DPB: A Benchmark for Design Pattern Detection Tools

Francesca Arcelli Fontana, Andrea Caracciolo, Marco Zanoni
Dipartimento di Informatica, Sistemistica e Comunicazione
University of Milano Bicocca
Milano, Italy
Email: {arcelli,marco.zanoni}@disco.unimib.it, a.caracciolo1@campus.unimib.it

***Abstract*—Many activities can be done to support software evolution and reverse engineering of a system. Design pattern detection is one of these activities. It is useful to gain knowledge on the design issues of an existing system, on its architecture and design quality, improving the comprehension of the system and hence its maintainability and evolution. Several tools for design pattern detection have been developed in the literature, but they usually provide different results when analyzing the same systems. Some works have been proposed in the literature to compare these results, but a standard and widely-accepted benchmark is not yet available. In this work we propose our benchmark web application for design pattern detection, based on a community driven evaluation.**

***Keywords*-design pattern detection; benchmark; meta-model.**

I. Introduction

Software architecture reconstruction and design pattern detection are important and useful for reverse engineering. Design pattern detection (DPD) is a topic that received a great interest during the past years. Finding design pattern (DP) [1] instances in a software system can give useful hints for the comprehension of a software system and on what kind of problems have been addressed during the development of the system itself. Moreover, they are important during the re-documentation process, in particular when the documentation is poor, incomplete, or not up-to-date.

Several DPD approaches and tools have been developed, exploiting different techniques for the detection such as fuzzy logic, constraints solving techniques, theorem provers, template matching methods, and classification techniques [2]–[10]. In spite of the many approaches proposed, the results obtained are often quite unsatisfactory and different from one tool to another.

Many tools find many pattern candidates which are false positives but other correct ones are not found. One common problem in DPD is the so called variant problem: DPs can be implemented in several ways, often different from one another. The main variants of each pattern are described in the catalog of Gamma et al. [1]. Other patterns are applied when the context of the application requires them. These variants cause the failure of most pattern instance recognition tools using rigid detection approaches, which are based only on canonical pattern definitions.

Hence the comparison of the results provided by the different tools is important, to be able to evaluate the best approach and tools. Some work regarding the comparison of DPD tools is described in the next section.

We analyzed and faced the problem related to the different results provided by the DPD tools, because we are developing MARPLE (Metrics and Architecture Reconstruction PLug-in for Eclipse) [11] whose main aims are related to DPD and software architecture reconstruction. The development of MARPLE brought to the definition of a meta-model able to represent DPD results, and we used that model as a base for the work described in this paper.

The aim of this work is to introduce and describe a benchmark web application to be used by software engineers to compare DPD tools. We propose to the DPD community some mechanisms to obtain agreed results and to make them easily available. Our approach is characterized by:

- a general meta-model for design pattern representation;
- a way of comparing results coming from different tools;
- the possibility to evaluate instances and discuss about their correctness.

The adoption by the DPD community of a benchmark can improve the cooperation among the researchers and the reuse of tools written by others instead of the development of new ones.

The paper is organized through the following Sections: in Section II we describe the principal related works, in Section III we introduce the main functionalities of our application, in Section IV the meta-model used to represent DP definitions and instances, in Section V we introduce the algorithm developed for comparing a given pair of DP instances. Finally in Section VI we conclude by outlining some future developments of our research.

II. Related Work

Undertaking a comparison among design pattern detection tools is a difficult task. Few benchmark proposals for the evaluation of design pattern detection tools have been presented in the literature. We already described our first proposal for a benchmark in previous work [12], and we extend and finalize it in this paper. The only other known effort on the same topic is called DEEBEE (DEsign pattern

Evaluation BEnchmark Environment) [13], a web application for evaluating and comparing design pattern detection tools. This application features a rich set of useful functionalities, but lacks usability, offers poor support to several key tasks and pays little attention to content organization and information presentation. As a result, it feels hard to find and compare data, provide a solid and justified evaluation and get an overall picture of the emerging benchmark results. Moreover, the meta-model used to describe pattern instances appears to be weak and not formalized. Our solution is able to correct most of the above highlighted shortcomings.

P-MARt [14] is a repository of pattern-like micro-architectures, which has the purpose to serve as baseline to assess the precision and recall of pattern identification tools. The repository contains the analysis of specific versions of different open source programs. We added the contents of P-MARt into our benchmark application in order to provide an interactive exploration of those data.

The availability and simple interchange of DPD results could be helpful also to support DPD techniques like the one proposed by Kniesel and Binum [15]: they compared different design patterns detection tools and proposed a novel approach based on data fusion, built on the synergy of proven techniques, without requiring any re-implementation of what is already available.

Another common exchange format for DPD tools has been proposed, called DPDX [16], with the aim to overcome the limitations coming from the existence of different output formats. DPDX provides the basis for an open federation of tools that perform comparison, fusion, visualization, and-or validation of DPD results. The main differences between DPDX and the model used in our application are described in Section IV.

In our work, we aim to provide an online application to support the comparison of DPD results, with the benefit of having a flexible underlying model and a collaborative environment.

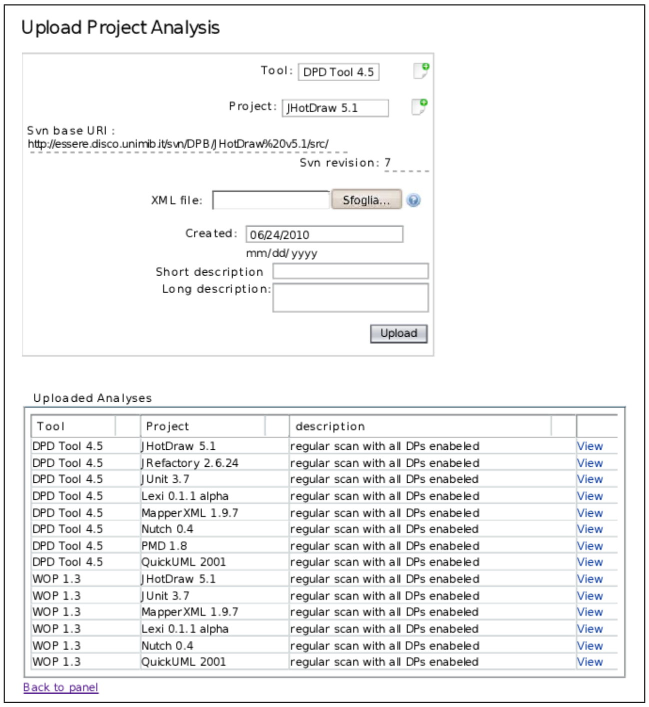
III. Web application description

The web application is available online [17] and is subdivided in:

- a *search* section, which lets the user find the results of a particular analysis, according to different parameters;
- a *compare* section, which allows to compare the instances found by different tools on the same input project;
- a *browse* section, which provides a tree-like view of the contents of the application.
- a *documentation* section, containing some references and guides to use the application; in addition the home page briefly introduces the system functionalities and provides a step by step tutorial;

Through these different kinds of sections, the user can obtain a detailed view of each pattern instance loaded on the application. Each instance is described by a set of five different informative *views* and can be rated through a functionality which allows the user to express an *evaluation* of the instance.

