The MARPLE Project - A Tool for Design Pattern Detection and Software Architecture Reconstruction

Francesca Arcelli, Christian Tosi, Marco Zanoni, Stefano Maggioni

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

Abstract

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, that let the practitioners focus on the overall architecture of the system without minding at 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 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.

Key words: Reverse Engineering, Software Architecture Reconstruction, Design Pattern Detection

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 [4,17]. A relevant

Email addresses: arcelli@disco.unimib.it (Francesca Arcelli), christian.tosi@essere.disco.unimib.it (Christian Tosi), marco.zanoni@essere.disco.unimib.it (Marco Zanoni), maggioni@disco.unimib.it (Stefano Maggioni).

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 obtaining 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 [10]. Finding design patterns in a software system gives hints on the comprehension of a software system and on what kind of problems have been addressed during the development of the system itself. Their presence can be considered as 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.

Both the activities related to design patterns detection (DPD) and software architecture reconstruction (SAR) are particularly relevant in the context of reverse engineering. The main objective of SAR is to abstract from the analyzed system's details in order to obtain general views, diagrams and evaluations on it. The extraction of such data helps the engineers in having a global understanding of the system and of its architecture.

Different tools for DPD have been proposed in the literature (e.g. [12,7,18,23,25]). They usually have problems in finding all the design patterns of the GoF catalogue [10], some tools recognize only a small subset of these patterns, but the main problem is that the found results contain many false positive DP instances and moreover they usually don't scale well when trying to analyze medium/large systems. Also for SAR different tools have been proposed (e.g. Codecrawler [15], Doxygen [27], SA4J [13], Codelogic [6],ARMIN [19] and Swagkit [21]), obtaining different views at different levels of abstraction, some exploiting only static analysis and other exploiting both static and dynamic analysis. Usually tools for SAR don't perform DPD.

The aim of this paper is to describe a project, on which we are currently working, named MARPLE (Metrics and Architecture Reconstruction PLug-in for Eclipse) that aim 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 ([2,1]), which can be considered indicators of the presence of patterns. We use static source code analysis: the ASTs of the analyzed projects are parsed in order to obtain the structures we need for our elaboration, which we called *basic elements* (BE).

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 be useful also for the DPD process. In fact we can detect a design pattern through the DPD module and then check the result through some views offered by the SAR module. For this reason we found interesting to start developing a tool able to do both DPD and SAR.

MARPLE is conceived as an Eclipse plug-in. This choice was supported by 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 the 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 speed up the development process.

We aim in this paper to describe the overall architecture of MARPLE, but not to go in details on how each module has been developed (also for space reasons). Starting from the different problems and results obtained experimenting on the same systems many other DPD tools, we decided to develop a new tool for DPD which exploits different classification techniques for DPD (briefly described in Section 2).

In this paper we focus our attention on describing the results obtained through MARPLE on DPD and in particular on the detection of the Abstract Factory DP in order to show the full detection process of a design pattern, and the results obtained on it.

The paper is organized as follows: Section 2 introduces the overall MARPLE architecture and goes into more details about the technologies, plug-ins and libraries used during its development; Section 3 presents some results about the already implemented modules of design pattern detection; Section 4 briefly discusses about the opportunity of migrating MARPLE to a distributed environment, and finally Section 5 gathers the conclusions and outlines possible future works.

2. An overview on MARPLE

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

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

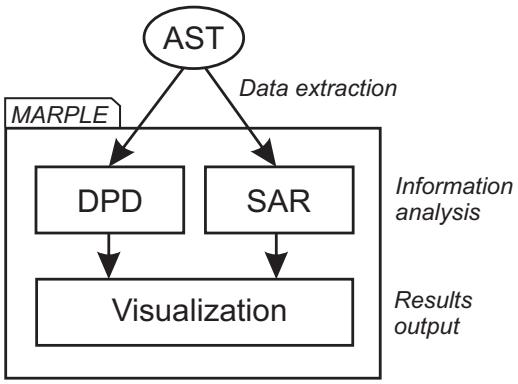

Fig. 1. An overview to the general process.

