lines of code (LOC), and for each attribute the number of local (NLA) and global (NGA) accesses.

In order to show the results we used the GEF (Graphical Editing Framework) Eclipse plug-in [67]. GEF allowed us to take advantage of rich representation mechanisms and of the MVC (Model-View-Controller) pattern, which allowed us to maintain the underlying model (i.e. the ASTs of the analyzed project) separated from the implemented view.

3.5. Distributed MARPLE

There is a parallel project under development that allows us to use MARPLE in a client–server environment (see Fig. 5). This project has been developed because loading the projects ASTs using the Eclipse APIs required a large amount of memory (i.e. the AST of Batik, a system composed by 1643 Java files, required about 1700 Mb of memory). The distributed version splits the classes of the system into k sets and the BED nodes analyze only their own set. This solution allows us to reduce the memory requirement for the nodes; we also use this solution in the normal version of MARPLE serializing the analysis of each set using only one BED instance.

Another problem that convinced us toward the development of the distributed version is the computational requirement of the classifier module. It is well known that some classification algorithms require a long time in order to classify their data set.

For all of these reasons, we decided to implement a distributed version in order to allow users to run their analysis through the internet on the elaboration server and to asynchronously check the results of the elaboration. This project is based on the J2EE v.5 platform and precisely on the Glassfish [39] application server.

The system architecture (see Fig. 5) is composed of four main modules:

Client: this module is developed as an Eclipse plugin (maintaining MARPLE's interface) and allows users to use the elaboration service. Users using this service can send their projects, they can check current elaboration status, and they can also manage (retrieve, modify, delete) all the already performed analyses.

Server: this module is developed on a J2EE Application Server, and it implements the middle layer of the system. It also implements the Web Service in order to receive users requests, manages the Persistence Backend (Database) that

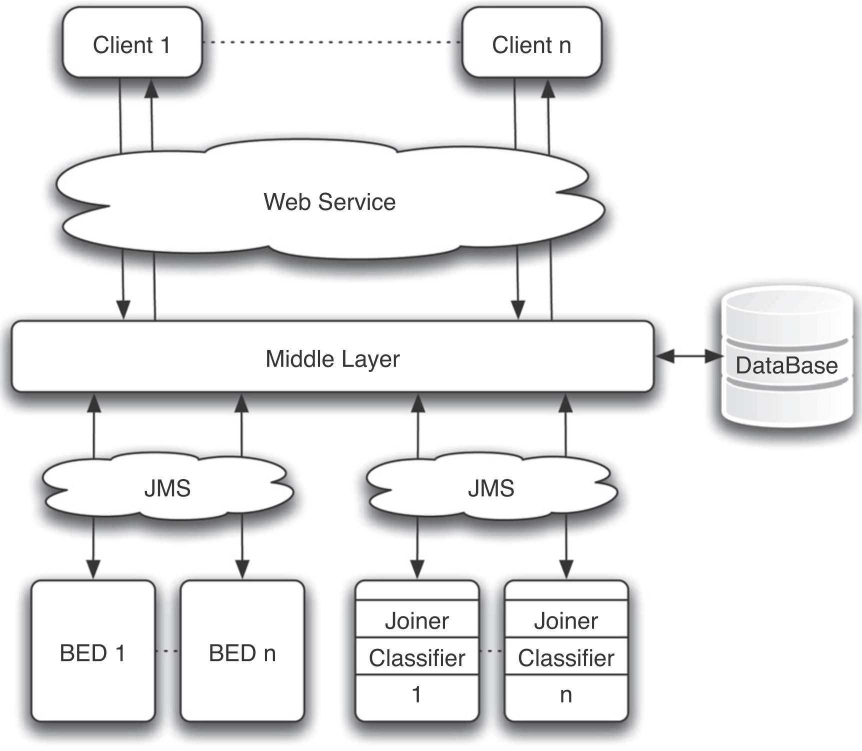


Fig. 5. The architecture of distributed MARPLE.

contains all the information of the users, and finally starts the elaboration of a project. When an elaboration starts, first the server calls in parallel all the BED Nodes, and at the end of their elaboration it calls in parallel all the Joiner–Classifier Nodes. This job serialization is necessary because, in order to go in execution, the Joiner–Classifier node has to own all the detected basic elements.

BED Node: this module receives from the Server, through JMS, a set of classes and the entire project source code; then it simply runs the BED on this set and returns to the server the detected basic elements.

Joiner–Classifier Node: this module receives from the server through JMS a rule specifying the pattern to find and all the detected basic elements. Next it sequentially runs the Joiner and the Classifier Module and then it returns to the Server all the found pattern instances with their classification values.

4. Experimental results for DPD

In this section we will provide some examples of outputs generated by the various modules of MARPLE on a set composed by different systems of different size, as reported in Table 2, which reports the size measured in number of compilation units and lines of code; in Eclipse internals a compilation unit is simply a Java file. We created this set with the aim of having a reliable dataset, taken from both the industry and academic worlds. We describe in detail as an example the results obtained for the detection of the Abstract Factory DP, and we show the dataset tailored for the detection of this pattern.

The projects were selected searching for:

- Example implementations of the Abstract Factory pattern;
- Open source projects, taken from Sourceforge, declaring the use of an Abstract Factory pattern implementation in their documentation.

All the experiments are executed using a 2Ghz notebook processor.

4.1. Results for the information detector engine module

The *Information Detector Engine* module, as we described before, is composed of two sub-modules: a *Basic Elements Detector* and a *Metrics Collector*: the first has to collect the basic elements on the classes, and the second calculates metrics on the code that will be useful for SAR purposes.

Table 2
Basic element detector performance and size results.