All the above functionalities are visible to all users; if a user wants to load a new analysis into the application, he must be registered and upload the XML file containing the instances, as shown in Figure 1. In our web application, an *analysis* consists of the combination of a set of instances, a description, the name of the used DPD tool and the analyzed project.



Upload Project Analysis

Tool: DPD Tool 4.5

Project: JHotDraw 5.1

Svn base URI :
http://essere.disco.unimib.it/svn/DPB/JHotDraw%20v5.1/src/

Svn revision: 7

XML file:

Created: 06/24/2010
mm/dd/yyyy

Short description

Long description:

Uploaded Analyses

Tool	Project	description	
DPD Tool 4.5	JHotDraw 5.1	regular scan with all DPs enabeled	View
DPD Tool 4.5	JRefactory 2.6.24	regular scan with all DPs enabeled	View
DPD Tool 4.5	JUnit 3.7	regular scan with all DPs enabeled	View
DPD Tool 4.5	Lexi 0.1.1 alpha	regular scan with all DPs enabeled	View
DPD Tool 4.5	MapperXML 1.9.7	regular scan with all DPs enabeled	View
DPD Tool 4.5	Nutch 0.4	regular scan with all DPs enabeled	View
DPD Tool 4.5	PMD 1.8	regular scan with all DPs enabeled	View
DPD Tool 4.5	QuickUML 2001	regular scan with all DPs enabeled	View
WOP 1.3	JHotDraw 5.1	regular scan with all DPs enabeled	View
WOP 1.3	JUnit 3.7	regular scan with all DPs enabeled	View
WOP 1.3	MapperXML 1.9.7	regular scan with all DPs enabeled	View
WOP 1.3	Lexi 0.1.1 alpha	regular scan with all DPs enabeled	View
WOP 1.3	Nutch 0.4	regular scan with all DPs enabeled	View
WOP 1.3	QuickUML 2001	regular scan with all DPs enabeled	View

Figure 1. Example of analysis loading

A. Views

Each DP instance can be viewed through three graphical representations:

- as a UML class diagram (view number 1 in Figure 2), where all the classes referenced in the instance are displayed as inter-connected entities labeled with the name of their respective role;
- as a dynamic tree graph (view number 2 in Figure 2), which provides a graphical tree-structure representation of the instance, where the nodes represent the elements contained in the associated instance model (see section IV);
- as a nested boxes diagram (view number 3 in Figure 2), where classes are grouped by role into hierarchically ordered boxes.

The first view in Figure 2 is generated by a tool (developed in the context of this project) which is able to build a class diagram containing the elements being part of the chosen DP instance. This task is accomplished by analyzing the source code of the project in which the instance was found. This tool currently supports any kind of entity (class, method, field) allowed by the meta-model (see section IV) and is able to recognize a subset of the standard UML relationships (generalization, realization, dependency) as well as the following extra relationships: "declares", "is nested in".

The second a third view in Figure 2 are simple graphical translations of the instance structure as it is saved in the application. The second view is delivered as a Java applet and allows the user to zoom, re-arrange (manually or by selecting a predefined auto-layout algorithm) and edit any displayed element. The result of these modifications can eventually be saved locally as an image. The third view in Figure 2 is built using just simple HTML code. Each box represents either a level, a level instance or a role. If there are more roles, of the same kind, associated to the same level instance, those roles are grouped in one single box whose name is the name of the grouped roles. The elements contained in a role box, are the values associated to the roles grouped in the box. Each role box features a different background color which defines the type of role it represents (class, method or field). If possible, a link pointing to the source code where the element is defined is placed beside the role value.

The first view in Figure 2 provides a synthetic and easy to read overview of a given instance, while the second and third views offer a more detailed representation of its structure. These last two views are useful in case the user has familiarity with the meta-model described in section IV and needs to analyze the pattern structure in its full complexity (i.e.: distinguishing roles by their position in the level hierarchy).

In addition to the above described views, the application provides the following additional views:

- the source code of any class, method and field referenced in the instance (only if explicitly specified in the uploaded input file);
- the Javadoc page of any class, method and field referenced in the instance (only for instances belonging to Java projects).

The first additional view contains color-coded and navigable source code. The code line of the elements contained in the DP instance, if not explicitly specified during the upload process, is estimated by doing a simple full text search on the displayed source file. The last additional view provides navigable pages, as generated by the javadoc tool bundled with the Java SDK.

Moreover, it is also possible to display two facing views in a full screen page overlay. This allows to easily compare and cross-analyze the information provided from two of the five above mentioned views.

We currently host more than 700 DP instances. Some of them have been generated by 2 DPD tools (DPD-Tool [6] and WOP [9]), others have been extracted from a verified dataset (P-MARt [14]).