DPD and SAR receive both 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) [24], design pattern clues [16] and micro patterns [11], that are intended as the basic information we exploit as hints for the

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

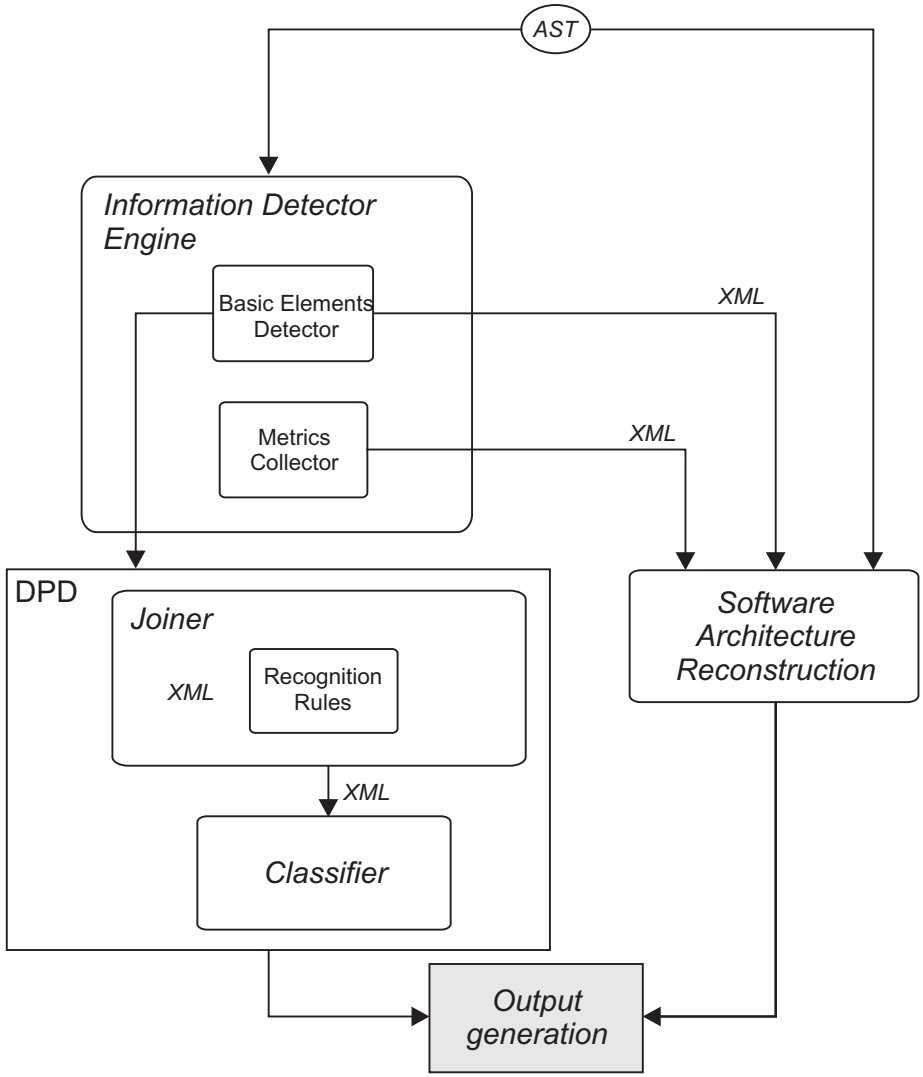

Fig. 2. The architecture of MARPLE.

The architecture is constituted by 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 architectures from the project that could match those of design patterns, basing 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 the canonical design patterns 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.

Each of these modules takes part to a different stage of computation (see Figure 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 stage of computation both for DPD and SAR, and the type of information it is produced as output from these two activities.

	Data extraction	Information analysis	Output generation
DPD	Information Detector Engine	Joiner, Classifier	XML, design pattern diagrams
SAR	Information Detector Engine, AST	SAR module	Abstracted views,metrics

Table 1
The modules involved in the various stages.

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 Figure 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, with the Apache XMLBeans library [8]. As this kind of data access is common for every module, we won't discuss it any further.

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.

2.1. *The Information Detector Engine Module*

Currently, the *Basic Elements Detector* (BED) sub-module has been completely developed. 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 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 detected without any problem.

We didn't implemented an AST structure from scratch, but we used the *org.eclipse.jdt.core.dom* library that provides all the classes and interfaces that can be used to access a project's ASTs. Moreover, it provides the class *ASTVisitor*, that is used to visit the nodes constituting the AST according to the Visitor design pattern [10]. Therefore, one visitor for each *basic element* has been extended from *ASTVisitor*. These visitors are invoked sequentially on the ASTs of the classes constituting the project and visit only those nodes that may contain the information they are able to detect (for example, the visitors that look for *method call* EDPs only analyze nodes that represent a method invocation, i.e. instances of the *MethodInvocation* class).