Project name	CU	LOC	Time (s)	Mem. (MB)	Num of BEs
AF-earthlink	1	⩽100	5	2	54
AF-Java.net	5	⩽100	4	9	30
jboot	6	⩽100	5	1	88
AF-WikiPedia	7	⩽100	5	7	36
AF-rice	8	⩽100	5	5	81
AF-vico.org	10	⩽100	5	5	44
AF-itec	10	⩽100	5	5	73
JVending-Registry-CDC	11	⩽100	6	3	174
AF-c-sharpcorner	11	⩽100	6	8	59
AF-fluffycat	12	⩽100	5	4	104
AF-apwebco	13	⩽100	6	5	69
AF-Javapractises	13	⩽100	5	6	94
AF-Gamma	21	⩽100	6	8	43
Dynamicdispatcher	23	593	6	17	528
Ehcache	86	10846	12	83	4820
Fdsapi	95	6372	10	46	4325
Sunxacml	155	16325	10	73	4702
Doubletype	175	46484	34	190	24201
Bexee	191	7899	16	115	3544
Wasa	213	9147	10	86	5597
Wstx	241	10574	15	130	16541
GroboUtils	285	20389	15	89	10648
Nodal	300	11203	19	163	18116
Rambutan	328	8898	15	124	6946
Sparql	332	20679	16	136	11919
Infovis	448	45585	31	299	25929
Fipaos	459	50615	28	278	24927
Mandarax	514	31338	26	251	16905
JasperReports	798	91161	58	671	49082
Batik	1643	141554	143	1725	93714

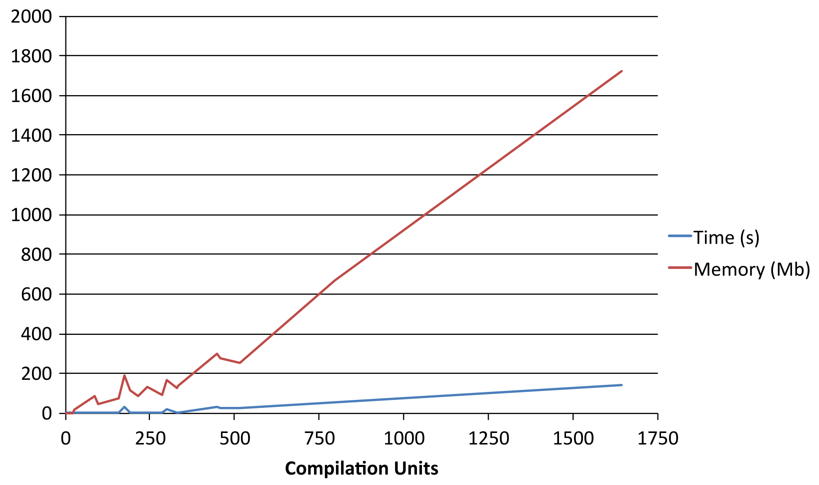


Fig. 6. Relationship between memory consumption, elapsed time and number of compilation units.

The detection of basic elements is the first step of our approach to Design Pattern detection, so we analyzed our dataset with the *Basic Element Detector*, and in Table 2 we report performance and size indicators for each system. We can easily observe from Table 2 that Memory Consumption and Elapsed Time are approximately in a linear relationship to the number of Compilation Units (CU) (see Fig. 6): this means that the detector scales linearly in the size of the analyzed system.

The main objective of the *Basic Elements Detector* is the extraction of information to be used in the further steps of computation for both DPD and SAR. The analysis of the results provided by this module are discussed in the next Sections.

4.2. Results for the Joiner module

The detection of design pattern candidates is the second step of our approach to design pattern detection, so we analyzed our dataset, or better the result of the BED module on our dataset, with the *Joiner* module. In Table 3 we report some performance and size indicators for each system.

Since the dataset has been tailored for the detection of the Abstract Factory design pattern, we tested only the Abstract Factory detection rule. The application of this rule gave us 69 correct instances on a total of 346 candidate pattern instances (see Table 3), therefore the Joiner module has a precision of about 20% on this dataset. This is not a high precision value, due to two main reasons: the first is that the recognition rule has been conceived to detect almost all the pattern instances, so it tends to maximize recall at the expense of precision; the second is that there is a peculiar project (Nodal) that produces a high number of false positives and three projects that contain only false positives, producing a quite unbalanced dataset. All the results have been also manually verified.

We proved [80] that the algorithm in the worst case has linear complexity in the number of the classes. Empirically we also noticed that on our dataset the detection time is approximately in a linear relation to the number of found instances, as shown in Fig. 7.

4.3. Results for the classifier module

The classification of design pattern candidates is the third step of our approach to design pattern detection, so we analyzed our dataset, or better the result of the Joiner on our dataset, with the *Classifier* module. The resulting dataset is composed of about 20% of instances having positive class and about 80% of instances having negative class.

In our experimentations we tried many numbers of clusters (NC) and a group of classifiers chosen through a manual data exploration using weka. All experiments are realized using Repeated Cross Validation with 10 folds and 10 repetitions. Below we show some performance indicators.

The classifiers we tested are the following:

ZeroR: a classifier that predicts the mean (for a numeric class) or the mode (for a nominal class);
NB: Naïve Bayes classifier [81];
J48: C4.5 classifier [56];
SMO: Support Vector Machine classifier [19];

The first indicator we show is the precision (see Table 4). The best classifier is the SMO-3 but SMO-2 gave a good performance too (see Fig. 8); the worst, excluding ZeroR, is the naïve bayes. The ZeroR classifier's precision is 0 because it classifies using the majority class, so it assigns all the instances to the negative class.

Table 3
Joiner performance and size results.

Project	# of BEs	Classes	Found	Valid	Precision (%)	Time (ms)
AF-earthlink	54	7	1	1	100.00	10
AF-WikiPedia	36	7	1	1	100.00	9
AF-rice	81	10	2	1	50.00	13
AF-vico.org	44	10	1	1	100.00	27
AF-itec	73	10	1	1	100.00	13
AF-c-sharpcorner	59	11	1	1	100.00	12
AF-fluffycat	104	16	1	1	100.00	13
AF-apwebco	69	13	1	1	100.00	12
AF-Javapractises	94	13	1	1	100.00	13
AF-Gamma	43	23	1	1	100.00	18
Dynamicdispatcher	528	47	2	2	100.00	19
Ehcache	4820	104	4	4	100.00	110
Fdsapi	4325	122	7	0	0.00	218
Sunxacml	4702	162	8	4	50.00	201
Doubletype	24201	357	9	2	22.22	300
Bexee	3544	201	20	2	10.00	237
Wasa	5597	303	3	0	0.00	97
Wstx	16541	257	23	6	26.09	220
GroboUtils	10648	511	16	6	37.50	158
Nodal	18116	602	109	4	3.67	2388
Rambutan	6946	488	33	6	18.18	936
Sparql	11919	429	32	0	0.00	1321
Infovis	25929	644	32	15	46.88	1068
Fipaos	24927	747	16	5	31.25	257
Mandarax	16905	621	21	3	14.29	388
Total	180305	5715	346	69	19.9	8058