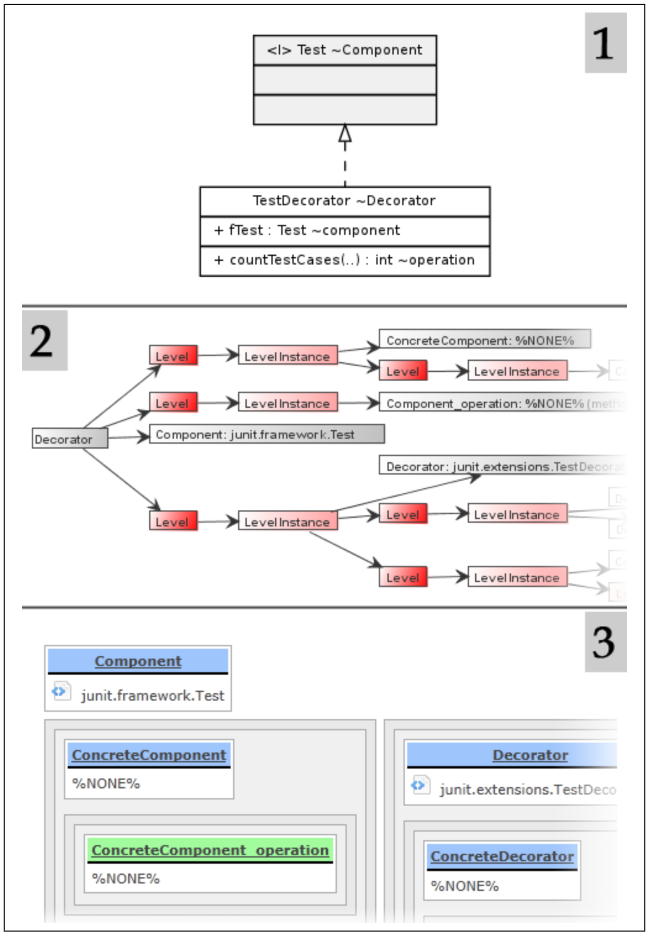

1
<I> Test ~Component
TestDecorator ~Decorator
+ fTest : Test ~component
+ countTestCases(..) : int ~operation
2
Decorator
Level
Level Instance
ConcreteComponent: %NONE%
Component_operation: %NONE%
Component: junit.framework.Test
Decorator: junit.extensions.TestDecor
3
Component
junit.framework.Test
ConcreteComponent
%NONE%
ConcreteComponent_operation
Decorator
junit.extensions.TestDeco
ConcreteDecorator

Figure 2. Three graphical views of the same instance

B. Evaluation

After viewing an instance, the user is invited to express his opinion regarding the correctness of the reported instance (see Figure 3). To do so, he needs to specify the following information:

- a score in the range of 1 to 5 "Stars", where 1 means the instance is fully incorrect, and 5 means it is correct;
- a comment to argument the evaluation;

To avoid semantic misinterpretation, each rating option is associated with a textual description.

Other users are able to express an agreement or disagreement ("Vote") with a previous evaluation. The difference between positive and negative "Votes" defines the number of points gained by that evaluation.

If this number is below a certain threshold, the evaluation (and all the related comments) is automatically collapsed and hidden from the user's sight.

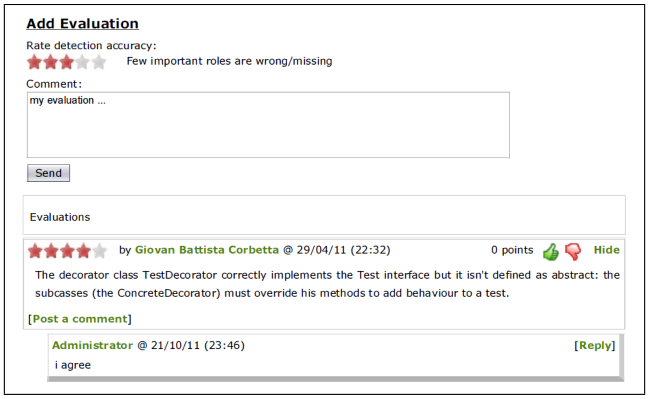
The overall rating associated with the instance is computed as the average of "stars" weighted with the number of points associated to each evaluation. This information is used in the search results (see Figure 4) to have an immediate idea about the number of correct instances found within a certain project analysis.

Discussions can be started by adding comments to existing evaluations. All existing comments are organized in the form of a forum thread, allowing users to easily follow the flow of conversations and to reply to specific messages without having to specify the context of their comment.

We believe that this community driven system of classifying found instances will provide good common datasets, for the test of new tools and the enhancement of the existing ones.

To obtain reliable and consistent results, we only accept evaluations provided by users with proven experience in the field (i.e.: DPD tool developers) and/or sufficient theoretical background (i.e.: students having attended at least one course covering GoF DPs).

We have currently collected more than 160 evaluations, and we hope to see more in the future.



Add Evaluation

Rate detection accuracy:

★★★☆☆ Few important roles are wrong/missing

Comment:

my evaluation ...

Send

Evaluations

★★★★☆ by Giovan Battista Corbetta @ 29/04/11 (22:32) 0 points Hide

The decorator class TestDecorator correctly implements the Test interface but it isn't defined as abstract: the subcasses (the ConcreteDecorator) must override his methods to add behaviour to a test.

[Post a comment]

Administrator @ 21/10/11 (23:46) [Reply]

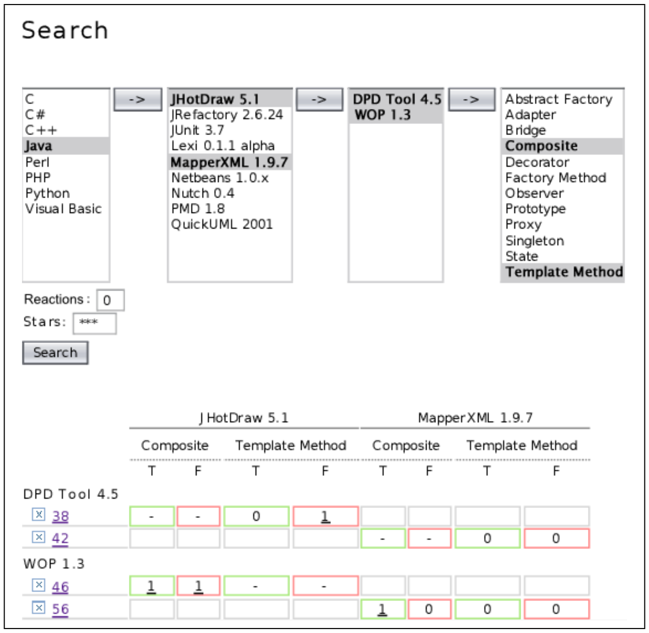
i agree

Figure 3. An example of evaluation

C. Search

The user can search a pattern instance according to the chosen programming language, project, detection tool, and design pattern. An example of a search session is shown in Figure 4. In the user interface, whenever the user chooses a filter, the page shows the available further filters, populating the right hand placed select menu. In each selection list, it is possible to choose more than one value, to make the search more flexible and personalized.

Figure 4 shows an example of search parameters and results, in the case the user looks for the analyses of projects written in the Java language, on the JHotDraw 5.1 and MapperXML 1.9.7 projects, analyzed by the DPDTool 4.5 and WOP 1.3 tools and filtered on the "Composite" and "Template Method" design patterns.