The results coming from the visitors, i.e. the instances of *basic elements* that have been found inside the project, are then stored in an XML file.

The BED module has been developed also for the .NET enviroment.

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.

2.2. *The Joiner Module*

As far as the *Joiner* module is concerned, no particular third party technologies have been used. Nonetheless, this module doesn't handle the system through its AST representation, but it manages it as a graph $G = (V, E)$, where the set of vertex V corresponds to the set of types (i.e. classes and interfaces) the project is constituted by, while E is 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). The system graph representation is directly derived from the output generated by the *Information Detector Engine*. As we have briefly anticipated at the begin of this section, 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 Figure 3.

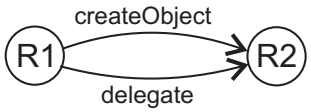

Fig. 3. An example of Joiner rule.

The *Joiner* module tries to match and extract this kind of architectures 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 [28].

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

2.3. *The Classifier Module*

The *Joiner* output is the input for the next analysis step performed by the *Classifier* module. This module takes all the candidate design pattern instances and tries to evaluate their grade of similarity to the searched design pattern in order to be able to rank the instances given as output. Figure 4 shows an example of the classification process.

Through our current approach, 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 i-th 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 [20]. All the detail of classification process can be found in [26].

2.4. *The Software Architecture Reconstruction Module*

One of the objectives of MARPLE is supporting the user with the visualization of abstractions about the analyzed systems. Currently, the SAR module generates six kinds of views on a system (see `http://essere.disco.unimib.it/reverse/Marple.html` for examples of the generated views):

- 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, similar to the one provided by CodeCrawler [15];
- A *type graph* of the entities constituting the project, similar to the *class graph* of Doxygen [27]; With the term *type* we mean both classes and interfaces, according to the Eclipse JDT API specifications;

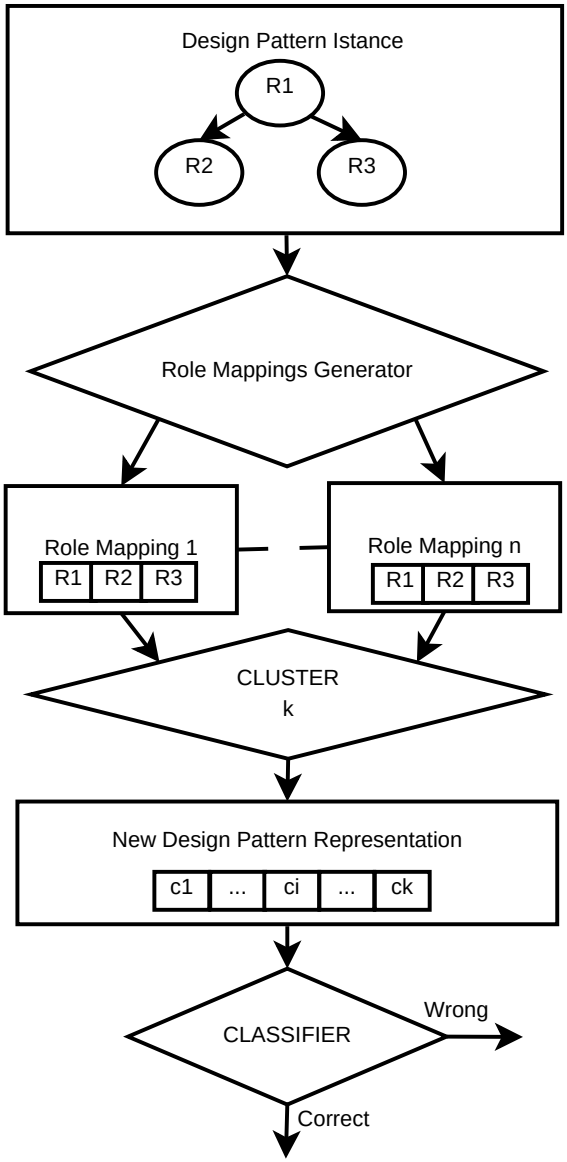

Fig. 4. Classification process.