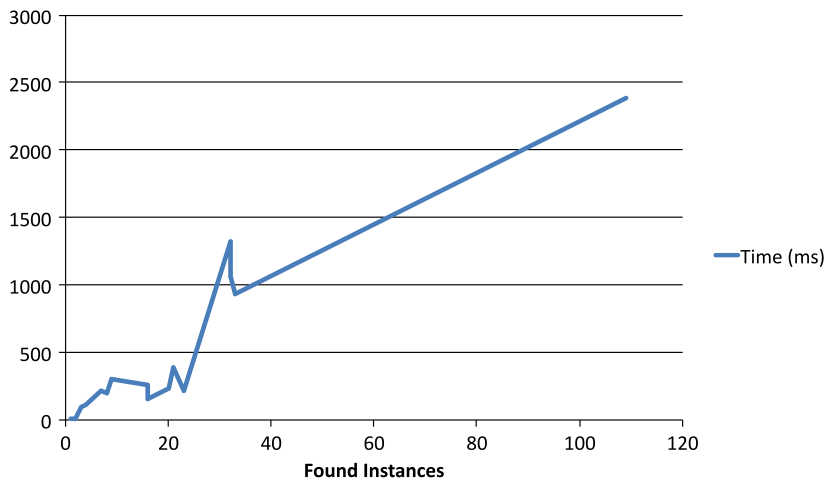


Fig. 7. Relationship between time and number of found instances.

The second indicator we show is the recall (see Table 5). The results evidence that the best classifier is SMO-2 (see Fig. 9), even if the value is not as high as for the precision, reaching about 60%; we think that here there is a way to improve the performance, tuning the classifiers' representations and parameters.

The ZeroR classifier's recall is 0 for the same reason explained for the precision; the worst classifiers are the J48 trees. The best precision value is given by the Support Vector Machines on the maximum numbers of clusters.

4.3.1. Comparison with other tools

It is not possible to make a real comparison with other DPD tools, since a concrete benchmark for this comparison is not yet available. In this context we proposed a model to be used for DPD benchmarks [7]; other studies have been made in this direction, as those cited in Section 2.1.

We compared our results with the results obtained through a tool for DPD called Web of Patterns (WoP) [21]. We used a significant part of the dataset shown in Table 3 and we observed (see Table 6) that MARPLE has better performances

Table 4
Experiment results: precision.

NC	NB	J48-1	J48-2	J48-3	SMO-1	SMO-2	SMO-3
400	0.36	0.53	0.52	0.42	0.54	0.55	0.62
500	0.36	0.53	0.54	0.52	0.68	0.68	0.79
600	0.38	0.54	0.53	0.46	0.63	0.63	0.79
700	0.32	0.58	0.55	0.49	0.67	0.66	0.77
800	0.29	0.51	0.53	0.42	0.75	0.74	0.77
900	0.25	0.66	0.51	0.51	0.76	0.75	0.8
1000	0.15	0.45	0.57	0.33	0.75	0.75	0.78
1200	0.1	0.52	0.6	0.32	0.7	0.71	0.68
1400	0.13	0.36	0.59	0.26	0.58	0.58	0.61

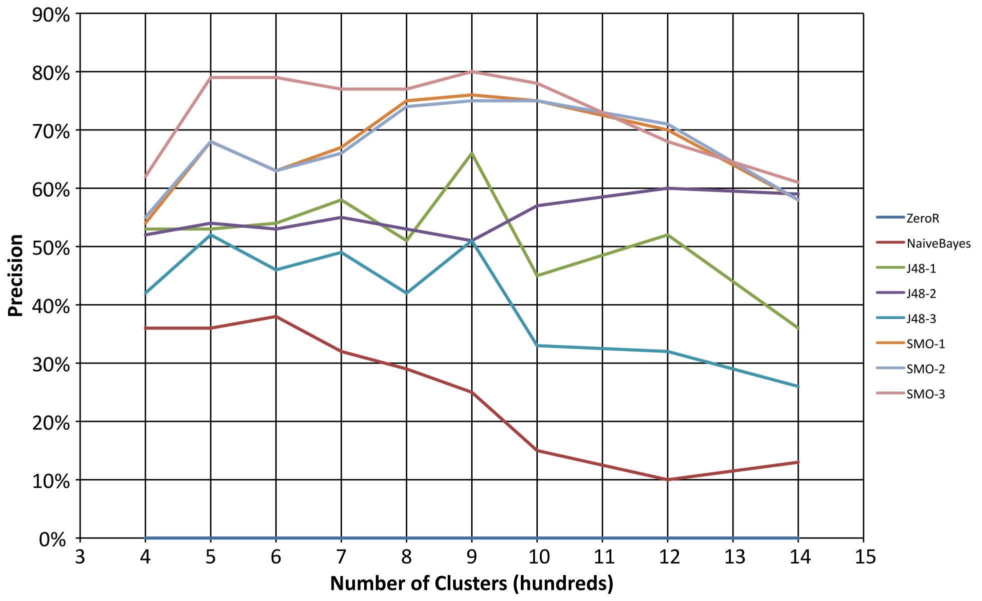


Fig. 8. Experiment results: precision graph.

Table 5
Experiment results: recall.

NC	NB	J48-1	J48-2	J48-3	SMO-1	SMO-2	SMO-3
400	0.35	0.14	0.35	0.11	0.45	0.45	0.36
500	0.41	0.12	0.34	0.11	0.51	0.51	0.39
600	0.37	0.1	0.33	0.1	0.53	0.53	0.44
700	0.29	0.11	0.31	0.09	0.51	0.51	0.39
800	0.25	0.11	0.32	0.08	0.52	0.53	0.36
900	0.19	0.12	0.32	0.09	0.53	0.53	0.31
1000	0.11	0.12	0.37	0.08	0.57	0.56	0.36
1200	0.07	0.14	0.31	0.07	0.49	0.49	0.36
1400	0.1	0.1	0.24	0.05	0.61	0.61	0.51

regarding both precision and recall. The performances we obtained are encouraging, and they suggest that a promising way has been found.