Search

C
C#
C++
Java
Perl
PHP
Python
Visual Basic

->

JHotDraw 5.1
JRefactory 2.6.24
JUnit 3.7
Lexi 0.1.1 alpha
MapperXML 1.9.7
Netbeans 1.0.x
Nutch 0.4
PMD 1.8
QuickUML 2001

->

DPD Tool 4.5
WOP 1.3

->

Abstract Factory
Adapter
Bridge
Composite
Decorator
Factory Method
Observer
Prototype
Proxy
Singleton
State
Template Method

Reactions : 0
Stars : ***

Search

	JHotDraw 5.1				MapperXML 1.9.7			
	Composite		Template Method		Composite		Template Method	
	T	F	T	F	T	F	T	F
DPD Tool 4.5								
☒ 38	-	-	0	1				
☒ 42					-	-	0	0
WOP 1.3								
☒ 46	1	1	-	-				
☒ 56					1	0	0	0

Figure 4. Example of a pattern instance search result

The user can specify the minimum overall rating score a pattern instance must have to be considered a correct instance. In addition, the user can specify the absolute number of "reactions" (evaluations plus "votes") each instance must have to be safely included in the overall correct/not correct count. Currently, the application allows the users to choose a number of stars in the range 1-5, and a number of "reactions" from 0 to 20.

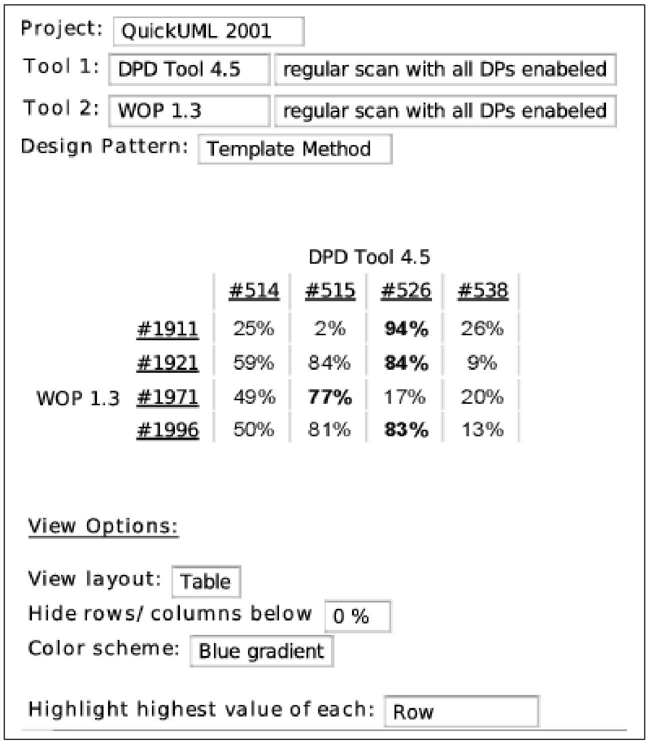
The table (see lower part of Figure 4) shows the results, with the analyses generated by the selected tools on the rows and the projects with the patterns on the columns. Each cell is the combination of an analysis, made by a certain tool, and a pattern belonging to a project. It contains the number of instances considered correct and the number of instances considered incorrect, respectively placed under the labels "T" and "F". An instance is considered correct if it has a number of "reactions" and an overall rating score which are both greater than or equal to the values specified in the above described "stars" and "reactions" filters.

In the example, the results contain 2 analyses for each of the selected DPD tools (38, 42 for DPD Tool 4.5 and 46, 56 for WOP 1.3). Looking at the numbers contained in the first row, we see that the analysis named "38" contains no instances of the pattern Composite, and some instances of Template Method. Of those instances, 1 has been evaluated as a false positive and none has yet been proven correct. Analysis "38" only contains results associated to the JHotDraw project.

When an analysis does not contain instances of a particular design pattern in a particular project, the corresponding cell contents are replaced by a "-" in light grey. If the project has not even been analyzed, the cells are empty.

D. Comparison

The user can compare the results produced by two different analyses, obtained by two different tools, on the same project and for the same chosen pattern definition. The comparison results (see Figure 5) are shown in a table where the two instance sets, found in the context of the selected two analyses, are shown one on the rows and the other on the columns. The cells of the table contain values indicating the similarity of the corresponding couple of instances. The similarity is currently evaluated through an algorithm described in section V. To make reading easier, the background color of each cell is proportional to its percentage value, according to the selected color scheme (red:0% to green:100% or white:0% to blue:100%).



Project: QuickUML 2001

Tool 1: DPD Tool 4.5 | regular scan with all DPs enabeled

Tool 2: WOP 1.3 | regular scan with all DPs enabeled

Design Pattern: Template Method

		DPD Tool 4.5			
		#514	#515	#526	#538
WOP 1.3	#1911	25%	2%	**94%**	26%
	#1921	59%	84%	**84%**	9%
	#1971	49%	**77%**	17%	20%
	#1996	50%	81%	**83%**	13%

View Options:

View layout: Table

Hide rows/ columns below 0 %

Color scheme: Blue gradient

Highlight highest value of each: Row

Figure 5. Example of a result comparison

Figure 5 shows an example of comparison: the user chooses to compare the instances of the Template Method pattern found on project QuickUML 2001 by:

- the DPD Tool 4.5 tool in the analysis named “regular scan with all DPs enabled”;
- the WOP 1.3 tool in the analysis named “regular scan with all DPs enabled”.

The table shows that each tool found four instances, and that for example instances #1911 and #526 are very similar, with a score of 94%, and instances #1911 and #515 are very different, having a score of only 2%. The numbers in bold are the highest value in their row: in fact the last option selected is to highlight the highest value of each row; it is also possible to do the same on the columns. This option simplifies the task of identifying the most similar pairs of instances.

The view can be also filtered removing all the rows or columns containing values which are less than or equal to a certain specified amount. This allows to simplify the table and to include only meaningful results.

To support the user in the process of identifying valid instances, we also added various sets of instances, which have been collected and verified by human experts. These instances have all been extracted from a repository called “P-MARt” [14]. In order to compare a DPD tool generated analysis against the said set of instances, the user needs to select an analysis created by the tool called “P-MARt”.

Clicking on the value representing the similarity score of two instances, opens up a new page which provides a detailed outlook of all the differences existing between those instances (see Figure 6). The information contained in this page is presented in the form of a single “nested box” diagram (see section III-A), which combines the roles of both selected instances into one merged representation, where equivalent role values are grouped in one box and displayed side by side. Spotting differences is made easy through the coloring of role values (strings that are equal are colored in green, while all the others are red) and the option to explore the data by applying filters.