– 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*. In this way, we exploit a common source of information for both DPD and SAR. More specifically, the SAR functionalities related to the generation of the *class compact* and the *class extended* diagrams are achieved only through the analysis of the *elemental design pattterns* 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 that 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. On the other hand, the remaining views exploit also 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 lines of code (WLOC); the *type graph* is built by analyzing the inheritance, implementation, association and containment relationships of each class with the rest of the system; finally, the *class blueprint* view reports for each method the number of invoked methods (NI) and its 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 [9]. GEF allowed us to take advantage of rich representation mechanisms and of the MVC (Model-View-Controller) pattern, that allowed us to maintain the underlying model (i.e. the ASTs of the analyzed project) separate from the implemented view.

3. Experimental results for DPD

As MARPLE is currently a project under development, we are not able to provide the complete results gained by exploiting every module of the project.

However, 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, where 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 the academic worlds.We described in this paper the results obtained for the detection of the Abstract Facotry DP and hence we show the dataset tailored for the detection of this pattern.

The projects were selected searching for canonical implementations of the Abstract Factory pattern and 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.

Project	Number of CU	Project	Number of CU
AF-earthlink	1	fdsapi	95
AF-java.net	5	sunxacml	155
jboot	6	doubletype	175
AF-WikiPedia	7	bexee	191
AF-rice	8	wasa	213
AF-vico.org	10	wfmopen	241
AF-itec	10	GroboUtils	285
JVending-Registry-CDC	11	Nodal	300
AF-c-sharpcorner	11	Rambutan	328
AF-fluffycat	12	sparql	332
AF-apwebco	13	infovis	448
AF-javapractises	13	fipaos	459
AF-Gamma	21	mandarax	514
dynamicdispatcher	23	JasperReports	798
ehcache	86	Batik	1643

Table 2
Projects in the dataset, with respective number of compilation units (CU)

3.1. *Results for the Information Detector Engine Module*

The *Information Detector Engine* module is composed by 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 has the aim to calculate metrics on the code that will be useful for SAR purposes.

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 we report in Table 3 some performance and size indicators for each system.

Project Name	Num of CU	Elapsed Time (s)	Memory Consumption (Mb)	Num of BEs
AF-earthlink	1	5	2	54
AF-java.net	5	4	9	30
jboot	6	5	1	88
AF-WikiPedia	7	5	7	36
AF-rice	8	5	5	81
AF-vico.org	10	5	5	44
AF-itec	10	5	5	73
JVending-Registry-CDC	11	6	3	174
AF-c-sharpcorner	11	6	8	59
AF-fluffycat	12	5	4	104
AF-apwebco	13	6	5	69
AF-javapractises	13	5	6	94
AF-Gamma	21	6	8	43
dynamicdispatcher	23	6	17	528
ehcache	86	12	83	4820
fdsapi	95	10	46	4325
sunxacml	155	10	73	4702
doubletype	175	34	190	24201
bexee	191	16	115	3544
wasa	213	10	86	5597
wstx	241	15	130	16541
GroboUtils	285	15	89	10648
Nodal	300	19	163	18116
Rambutan	328	15	124	6946
sparql	332	16	136	11919
infovis	448	31	299	25929
fipaos	459	28	278	24927
mandarax	514	26	251	16905
JasperReports	798	58	671	49082
Batik	1643	143	1725	93714

Table 3
Basic Element Detector performance and size results

We can easily observe from Table 3 that Memory Consumption and Elapsed Time are

approximately in a linear relation to the number of Compilation Units (see Figure 5): this means that the detector scales linearly in the size of the analyzed system.

Fig. 5. Relationship between Memory Consumption, Elapsed Time and Number of Compilation Units.

As the main objective of the *Basic Elements Detector* is the extraction of information to be used in the further stages of computation for both DPD and SAR, discussing in detail about the results provided by this module "as they are" is not really meaningful.

3.2. *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 on our dataset, with the *Joiner* module. We report in Table 4 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, therefore the Joiner module has, on this dataset, a precision of about 20%. This is not an high precision value, but it is the result of 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 an anormal project (Nodal) that produces an high number of wrong instances and three projects that contain only wrong instances, producing a quite unbalanced dataset. All the results have been also manually evaluated.

Project	Num of BEs	Num of Classes	Found Instances	Valid Instances	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

Table 4
Joiner performance and size results

We proved, as previously written, 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 Figure 6.