We cannot compare the results shown in this paper for the Abstract Factory DP with other tools which are not able to detect this pattern (for example DPD-Tool [69] and Ptidej [29]), or with tools which fail to recognize it (for example Fujaba [52]) or are not able to detect its position in the code (for example Pinot [60]).

4.3.2. Results on other design patterns

In order to obtain better validation of our approach, in this section we report the results for two other design patterns: the Composite and the Visitor. The recall indicator for the real system has been built considering as the total number of pattern instances the ones found in the documentation, together with validation of all the results given by the tool. For this

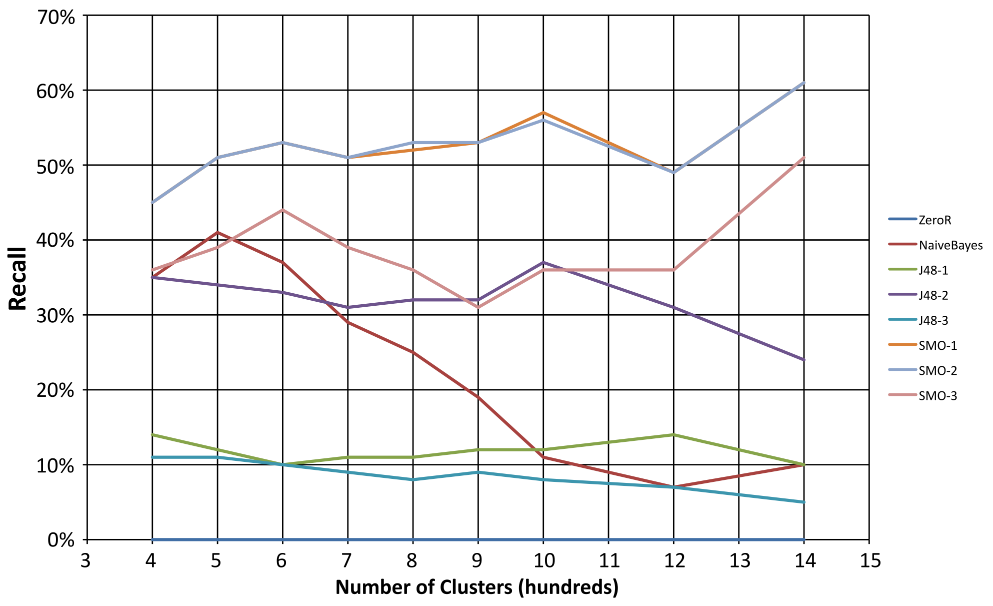


Fig. 9. Experiment results: recall graph.

Table 6
A comparison with web of pattern.

Project	MARPLE		Web of pattern	
	Precision (%)	Recall (%)	Precision (%)	Recall (%)
11 Canonical impls	100	100	100	54
Bexee	67	100	100	50
Doubletype	50	50	0	0
Ehcache	100	100	57	100
Fipaos	75	60	25	40
Groboutils	80	66	62	83

experiment we used the SVM classifier. As in the case of the Abstract Factory, we compare the results obtained through MARPLE and WoP.

Composite. For the analysis of the Composite pattern we decided to analyze five canonical implementations and two real systems in which the DP is certainly present (according to the documentation): the results of the analysis are reported in Table 7. Our tool correctly detects every canonical implementation, but it also has good precision on the real systems with 100% recall in most cases.

As we can observe from Table 7, many instances are not found through WoP, but it performs better than Marple on the infovis project.

Visitor. For the analysis of the Visitor pattern we decided to analyze seven canonical implementations and three real systems, containing pattern instances according to their documentation. The results of the analysis are represented in Table 8.

In Table 8 we can observe, similarly as in the case of the Composite design pattern, that WoP did not find many instances, but it performs better than Marple on a project.

Table 7
Composite design pattern: found instances. TP: True Positive; FP: False Positive.

Project	Known	MARPLE				Web of pattern			
		TP	FP	Prec. (%)	Rec. (%)	TP	FP	Prec. (%)	Rec. (%)
Sun [14]	1	1	0	100	100	0	0	–	0
Kuchana [15]	1	1	0	100	100	0	0	–	0
Vincehuston [16]	1	1	0	100	100	1	0	100	100
Javastaff [17]	1	1	0	100	100	0	0	–	0
Fluffycat [18]	1	1	0	100	100	0	0	–	0
Batik [1]	11	11	2	85	100	0	0	–	0
Infovis [38]	8	6	3	67	75	8	6	57	100

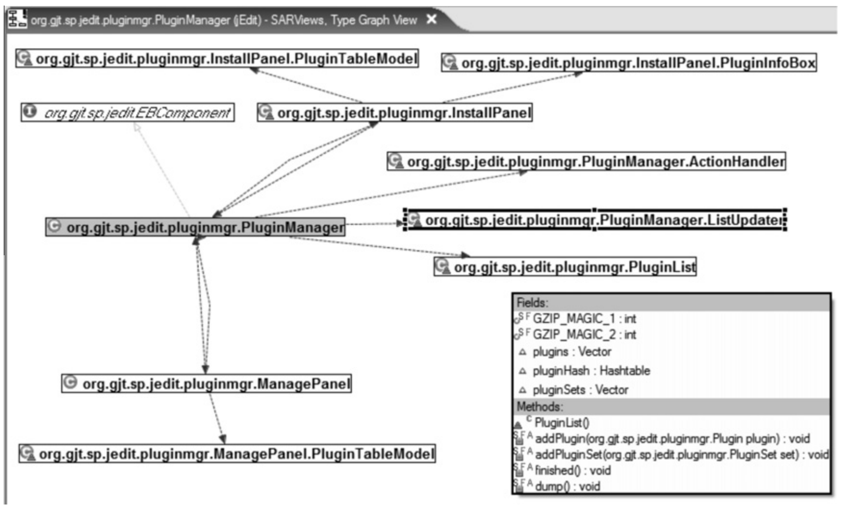
4.4. Results for the SAR module

The SAR module generates diagrams about the architecture of the analyzed systems, using both the information retrieved by the *Metrics Collector* and by the *Basic Element Detector*. Moreover, some views, like the *type graph* depicted in Fig. 10, are generated by analyzing the static relationships among the classes that compose a system. Each *type graph* is focused and centered on a single class or interface, and shows the classes and interfaces it maintains a direct relationship with. The relationships are represented by arrows through the standard UML notation used to represent the different relationship (like inheritance, association, aggregation, containment and so on). This view helps the users focus on single classes rather than on the overall package or system, so that it does not have to cope with the overall system architecture, but only with subsets of classes that are surely easier to manage. Fig. 10 represents the *type graph* for the class org.gjt.sp.jedit.pluginmgr.PluginManager of the JEdit System [40].