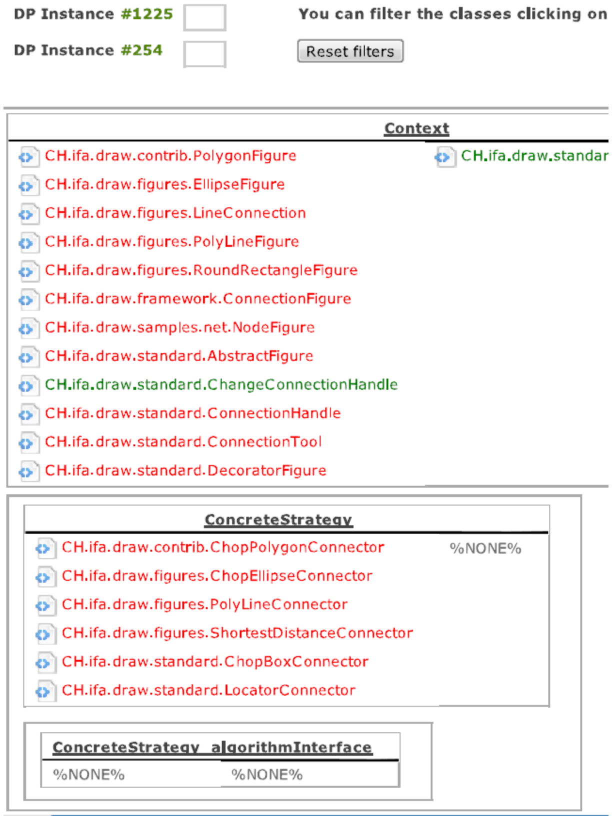
E. Browsing

The Browse section offers a tree view of the content of the application. Basically it allows the user to find all the analyses made by a tool on a project: the user can choose as entry point both the available projects or tools. Whenever the user clicks on one of them, the page shows the analyses available in combination with the other. Clicking on the name of a certain analysis a page is shown, containing the list of all found DP instances, grouped by design pattern. This feature can be experimented by accessing the Browse section [18] of the DPB website.

IV. DP REPRESENTATION META-MODEL

In order to work on design pattern instances, we need a way to represent them in some kind of data structure. In [19], we presented a meta-model for the representation of DP definitions and instances. That meta-model has been adapted to be used to represent DP definitions and instances on the DPB web application.

The model specifies that a design pattern can be defined from the structural point of view using the roles it contains and the cardinality relationship between couple of roles. A similar approach is also suggested by Guéhéneuc et al. [20]. A design pattern is defined as a tree whose nodes are called *Levels*; each level has to contain at least one of the roles



DP Instance #1225 □

DP Instance #254 □

You can filter the classes clicking on

Reset filters

Context

- CH.ifa.draw.contrib.PolygonFigure
- CH.ifa.draw.figures.EllipseFigure
- CH.ifa.draw.figures.LineConnection
- CH.ifa.draw.figures.PolyLineFigure
- CH.ifa.draw.figures.RoundRectangleFigure
- CH.ifa.draw.framework.ConnectionFigure
- CH.ifa.draw.samples.net.NodeFigure
- CH.ifa.draw.standard.AbstractFigure
- CH.ifa.draw.standard.ChangeConnectionHandle
- CH.ifa.draw.standard.ConnectionHandle
- CH.ifa.draw.standard.ConnectionTool
- CH.ifa.draw.standard.DecoratorFigure

CH.ifa.draw.standar

ConcreteStrategy

- CH.ifa.draw.contrib.ChopPolygonConnector
- CH.ifa.draw.figures.ChopEllipseConnector
- CH.ifa.draw.figures.PolyLineConnector
- CH.ifa.draw.figures.ShortestDistanceConnector
- CH.ifa.draw.standard.ChopBoxConnector
- CH.ifa.draw.standard.LocatorConnector

%NONE%

ConcreteStrategy	algorithmInterface
%NONE%	%NONE%

Figure 6. Example of a detailed comparison

of the pattern and it can contain other recursively nested levels. Figure 7 shows the tree structure of the *LevelDef* class (representing the level definition), and the *RoleDef*s that it owns; finally, *DpDef* defines that a design pattern definition is a tree having as root one *LevelDef*.

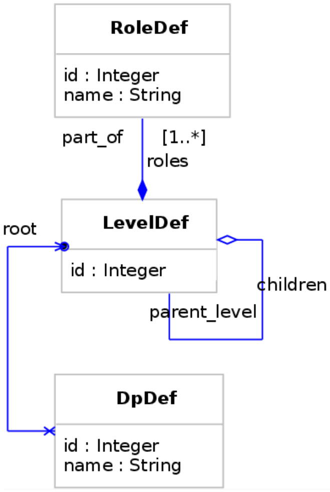


Figure 7. DP Definition UML class diagram

When two roles are contained within the same level, they are in a one-to-one relationship; instead when a role is placed in a nested level it means that for each instance of the roles in the parent level, there can be many sets of roles of the child level. The most common case is when a pattern defines that a class must extend another class. In most cases we identify a single instance of that pattern as the parent class connected with all the children classes. Instances are modeled as in Figure 8; the model is simply an extension of the definition, as it models the instantiation of the concepts contained in the definition: a *RoleAssociation* is the realization of a *RoleDef*, a *LevelInstance* is the realization of a *LevelDef*, and so on. The only complex detail is the splitting of *Level* and *LevelInstance*; the explanation is that each *LevelDef* is instantiated as a *LevelInstance* when the *RoleAssociation*s are filled, but to define a child *Level*, we must specify which particular parent instance it belongs to.

The XML format for specifying a design pattern instance is modeled according to the representation model, so it follows the same concepts. The XML Schema Definition, which the submitted XML file must comply to, is available at [21]; moreover, the list of the pattern definitions the application supports, is available at [22].

To have more chances of being able to compare results coming from different tools, currently users cannot supply their own pattern definitions. When the application will be more tested and filled with data, we will add this functionality. Meanwhile we accept and encourage suggestions coming from the users about new definitions or mistakes in the current ones.

In listing 1, we report an XML file example that represents an instance of an Abstract Factory design pattern.

The file refers to the roles *Abstract Factory*, *Concrete Factory*, *Abstract Product*, *Concrete Product*. Each of them is associated to a class name, an optional location path, line number and comment, and is organized following the Abstract Factory definition.

The here described meta-model has the same purpose as the previously mentioned exchange format DPDX. The main differences between these two models can be summarized as follows: the DPDX format is much more verbose, extensive and difficult to read, provides no shared model for DP definitions and does not allow for simple automatic schema compliance verification. All these elements make the DPDX meta-model not suitable for practical use in a web-based environment (files tend to get too large and difficult to parse), force the users to provide more information than necessary, and make it difficult to create a shared repository of reference models to which any DP instance has to comply to. The meta-model described in this section provides a more essential way of describing data, a layered data structure which allows easy organization of multi-valued roles, and an effective encoding policy which supports fast and easy schema compliance verification.