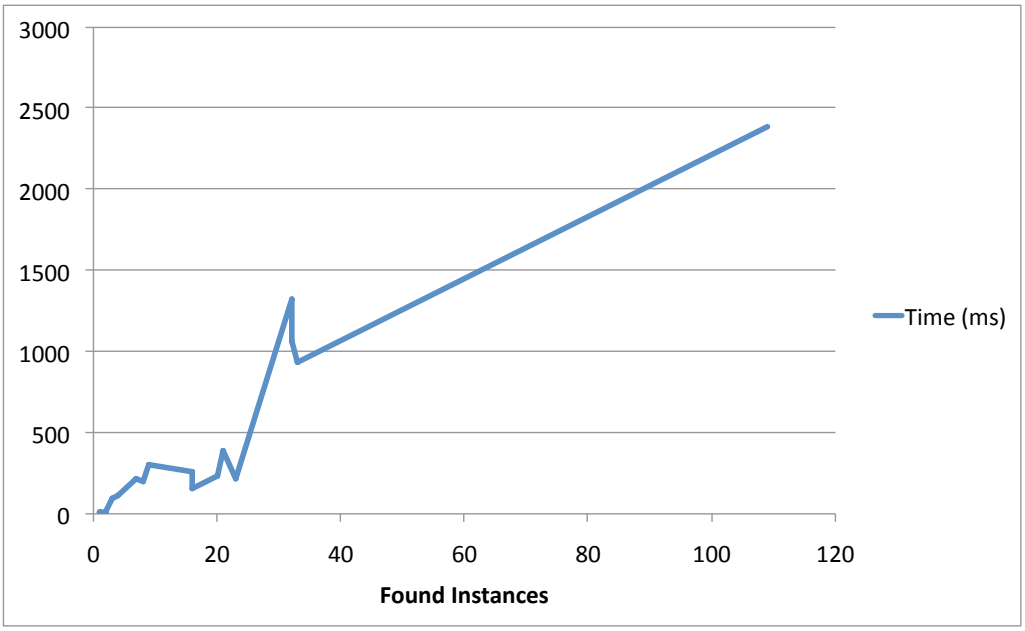

Fig. 6. Relationship between Time and Number of Found Instances.

3.3. *Results for the Classifier module*

The classifier module therefore works on a dataset composed by about 20% of instances having positive class and about 80% of instances having negative class.

In our experimentations we tried the Simple K-means clustering algorithm with many different numbers of clusters K (NC) and a bunch of classifiers chosen through data exploration using weka [20]; the results of some experiments are shown in Tables 6, 5 and in Figures 8, 7. All experiments are realized using Repeated Cross Validation with 10 folds and 10 repetitions.

The classifiers we tested are:

ZeroR: a classifier that predicts the mean (for a numeric class) or the mode (for a nominal class);
NB: Naïve Bayes classifier [29];
J48: C4.5 classifier [22];
SMO: Support Vector Machine classifier [5];

NC	ZeroR	NB	J48-1	J48-2	J48-3	SMO-1	SMO-2	SMO-3
400	0	0,36	0,53	0,52	0,42	0,54	0,55	0,62
500	0	0,36	0,53	0,54	0,52	0,68	0,68	0,79
600	0	0,38	0,54	0,53	0,46	0,63	0,63	0,79
700	0	0,32	0,58	0,55	0,49	0,67	0,66	0,77
800	0	0,29	0,51	0,53	0,42	0,75	0,74	0,77
900	0	0,25	0,66	0,51	0,51	0,76	0,75	0,8
1000	0	0,15	0,45	0,57	0,33	0,75	0,75	0,78
1200	0	0,1	0,52	0,6	0,32	0,7	0,71	0,68
1400	0	0,13	0,36	0,59	0,26	0,58	0,58	0,61