Fig. 11 depicts a *class compact* diagram about a package belonging to BCEL [2]. Differently from the *type graph*, this view shows all the classes belonging to a package, in order to have a general overview about the classes composing it and their relationships. The kind of relationship between each couple of classes is derived from the analysis of the elemental design patterns that connect them. For example, association relationships are derived from the existence of a Create Object EDP between the two classes, inheritance and implementation from the presence of the Inheritance EDP and so on. Additional information is provided for critical classes, namely butterfly (classes with many dependents), breakable (classes with many dependencies) and hub classes (that are both butterfly and breakable) both on a local (i.e. within their package) and global (i.e.over the whole system) level. The identification of these critical components is directly derived from the number of relationships each single class has with the rest of the system, and therefore indirectly from the EDPs that connect the various classes, as these relationships are obtained through the analysis of the EDPs themselves. In this example, we can observe that Repository is a global breakable class. This means that its functionalities may and should be subdivided into more classes, in order to avoid the concentration of many functionalities within a single class. Finally, by selecting a class the user can see the set of attributes and methods it declares.

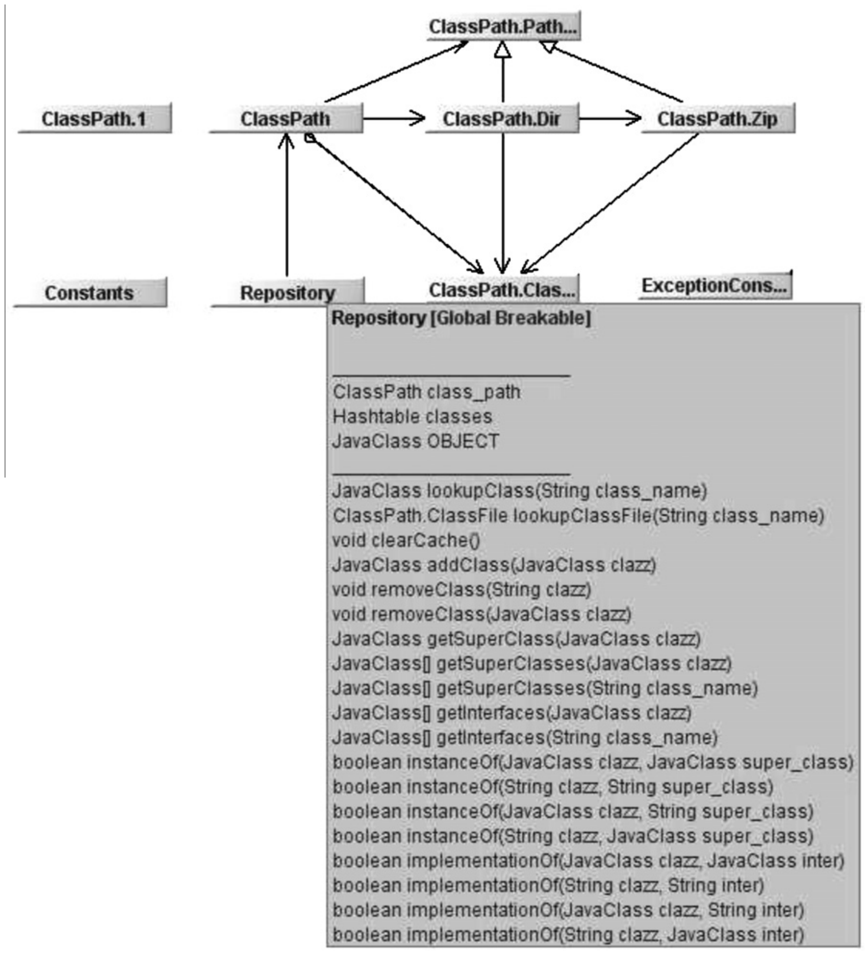
Table 8
Visitor design pattern: found instances. TP: True Positive; FP: False Positive.

Project	Known	MARPLE				Web of pattern			
		TP	FP	Prec. (%)	Rec. (%)	TP	FP	Prec. (%)	Rec. (%)
Sun [72]	1	1	0	100	100	0	0	–	100
Cooper [73]	1	1	0	100	100	0	0	–	100
Fluffycat [74]	1	1	0	100	100	0	0	–	100
Wikipedia [75]	1	1	0	100	100	0	0	–	100
Sm1 [76]	1	1	0	100	100	0	0	–	100
Sm2 [77]	1	1	0	100	100	0	0	–	100
Sm3 [78]	1	1	0	100	100	0	0	–	100
Bcel [2]	5	1	0	100	20	5	0	100	100
Dom4j [23]	1	1	0	100	100	0	0	–	100
Sparta [63]	1	1	1	50	100	0	0	–	100


org.gjt.sp.jedit.pluginmgr.PluginManager (jEdit) - SARViews, Type Graph View
org.gjt.sp.jedit.pluginmgr.InstallPanel.PluginTableModel
org.gjt.sp.jedit.pluginmgr.InstallPanel.PluginInfoBox
org.gjt.sp.jedit.EBComponent
org.gjt.sp.jedit.pluginmgr.InstallPanel
org.gjt.sp.jedit.pluginmgr.PluginManager.ActionHandler
org.gjt.sp.jedit.pluginmgr.PluginManager
org.gjt.sp.jedit.pluginmgr.PluginManager.ListUpdater
org.gjt.sp.jedit.pluginmgr.PluginList
org.gjt.sp.jedit.pluginmgr.ManagePanel
org.gjt.sp.jedit.pluginmgr.ManagePanel.PluginTableModel

```
Fields:
GZIP_MAGIC_1 : int
GZIP_MAGIC_2 : int
plugins : Vector
pluginHash : Hashtable
pluginSets : Vector
Methods:
PluginList()
addPlugin(org.gjt.sp.jedit.pluginmgr.Plugin plugin) : void
addPluginSet(org.gjt.sp.jedit.pluginmgr.PluginSet set) : void
finished() : void
dump() : void
```