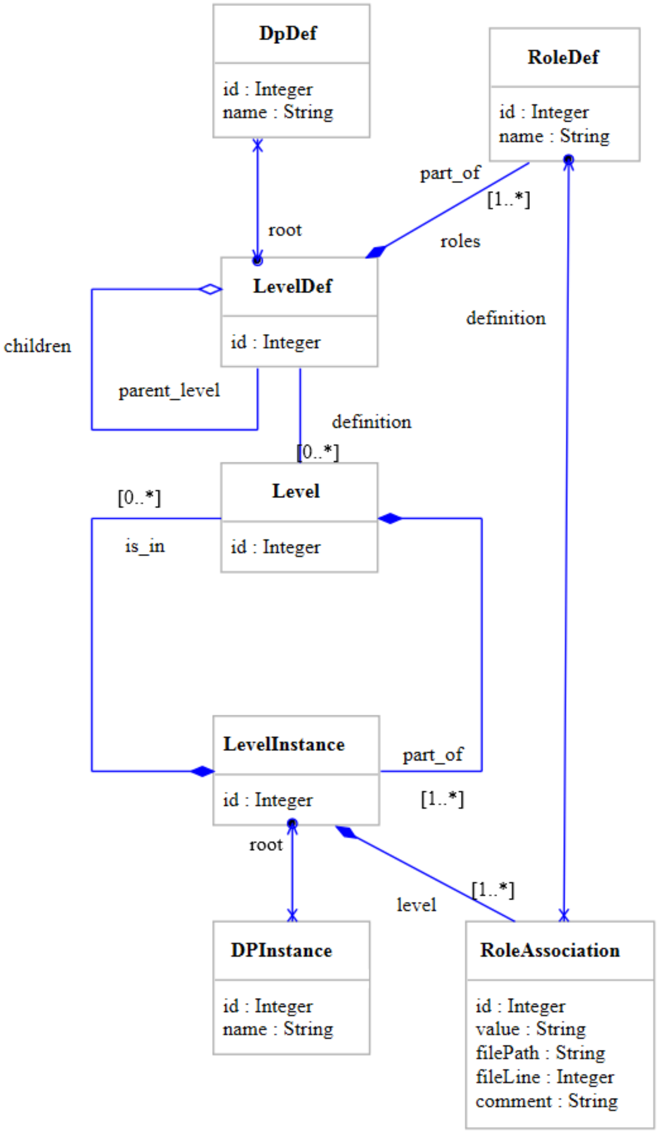


Figure 8. Model UML class diagram

V. Comparison algorithm

In the current version of the benchmark application, we defined and implemented a comparison algorithm specifically developed for the meta-model defined in the previous section. The algorithm tries to express the similarity of two instances giving more weight to the parent roles and less to the children roles, and produces a number between 0 (total difference; meaning that all role values are different) and 1 (total equivalence; meaning that all role values are equal).

A. Algorithm description

Our algorithm gives a descending score to each level depth in the pattern definition according to the formula 1.

Listing 1. Example of Abstract Factory instance

```
<analysis><pattern name=”AbstractFactory”><patternInstance>
<level><levelinstance>
  <role name=”AbstractFactory” value=”foo.Class1”>
    <location file=”foo/Class1.java” line=”13” />
    <comment>Some text</comment>
  </role>
  <level><levelinstance>
    <role name=”AbstractProduct” value=”foo.Class1”>
      <location file=”foo/Class1.java” line=”53” />
      <comment>Some text</comment>
    </role>
    <level><levelinstance>
      <role name=”ConcreteProduct” value=”foo.Class1”>
        <location file=”foo/Class1.java” line=”93” />
        <comment>Some text</comment>
      </role>
    </levelinstance></level>
  </levelinstance></level>
  <level><levelinstance>
    <role name=”ConcreteFactory” value=”foo.Class1”>
      <location file=”foo/Class1.java” line=”93” />
      <comment>Some text</comment>
    </role>
    <level><levelinstance>
      <role name=”ConcreteFactory_createProduct”
        value=”foo.Class.Method(java.lang.String,int)”>
        <location file=”foo/Class1.java” line=”133” />
        <comment>Some text</comment>
      </role>
    </levelinstance></level>
  </levelinstance></level>
  <level><levelinstance>
    <role name=”AbstractFactory_createProduct”
      value=”foo.Class.Method(java.lang.String,int)”>
      <location file=”foo/Class1.java” line=”133” />
      <comment>Some text</comment>
    </role>
  </levelinstance></level>
</levelinstance></level>
</patternInstance></pattern></analysis>
```

$$depthScore_{depth} = \log_{10}(treeHeight - depth) + 1 \tag{1}$$

This weight assignment schema is based on the assumption that roles forming a given DP definition are placed on levels ordered by decreasing importance.

The use of a logarithmic function is aimed to reduce the difference in weight between the levels of the pattern.

Consider the following example: in a definition with only a parent level and a child level, the parent level depth takes a score of $\log_{10}(2) + 1$ and the child level depth takes a score of $\log_{10}(1) + 1$; with three level depths, the first one (the root) would take weight $\log_{10}(3) + 1$, and so on.

Then each level in the definition takes the score of the depth it belongs to, and an overall score is calculated as the sum of the score of all levels. Finally a weight is assigned to each level dividing the level score by the overall score. In this way the sum of all the weights is 1. The following formula sums up what said above:

$$weight_i = \frac{depthScore_i}{\sum_{j=0}^{treeH} depthScore_i \cdot |levels_i|} \quad (2)$$

where:

- $treeH$ is the height of the tree structure representing the analyzed pattern.
- $levels_i$ is the set of all the levels located at depth i.
- $depthScore_i$ is the score associated to depth i, as defined in formula 1.