Table 5
Experiments precision

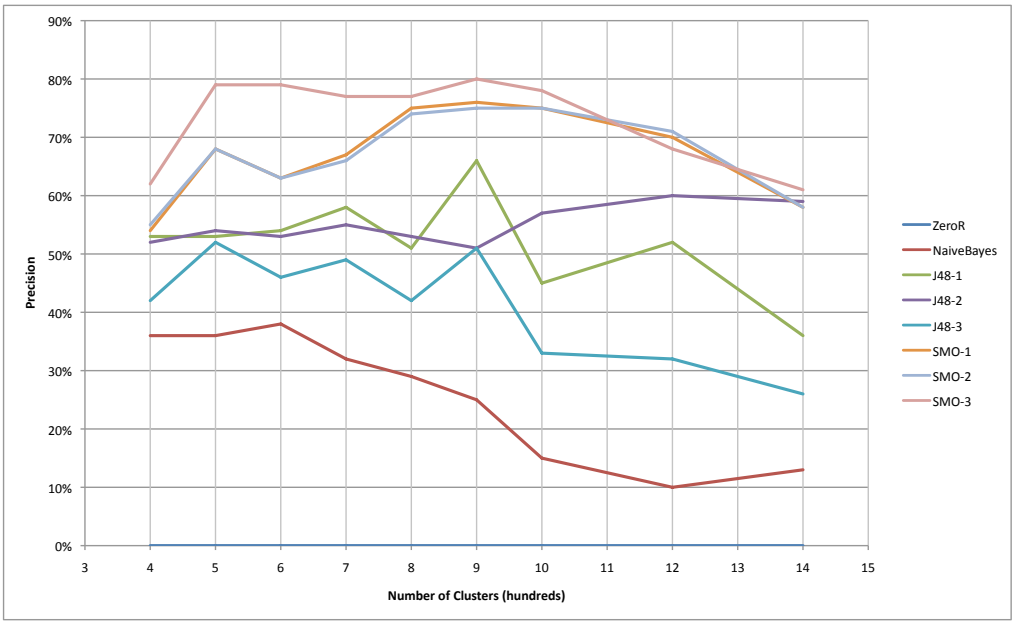

Fig. 7. Experiments precision graph

The first indicator we show is the precision (see Table 5). The best classifier is the SMO-3 but SMO-2 gave a good performance too (see Figure 7); the worst, excluding ZeroR, is the naïve bayes. The ZeroR classifier's recall is 0 because it classifies using the majority class, so it assigns all the instances to the negative class.

NC	ZeroR	NB	J48-1	J48-2	J48-3	SMO-1	SMO-2	SMO-3
400	0	0,35	0,14	0,35	0,11	0,45	0,45	0,36
500	0	0,41	0,12	0,34	0,11	0,51	0,51	0,39
600	0	0,37	0,1	0,33	0,1	0,53	0,53	0,44
700	0	0,29	0,11	0,31	0,09	0,51	0,51	0,39
800	0	0,25	0,11	0,32	0,08	0,52	0,53	0,36
900	0	0,19	0,12	0,32	0,09	0,53	0,53	0,31
1000	0	0,11	0,12	0,37	0,08	0,57	0,56	0,36
1200	0	0,07	0,14	0,31	0,07	0,49	0,49	0,36
1400	0	0,1	0,1	0,24	0,05	0,61	0,61	0,51

Table 6
Experiments recall

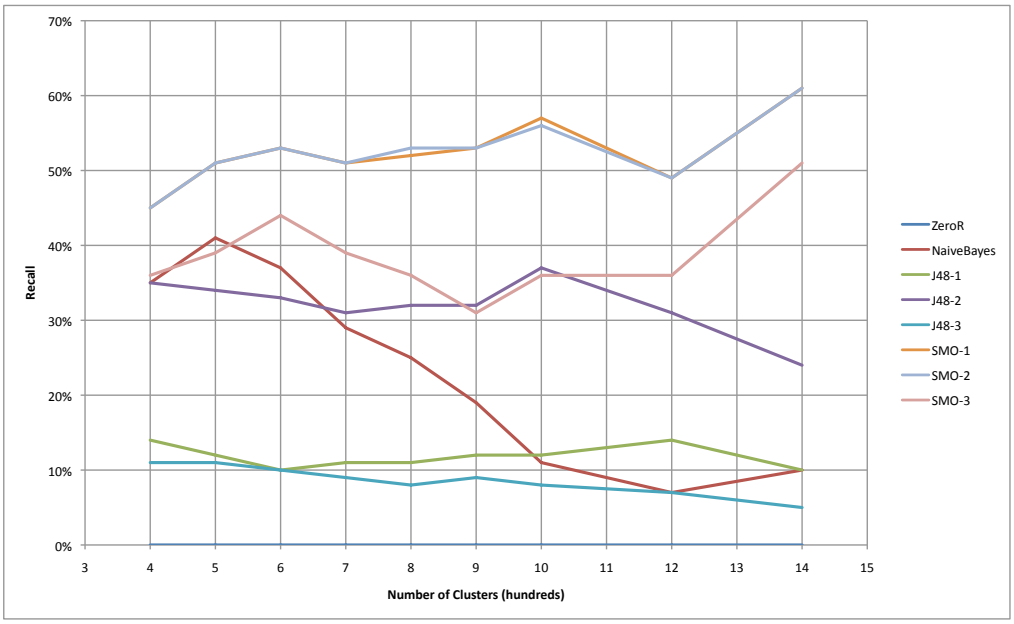

Fig. 8. Experiments recall graph

The second indicator we show is the recall (see Table 6). The results evidence that the best classifier is SMO-2 (see Figure 8); 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 maximum numbers of clusters.