Fig. 10. An example of type graph on JEdit's PluginManager class.



```
Repository [Global Breakable]

________________________
ClassPath class_path
Hashtable classes
JavaClass OBJECT
________________________
JavaClass lookupClass(String class_name)
ClassPath.ClassFile lookupClassFile(String class_name)
void clearCache()
JavaClass addClass(JavaClass clazz)
void removeClass(String clazz)
void removeClass(JavaClass clazz)
JavaClass getSuperClass(JavaClass clazz)
JavaClass[] getSuperClasses(JavaClass clazz)
JavaClass[] getSuperClasses(String class_name)
JavaClass[] getInterfaces(JavaClass clazz)
JavaClass[] getInterfaces(String class_name)
boolean instanceOf(JavaClass clazz, JavaClass super_class)
boolean instanceOf(String clazz, String super_class)
boolean instanceOf(JavaClass clazz, String super_class)
boolean instanceOf(String clazz, JavaClass super_class)
boolean implementationOf(JavaClass clazz, JavaClass inter)
boolean implementationOf(String clazz, String inter)
boolean implementationOf(JavaClass clazz, String inter)
boolean implementationOf(String clazz, JavaClass inter)
```

Fig. 11. An example of compact class diagram on BCEL.

Therefore, using the information provided by the basic elements, we can exploit the same kind of information both for DPD and for SAR activities, without further extracting information from the analyzed systems. More examples of the generated views are available at [9].

As mentioned in Section 2.2, many tools for program comprehension and software architecture reconstruction exist, providing different results and views, at different levels of abstraction, each one focusing on specific analysis. For these reasons a comparison among them is a very difficult task. A taxonomy of 34 SAR tools and approaches, considering the kind of input they work on, the analysis techniques they adopt, and the kind of output they produce is described in [24], however no experimentations of the tools considered are reported.

5. Conclusions and future works

In this paper we have presented MARPLE, a tool for design pattern detection and software architecture reconstruction that has been developed as an Eclipse plug-in. We are very interested in using such capabilities also in the context of system modernization and in particular for what concerns systems migration to SOA. In this context we have also explored if detecting design patterns in a system can give useful information for the migration of the system towards a Service Oriented Architecture [8].

Currently, some modules have been completely implemented, and we are currently working on others.

The *Information Detector Engine* has been completely developed as far as the *Basic Elements Detector* is concerned. It detects all of the elemental design patterns, clues and micro pattern that we think are useful as hints for design pattern detection.

As far as the *Joiner* is concerned, we have defined rules for the extraction of DP instances for the Abstract Factory, the Builder, the Factory Method, the Prototype, the Singleton, the Adapter (both based on classes and on objects), the Composite and the Proxy design patterns. Rules for the remaining patterns are under development.

We have developed the *Classifier* module, which proved to us that we can successfully extract information from our representation of the problem, as the performance values are higher than the *a priori* ones. We are currently analyzing if adding or removing some *basic elements*, we can improve the performances.

The views provided by the SAR module have been completely developed, and we are now working on the implementation of further views and on the evaluation of metrics starting from the *Information Detector Engine* output that should be useful for SAR purposes. The evaluation of such metrics is the core task of the *Metrics Collector* module within the *Information Detector Engine*. We are also interested in defining some metrics based on basic elements to be used as indicators to assess the quality of the system's design.

Future works are related to complete the detection of all the design patterns of [27], to add new views based on both metrics and basic elements and to better integrate all the modules: we started the implementation of MARPLE through the development of separated modules, but now we need them to cooperate in order to enhance the user experience and to let the tool to be more effective. Moreover, we are working to enhance the benchmark platform [11] for the comparison of design pattern detection tools, also in order to be able to feed better data to the classifier.

Other future research regards the detection of code smells and the detection of different anti patterns in order to evaluate and improve the quality of design and code. These functionalities are under development, and they exploit the metric computation and the basic element detection modules available in MARPLE. Since the design of MARPLE architecture has been done in order to be language independent, future works will consider other languages, as for example C++.

In this paper we have considered the functionalities provided by MARPLE to support software evolution and reverse engineering. We are currently interested in exploiting our pattern-based approach to unify both software and data reverse engineering [10]. In this perspective we would like to analyze other approaches, such as those proposed in [34] for mining patterns in databases.

Acknowledgements

We would like to thank Prof. Fabio Stella and Prof. Enrico Fagiuoli from the University of Milano Bicocca for the useful discussions and suggestions during our work, in particular about the data mining techniques we used and experimented during this research.