When the weights are set, the algorithm compares a couple of instances starting from the root and recursively distributing the weight of each level on its level instances. The similarity score between two equivalent levels is computed by the following equation:

$$\begin{aligned} & simL(l_1, l_2, depth) = \\ & \textstyle\sum_{i=0}^{n} simLI(subLi_{1,i}, subLi_{1,i}'') \cdot weight_{depth} \cdot 1/n \\ & + \textstyle\sum_{i=0}^{m} simL(subL_{1,i}, subL_{1,i}', depth + 1) \\ & \\ & simLI(li_1, li_2) = \frac{|sharedRoles|}{\max(|subRoles_1|, |subRoles_2|)} \end{aligned} \quad (3)$$

where:

- l_1 and l_2 are the two levels, located at a certain $depth$, which are to be compared. They respectively belong to the first and the second instance to be compared.
- $subLi_{1,i}$ is the i-th instance of level l_1.
- $subLi_{1,i}''$ is the instance of level l_2, which is most similar to $subLi_{1,i}$. Similarity between level instances is calculated by applying the function $simLI$, which simply provides a percentage based on the number of roles of the same kind, associated to the level instances passed as arguments, sharing the same values.
- $subL_{1,i}$ is the i-th child level of level l_1.
- $subL_{1,i}'$ is the child level of l_2 which is equivalent to $subL_{1,i}$. Equivalence between levels is deduced by comparing the references to the common DP definition. If those references are identical, the two levels are equivalent.
- $weight_{depth}$ is the weight associated to the levels located at the specified $depth$, as defined in formula 2.
- $n = \max(|subLis_1|, |subLis_2|)$ where $subLis_i$ is the set of instances of l_i.
- m is the number of child levels of l_1 (or ,equivalently, of l_2).
- $sharedRoles$ is a set made of all roles which are common to both li_1 and li_2.
- $subRoles_i$ is the set of roles associated to li_i

Both functions described in formula 3 are symmetric with respect to their first two arguments, meaning that $simL(l_1, l_2, depth) = simL(l_2, l_1, depth)$ and $simLI(li_1, li_2) = simLI(li_2, li_1)$.

The overall similarity score between two DP instances(i_1 and i_2) is calculated as follows:

$$sim(i_1, i_2) = \begin{cases} simLI(root_1, root_2) \cdot weight_0 \\ + \sum_{i=0}^{n} simL(subL_{1,i}, subL_{1,i}', 1) \\ \qquad [if\ simLI(root_1, root_2) > 0] \\ \\ 0 \quad [otherwise] \end{cases} \quad (4)$$

where:

- $root_i$ is the root level instance associated to the i-th DP instance.
- $subL_{1,i}$ is the i-th child of the level instance $root_1$.
- $subL_{1,i}'$ is the child of level instance $root_2$, which is equivalent to $subL_{1,i}$.
- $n = \max(|subLs_1|, |subLs_2|)$ where $subLs_i$ is the set of children of level instance $root_i$.

B. Example

To better understand how the comparison algorithm works, consider the simple example described below. Listing 2 contains the description of two Abstract Factory instances. Both instances are represented as tree structures where each line corresponds to a node and indentation is used to specify hierarchical relationships.

Each node of type "LevelInstance" is formatted as follows:

$$LI : NodeLabel => Roles : \\ \{RoleName : RoleValue, ..\}$$

The roles listed on the right side, are those to which the level instance element is associated to. Roles are considered equivalent if they have the same name and value.

Each node of type "Level" is formatted as follows:

$$L : NodeLabel => ID : IDValue$$

Two levels are considered equivalent if they have the same IDValue.

Now let's simulate the steps needed to calculate the similarity value between these two instances. First of all, we need to define the weights to assign to each level based upon the depth it is located at:

$$\begin{aligned} depthScore_0 &= \log_{10}(3-0) + 1 = 1.48 \\ depthScore_1 &= \log_{10}(3-1) + 1 = 1.3 \\ depthScore_2 &= \log_{10}(3-2) + 1 = 1 \end{aligned} \quad (5)$$

$$\begin{aligned} & \textstyle\sum_{j=0}^{treeH} depthScore_i \cdot |levels_i| = \\ & = 1.48 \cdot 1 + 1.3 \cdot 2 + 1 \cdot 1 = 5.08 \end{aligned} \quad (6)$$

Listing 2. Two Abstract Factory instances to be compared

```
================================================
instance 1
================================================
LI:Root11  => Roles: {AF:A}
    L:L_11 => ID: 1
        LI:LI_11 => Roles: {AP:B, AP:C}
            L:L_12 => ID: 2
                LI:LI_12 => Roles: {CP:D, CP:E, CP:F}
                LI:LI_13 => Roles: {CP:G}
        LI:LI_14 => Roles: {AP:H, AP:I}
            L:L_13 => ID: 2
                LI:LI_15 => Roles: {CP:L, CP:M, CP:N}
        LI:LI_16 => Roles: {AP:O}
            L:L_14 => ID: 2
                LI:LI_17 => Roles: {CP:P}
    L:L_15 => ID: 3
        LI:LI_18 => Roles: {CF:Q}
```

```
================================================
instance 2
================================================
LI:Root21 => Roles: {AF:A}
    L:L_21 => ID: 1
        LI:LI_21 => Roles: {AP:B, AP:C}
            L:L_22 => ID: 2
                LI:LI_22 => Roles: {CP:D, CP:E}
                LI:LI_23 => Roles: {CP:X}
        LI:LI_24 => Roles: {AP:H}
            L:L_23 => ID: 2
                LI:LI_25 => Roles: {CP:L, CP:Y}



    L:L_24 => ID: 3
        LI:LI_26 => Roles: {CF:Z}
```

$$\begin{aligned} weight_0 &= 1.48/5.08 = 0.29 \\ weight_1 &= 1.3/5.08 = 0.26 \\ weight_2 &= 1/5.08 = 0.2 \end{aligned} \tag{7}$$

Next we recursively compare each level, its instances and the roles which its associated to.

$$\begin{aligned} &sim(inst1, inst2) = simLI(root_{11}, root_{21}) \cdot weight_0 \\ &+ simL(L_{11}, L_{21}, 1) + simL(L_{15}, L_{24}, 1) \end{aligned} \tag{8}$$

Level comparison at depth 1:

$$\begin{aligned} &simL(L_{11}, L_{21}, 1) = (simLI(LI_{11}, LI_{21}) \\ &+ simLI(LI_{14}, LI_{24}) + simLI(LI_{16}, null)) \cdot weight_1 \cdot 1/3 \\ &+ (simL(L_{12}, L_{22}, 2) + simL(L_{13}, L_{23}, 2) \\ &+ simL(L_{14}, null, 2)) \\ \\ &simL(L_{15}, L_{24}, 1) = simLI(LI_{18}, LI_{26}) \cdot weight_1 \cdot 1/1 \end{aligned} \tag{9}$$