4. Distributed MARPLE

There is a parallel project that allows us to use MARPLE in a client-server environment (Figure 9).

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 1700Mb 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's well known that some classification algorithms require a lot of time in order to classify their dataset.

For all of this reasons we decided to implement this version in order to allow users to run their analysis through the internet on the elaboration server and to check asynchronously the results of the elaboration.

This project is based on the J2EE v.5 platform and precisely on the Glassfish [14] application server.

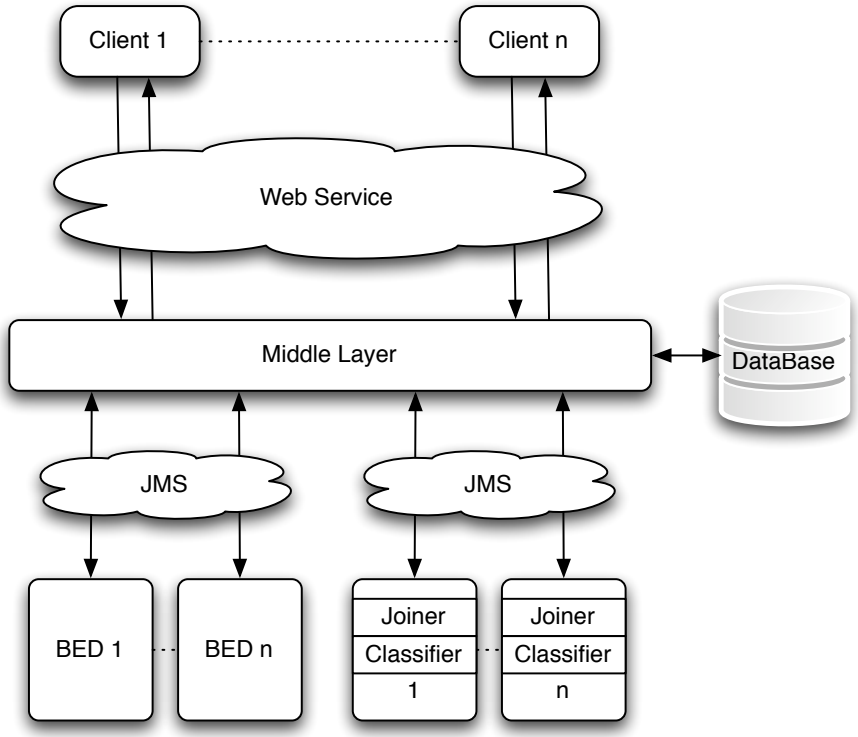

Fig. 9. The architecture of distributed MARPLE.

The system architecture, as you can see in Figure 9, is composed by 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 analysis.

Server: this module is developed on a J2EE Application Server and it implements the middle layer of the system. It implements also the Web Service in order to receive users requests, manage the Persistence Backend (Database) that contains all the information of the users, and finally start 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 run, the Joiner-Classifier node must possess all the detected Basic Elements.

BED Node: this module receives from the Server, through JMS, a set of classes to analyze and the entire project source code; than 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 runs sequentially the Joiner and the Classifier Module and then it returns to the Server all the found pattern instances with their classification values.

5. Conclusions and future works

In this paper we have presented MARPLE, a tool for design pattern detection and software architecture reconstruction that is being 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 explored if detecting design patterns in a system can give useful information towards SOA migration [3].

Currently, some modules have been completely implemented, others only partially.

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 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 candidates 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 have to be defined and tests have to be performed on them.

We have developed the *Classifier* module, which proved that we can extract information from our representation of the problem, as the performance values are higher than the apriori ones. We are currently analysing if, adding or removing some *basic elements*, we could improve the performance.

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 to define some metrics based on the basic elements to be used as indicators of the quality design assesment.

Future works are related to complete the detection of all the Design Patterns of [10], 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 would like to define a benchmark for the comparison of design pattern detection tools.

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++.

From our experience, the Eclipse framework demonstrated to be really useful and flexible for the development of a project like MARPLE. Many features are available directly from the framework (like the AST representations we have used for information extraction, code inspection, etc) or through ad-hoc plug-ins (like the GEF framework used for some of the views we have implemented). While the major efforts requested are related to the extension of such functionalities and to the understanding of the rationale and issues behind each component and plug-in we have used and the major problems we had to face are related to the AST memory consumption.