References

[1] Apache Software Foundation, Batik, Web site, 2009. <http://xmlgraphics.apache.org/batik/> version:svn-667266, 1643 CUs.
[2] Apache Software Foundation, Bcel, Web site, 2009. <http://jakarta.apache.org/bcel/> version: cvs-HEAD-20-apr-2009, 335 CUs.
[3] Apache Software Foundation, Xmlbeans, Web site, 2009. <http://xmlbeans.apache.org/>.
[4] F. Arcelli, L. Cristina, Enhancing software evolution through design pattern detection, in: Third International IEEE Workshop on Software Evolvability, IEEE Computer Society, Paris, France, 2007, pp. 7–14.
[5] F. Arcelli, L. Cristina, D. Franzosi, nMARPLE:.NET reverse engineering with marple, in: Proceedings of the ECOOP 2007 Workshop on Object-Oriented Re-engineering (WOOR 2007) 10th Anniversary Edition (2007), Springer, Berlin, Germany, 2007.
[6] F. Arcelli, S. Masiero, C. Raibulet, F. Tisato, A comparison of reverse engineering tools based on design pattern decomposition, in: Proceedings of the 2005 Australian Conference on Software Engineering (ASWEC '05), IEEE Computer Society, Brisbane, Australia, 2005, pp. 262–269.
[7] F. Arcelli, C. Tosi, M. Zanoni, A benchmark proposal for design pattern detection, in: Proceedings of the 2nd Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2008), Co-located with WCRE 2008, <http://moose.unibe.ch/events/famoosr2008>, Antwerp, Belgium, 2008.
[8] F. Arcelli, C. Tosi, M. Zanoni, Can design pattern detection be useful for legacy system migration towards SOA?, in: Proceedings of the 2nd International Workshop on Systems Development in SOA Environments (SDSOA '08), ACM, New York, NY, USA, 2008, pp 63–68.
[9] F. Arcelli, C. Tosi, M. Zanoni, S. Maggioni, Marple, Web site, 2009. <http://essere.disco.unimib.it/reverse/Marple.html>.
[10] F. Arcelli, G. Viscusi, M. Zanoni, Unifying software and data reverse engineering, in: Proceedings of the 5th International Conference on Software and Data Technologies (ICSOFT 2010), Springer-Verlag, Athens, Greece, 2010.
[11] F. Arcelli, M. Zanoni, A benchmark platform for design pattern detection, in: Proceedings of the 2nd International Conference on Pervasive Patterns and Applications (PATTERNS 2010), IEEE Computer Society, Lisbon, Portugal, 2010.
[12] W. Bischofberger, J. Kühl, S. Löffler, Sotograph – a pragmatic approach to source code architecture conformance checking, Software Architecture 3047 (2004) 1–9.
[13] E.J. Chikofsky, Reverse engineering and design recovery: a taxonomy, IEEE Software 7 (1990) 13–17. J.H.C. II.
[14] Composite example, Web site, 2009. <http://www.java2s.com/Code/Java/Design-Pattern/CompositePattern2.htm>.
[15] Composite example, Web site, 2009. <http://www.java2s.com/Code/Java/Design-Pattern/CompositePatternsinJava2.htm>.
[16] Composite example, Web site, 2009. <http://www.vincehuston.org/dp/CompositeDemosJava>.
[17] Composite example, Web site, 2009. <http://www.javastaff.com/article.php?story=20080317161523371>.
[18] Composite example, Web site, 2009. <http://www.FluffyCat.com/Java-Design-Patterns/Composite/>.
[19] C. Cortes, V. Vapnik, Support-vector networks, Machine Learning 20 (1995) 273–297.
[20] Coverity, Inc., Coverity, Web Site, 2009. <http://www.coverity.com/>.
[21] J. Dietrich, C. Elgar, Towards a web of patterns, Web Semantics: Science, Services and Agents on the World Wide Web 5 (2007) 108–116. Software Engineering and the Semantic Web.
[22] J. Dietrich, V. Yakovlev, C. McCartin, G. Jenson, M. Duchrow, Cluster analysis of Java dependency graphs, in: SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization, ACM, Herrsching am Ammersee, Germany, 2008, pp. 91–94.
[23] Dom4j, Web site, 2009. <http://www.dom4j.org/> version: cvs-HEAD-20-apr-2009, 262 CUs.
[24] S. Ducasse, D. Pollet, Software architecture reconstruction: a process-oriented taxonomy, IEEE Transactions on Software Engineering 35 (2009) 573–591.
[25] R. Ferenc, F. Magyar, A. Beszedes, A. Kiss, M. Tarkiainen, Columbus – reverse engineering tool and schema for C++, in: Proceedings of the International Conference on Software Maintenance (ICSM'02), IEEE Computer Society, Montréal, Canada, 2002, p. 172.
[26] L.J. Fülöp, P. Hegedus, R. Ferenc, T. Gyimóthy, Towards a benchmark for evaluating reverse engineering tools, in: WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering, IEEE Computer Society, Antwerp, Belgium, 2008, pp. 335–336.
[27] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1995.