Level comparison at depth 2:

$$\begin{aligned} &simLI(LI_{11}, LI_{21}) = 1 \\ &simLI(LI_{14}, LI_{24}) = 0.5 \\ &simLI(LI_{16}, null) = 0 \\ &simL(L_{12}, L_{22}, 2) = (simLI(LI_{12}, LI_{22}) \\ &+ simLI(LI_{13}, LI_{23})) \cdot weight_2 \cdot 1/2 \\ &simL(L_{13}, L_{23}, 2) = (simLI(LI_{15}, LI_{25})) \cdot weight_2 \cdot 1/1 \\ &simL(L_{14}, null, 2) = 0 \\ &simLI(LI_{18}, LI_{26}) = 0 \end{aligned} \tag{10}$$

Comparison of the tree leaves:

$$\begin{aligned} simLI(LI_{12}, LI_{22}) &= 0.66 \\ simLI(LI_{13}, LI_{23}) &= 0 \\ simLI(LI_{15}, LI_{25}) &= 0.33 \end{aligned} \tag{11}$$

Solving the equation brings to the following result:

$$sim(inst1, inst2) = 1 \cdot 0.29 + 0.262 + 0 = 0.552 \tag{12}$$

The similarity score between the two DP instances is equal to 55.2%.

VI. Conclusion

In this paper we presented a web application helping the design pattern detection community having a way to compare the results produced by the tools and the techniques that have been proposed in the literature.

Our final intent is not only the tool "competition" but also the creation of a container for design pattern instances that, through the users' voting, will allow us to build a large and "community validated" dataset for the testing and benchmarking of DPD tools.

For all these reasons we are convinced that this kind of application can be really valuable in this research area because it allows the real sharing of information and knowledge among all research groups interested in design patterns for both reverse and forward engineering.

In future work we are interested in integrating different comparison algorithms, maybe suggested and discussed with the DPD community, in order to let the users choose the algorithm they think is the most appropriate. We are also investigating the possibility to expose some kind of web services in order to provide a mechanism for making the tools able to automate the loading of their analysis into the application. Moreover we are interested to experiment the correctness and completeness of our approach, exchanging data with other applications and models as those cited in Section II.

Other further efforts will be directed towards the creation of a more complete documentation, including examples and case studies. Our priority will be to provide a clear explanation on how to manage and interpret the different results generated by the application.

We are planning to add new results coming from as many tools as possible. We are currenly collaborating with several authors to help them sharing their results. Among them there are Binun and Kniesel, authors of the DPJF tool [23].

We will also continue improving overall usability, focusing on accessibility, affordability and portability (as described in a paper by Sim et al. [24]).

Acknowledgment

The authors would like to kindly thank Günter Kniesel, Nikos Tsantalis and Yann-Gaël Guéhéneuc for all their useful suggestions and great support in our research.

References

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

[2] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177–1193, 2009.

[3] H. Huang, S. Zhang, J. Cao, and Y. Duan, "A practical pattern recovery approach based on both structural and behavioral analysis," *Journal of Systems and Software*, vol. 75, no. 1-2, pp. 69–87, 2005.

[4] M. von Detten, M. Meyer, and D. Travkin, "Reverse engineering with the reclipse tool suite," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA: ACM, 2010, pp. 299–300.

[5] R. A. Olsson and N. Shi, "Reverse engineering of design patterns from java source code," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 123–134.

[6] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, 2006.

[7] Y.-G. Guéhéneuc, "Ptidej: Promoting patterns with patterns," in *Proceedings of the 1st ECOOP workshop on Building a System using Patterns*, M. E. Fayad, Ed. Springer Verlag, July 2005, 9 pages.

[8] 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*. New York, NY, USA: ACM, 2002, pp. 338–348.

[9] J. Dietrich and C. Elgar, "Towards a web of patterns," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 108–116, 2007, software Engineering and the Semantic Web.

[10] Y.-G. Guéhéneuc and G. Antoniol, "DeMIMA: A multilayered approach for design pattern identification," *IEEE Trans. Softw. Eng.*, vol. 34, pp. 667–684, 2008.

[11] F. Arcelli Fontana and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Information Sciences*, vol. 181, no. 7, pp. 1306–1324, April 2011.

[12] F. Arcelli Fontana, M. Zanoni, and A. Caracciolo, "A benchmark platform for design pattern detection," in *Proceedings of The Second International Conferences on Pervasive Patterns and Applications PATTERNS 2010*, IARIA. Lisbon, Portugal: Think Mind, November 2010, pp. 42–47.

[13] L. Fulop, R. Ferenc, and T. Gyimothy, "Towards a benchmark for evaluating design pattern miner tools," in *Proceeding of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, 1-4 2008, pp. 143–152.

[14] Y.-G. Guéhéneuc, "Pmart: Pattern-like micro architecture repository," in *Proceedings of the 1st EuroPLoP Focus Group on Pattern Repositories*, M. Weiss, A. Birukou, and P. Giorgini, Eds., July 2007.

[15] G. Kniesel and A. Binun, "Standing on the shoulders of giants - a data fusion approach to design pattern detection," in *Proceedings of IEEE 17th International Conference on Program Comprehension (ICPC '09.)*. IEEE Computer Society, 2009, pp. 208–217.

[16] G. Kniesel, A. Binun, P. Hegedűs, L. J. Fülöp, N. Tsantalis, A. Chatzigeorgiou, and Y.-G. Guéhéneuc, "DPDX—towards a common result exchange format for design pattern detection tools," in *Proceedings of 14th European Conference on Software Maintenance and Reengineering (CSMR 2010)*, March 2010, pp. 232–235.

[17] ESSeRE, "Browse section," Web Site, 2010, http://essere.disco.unimib.it:8080/DPBWeb/faces/Browse.jsp.

[18] ——, "Design pattern benchmark platform," Web Site, 2010, http://essere.disco.unimib.it/DPB/.

[19] F. Arcelli, C. Tosi, and M. Zanoni, "A benchmark proposal for design pattern detection," in *Proceedings of 2nd Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2008)*, Antwerp, Belgium, 2008, pp. 24–27.

[20] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, nov. 2004, pp. 172–181.

[21] ESSeRE, "Design pattern analysis schema definition," Web Site, 2010, http://essere.disco.unimib.it:8080/DPBWeb/faces/resources/DpAnalysis.xsd.

[22] ——, "Design pattern definitions documentation," Web Site, 2010, http://essere.disco.unimib.it:8080/DPBWeb/faces/Doc_DpDef.jsp.

[23] A. Binun and G. Kniesel, "Joining forces for improving precision and recall of design pattern detection," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR2012)*, Szeged, Hungary, March 2011.

[24] S. Sim, S. Easterbrook, and R. Holt, "Using benchmarking to advance research: a challenge to software engineering," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, may 2003, pp. 74 – 83.