Therefore, dealing with the design and development of MARPLE, we achieved not only confidence with DPD and SAR techniques, but also with those instruments that constitute the grounds of the MARPLE project itself, namely the Eclipse framework and all the components and libraries we have cited along this paper.

References

[1] F. Arcelli, S. Masiero, C. Raibulet, and F. Tisato. A comparison of reverse engineering tools based on design pattern decomposition. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 262–269, Washington, DC, USA, 2005. IEEE Computer Society.

[2] F. Arcelli and C. Raibulet. The role of design pattern decomposition in reverse engineering tools. In *Pre-Proceedings of the IEEE International Workshop on Software Technology and Engineering Practice (STEP 2005)*, pages 230–233, Budapest, Hungary, 2005.

[3] F. Arcelli, C. Tosi, and M. Zanoni. Can design pattern detection be useful for legacy systemmigration towards soa? In *SDSOA '08: IEEE Proceedings of the 2nd international ICSE workshop on Systems development in SOA environments*, pages 63–68, New York, NY, USA, 2008. ACM.

[4] E. J. Chikofsky and J. H. C. II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[5] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[6] L. Explorers. Codelogic. Web site. `http://www.logicexplorers.com/index.html`.

[7] R. Ferenc, F. Magyar, A. Beszedes, A. Kiss, and M. Tarkiainen. Columbus - reverse engineering tool and schema for c++. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 172, Washington, DC, USA, 2002. IEEE Computer Society.

[8] T. A. S. Foundation. Xmlbeans. Web site. `http://xmlbeans.apache.org/`.

[9] T. E. Foundation. Gef. Web site. `http://www.eclipse.org/gef/`.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[11] J. Y. Gil and I. Maman. Micro patterns in java code. *SIGPLAN Not.*, 40(10):97–116, 2005.

[12] Y.-G. Guéhéneuc. Ptidej: Promoting patterns with patterns. In *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*. Springer Verlag, 2005.

[13] IBM. Sa4j: Structural analysis for java. Web site, 2004. `http://www.alphaworks.ibm.com/tech/sa4j`.

[14] java.net. Glassfish. Web site. `https://glassfish.dev.java.net/`.

[15] M. Lanza. *Object oriented Reverse Engineering.* PhD thesis, University of Berna, 2003.

[16] S. Maggioni. Design pattern clues for creational design patterns. In *In Proeedings, of the DPD4RE Workshop, co-located event with IEEE WCRE 2006 Conference*, Benevento, Italy, 2006.

[17] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong. Reverse engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 47–60, New York, NY, USA, 2000. ACM.

[18] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM.

[19] L. O'Brien, D. Smith, and 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*, pages 81–91, Washington, DC, USA, 2005. IEEE Computer Society.

[20] U. of Waikato. Weka. Web site. `http://www.cs.waikato.ac.nz/ml/weka/`.

[21] S. U. of Waterloo. Swag kit. Web site. `http://www.swag.uwaterloo.ca/swagkit/`.

[22] J. R. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[23] N. Shi and R. A. Olsson. Reverse engineering of design patterns for high performance computing. In *Proceedings of the 2005 Workshop on Patterns in High Performance Computing*, 2005.

[24] J. Smith. An elemental design pattern catalog. Technical Report 02-040, Dept. of Computer Science, Univ. of North Carolina - Chapel Hill, December 2002.

[25] J. M. Smith and P. D. Stotts. Spqr: Flexible automated design pattern extraction from source code. In *ASE '03: Proceedings of the 18st IEEE/ACM International Conference on Automated Software Engineering*, pages 215–224, 2003.

[26] C. Tosi. Marple: classification techniques applied to design pattern detection. Master's thesis, Università degli studi di Milano-Bicocca, 2008.

[27] D. van Heesch. Doxygen. Web site. `http://www.stack.nl/~dimitri/doxygen/`.

[28] M. Zanoni. Marple: discovering structured groups of classes for design pattern detection. Master's thesis, Università degli studi di Milano-Bicocca, 2008.

[29] H. Zhang. The optimality of naive bayes. In V. Barr and Z. Markov, editors, *Proceedings of the Sixteenth International Florida Artificial Intelligence Research Society Conference, May 12-14, 2003, St. Augustine, Florida, USA*. AAAI Press, 2004.