[28] J.Y. Gil, I. Maman, Micro patterns in Java code, in: OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, San Diego, CA, USA, 2005, pp. 97–116.
[29] Y.G. Guéhéneuc, Ptidej: promoting patterns with patterns, in: Proceedings of the 1st ECOOP Workshop on Building a System Using Patterns, Springer-Verlag, Glasgow, UK, 2005.
[30] Y.G. Guéhéneuc, Pmart: Pattern-like micro architecture repository, in: M. Weiss, A. Birukou, P. Giorgini (Eds.), Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories, Irsee Monastery, Springer, Bavaria, Germany, 2007.
[31] Y.G. Guéhéneuc, G. Antoniol, DeMIMA: a multilayered approach for design pattern identification, IEEE Transactions on Software Engineering 34 (2008) 667–684.
[32] Y.G. Guéhéneuc, K. Mens, R. Wuyts, A comparative framework for design recovery tools, in: CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering, IEEE Computer Society, Bari, Italy, 2006, pp. 123–134.
[33] D. van Heesch, Doxygen, Web site, 2009. <http://www.stack.nl/dimitri/doxygen/>.
[34] H.L. Hu, Y.L. Chen, Mining typical patterns from databases, Information Sciences 178 (2008) 3683–3696.
[35] IBM, Sa4j: Structural analysis for Java, Web site, 2004. <http://www.alphaworks.ibm.com/tech/sa4j>.
[36] IBM, Jikes, Web site, 2009. <http://jikes.sourceforge.net>.
[37] IBM, Rational software architect for websphere software, Web Site, 2009. <http://www-01.ibm.com/software/awdtools/swarchitect/websphere/>.
[38] Infovis, Web site, 2009. <http://ivtk.sourceforge.net/> version:0.9beta2, 448 CUs.
[39] java.net, Glassfish, Web site, 2009. <https://glassfish.dev.java.net/>.
[40] JEdit, Web site, 2009. <http://www.jedit.org/>.
[41] O. Kaczor, Y.G. Guéhéneuc, S. Hamel, Identification of design motifs with pattern matching algorithms, Information and Software Technology 52 (2010) 152–168.
[42] R. Kazman, C. Verhoef, W. O'Brien, C. Verhoef, Architecture Reconstruction Guidelines, Third Edition, Technical Report, Software Engineering Institute, Carnegie Mellon University, 2004.
[43] R.K. Keller, R. Schauer, S. Robitaille, P. Pagé, Pattern-based reverse-engineering of design components, in: ICSE '99: Proceedings of the 21st International Conference on Software Engineering, IEEE Computer Society Press, Los Angeles, California, United States, 1999, pp. 226–235.
[44] G. Kniesel, A. Binun, Standing on the shoulders of giants – a data fusion approach to design pattern detection, in: Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC 2009), IEEE Computer Society, Vancouver, Canada, 2009, pp. 208–217.
[45] G. Kniesel, A. Binun, P. Hegedűs, L.J. Fülöp, N. Tsantalis, A. Chatzigeorgiou, Y.G. Guéhéneuc, A common exchange format for design pattern detection tools, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR 2010), IEEE Computer Society, Madrid, Spain, 2010.
[46] M. Lanza, S. Ducasse, Polymetric views-a lightweight visual approach to reverse engineering, IEEE Transactions on Software Engineering 29 (2003) 782–795.
[47] M. Lanza, S. Ducasse, H. Gall, M. Pinzger, Codecrawler: an information visualization tool for program comprehension, in: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, ACM, St. Louis, MO, USA, 2005, pp. 672–673.
[48] Logic Explorers, Codelogic, Web site, 2009. <http://www.logicexplorers.com/index.html>.
[49] S. Maggioni, Design pattern clues for creational design patterns, in: Proceedings of the DPD4RE Workshop, Co-located Event with IEEE WCRE 2006 Conference, IEEE Computer Society, Benevento, Italy, 2006.
[50] J. Malnati, X-ray – An Eclipse Plug-in for Software Visualization, Master's thesis, University of Lugano, 2007. <http://xray.inf.usi.ch/>.
[51] H.A. Mü ller, J.H. Jahnke, D.B. Smith, M.A. Storey, S.R. Tilley, K. Wong, Reverse engineering: a roadmap, in: ICSE '00: Proceedings of the Conference on The Future of Software Engineering, ACM, Limerick, Ireland, 2000, pp. 47–60.
[52] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, J. Welsh, Towards pattern-based design recovery, in: ICSE '02: Proceedings of the 24th International Conference on Software Engineering, ACM, Orlando, Florida, 2002, pp. 338–348.
[53] L. O'Brien, D. Smith, G. Lewis, Supporting migration to services using software architecture reconstruction, in: STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice, IEEE Computer Society, Budapest, Hungary, 2005, pp. 81–91.
[54] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A.I. Verkamo, Software metrics by architectural pattern mining, in: Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), Kluwer, Beijing, China, 2000, pp. 325–332.
[55] L. Prechelt, C. Kramer, Functionality versus practicality: employing existing tools for recovering structural design patterns, Journal of Universal Computer Science 4 (1998) 866–883.
[56] J.R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, 1993.
[57] A. Raza, G. Vogel, E. Plödereder, Bauhaus – a tool suite for program analysis and reverse engineering, in: L. Pinho, M. Gonzlez Harbour (Eds.), 11th International Conference on Reliable Software Technologies - Ada-Europe 2006, Springer, Berlin/Heidelberg, Porto, Portugal, 2006, pp. 71–82.
[58] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM, San Diego, CA, USA, 2005, pp. 167–176.
[59] Scientific Toolworks, Inc., Understand, Web Site, 2009. <http://www.scitools.com/>.
[60] N. Shi, R.A. Olsson, Reverse engineering of design patterns from Java source code, in: ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Tokyo, Japan, 2006, pp. 123–134.
[61] J. Smith, An Elemental Design Pattern Catalog, Technical Report 02-040, Department of Computer Science, University of North Carolina – Chapel Hill, 2002.
[62] J.M. Smith, P.D. Stotts, SPQR: Flexible automated design pattern extraction from source code, in: ASE '03: Proceedings of the 18th IEEE/ACM International Conference on Automated Software Engineering, IEEE Computer Society, Montreal, Canada, 2003, pp. 215–224.
[63] Sparta xml, Web site, 2009. <http://sparta-xml.sourceforge.net/> version: cvs-HEAD-20-apr-2009, 142 CUs.
[64] Sparx Systems Pty Ltd., Enterprise architect, Web Site, 2009. <http://www.sparxsystems.com.au/products/ea/>.
[65] M.A. Storey, C. Best, J. Michand, Shrimp views: an interactive environment for exploring Java programs, in: IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension, IEEE Computer Society, Toronto, Canada, 2001, pp. 111–112.
[66] T. Taibi, Design Pattern Formalization Techniques, IGI Publishing, Hershey, PA, USA, 2007, 2007.
[67] The Eclipse Foundation, Gef, Web site, 2009. <http://www.eclipse.org/gef/>.
[68] C. Tosi, Marple: Classification Techniques Applied to Design Pattern Detection, Master's thesis, Università degli studi di Milano-Bicocca, 2008.
[69] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis, Design pattern detection using similarity scoring, IEEE Transactions on Software Engineering 32 (2006) 896–909.
[70] University of Waikato, Weka, Web site, 2009. <http://www.cs.waikato.ac.nz/ml/weka/>.
[71] University of Waterloo, Software Architecture Group, Swag kit, Web site, 2009. <http://www.swag.uwaterloo.ca/swagkit/>.
[72] Visitor example, Web site, 2009. <http://www.java2s.com/Code/Java/Design-Pattern/VisitorPattern1.htm>.
[73] Visitor example, Web site, 2009. <http://www.java2s.com/Code/Java/Design-Pattern/VisitorpatterninJava.htm>.
[74] Visitor example, Web site, 2009. <http://www.FluffyCat.com/Java-Design-Patterns/Visitor/>.
[75] Visitor example, Web site, 2009. <http://en.wikipedia.org/wiki/Visitor_pattern>.
[76] Visitor example, Web site, 2009. <http://sourcemaking.com/design_patterns/visitor/java/1>.
[77] Visitor example, Web site, 2009. <http://sourcemaking.com/design_patterns/visitor/java/4>.
[78] Visitor example, Web site, 2009. <http://sourcemaking.com/design_patterns/visitor/java/2>.

[79] R. Wettel, Visual exploration of large-scale evolving software, in: Proceedings of the 31st international conference on software engineering – companion volume, 2009, ICSE-Companion 2009, IEEE Computer Society, Vancouver, Canada, 2009, pp. 391–394.
[80] M. Zanoni, MARPLE: Discovering Structured Groups of Classes for Design Pattern Detection, Master's thesis, Università degli studi di Milano-Bicocca, 2008.
[81] H. Zhang, The optimality of naive bayes, in: V. Barr, Z. Markov (Eds.), Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference May 12–14, 2003, AAAI Press, St. Augustine, Florida, USA, 2004.