jet INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2015, VOL. 61, NO. 4, PP. 321–326
Manuscript received October 10, 2015; revised December, 2015. DOI: 10.1515/eletel-2015-0041

Searching Design Patterns Fast by Using Tree Traversals

Stefano Cicciarella, Christian Napoli and Emiliano Tramontana

***Abstract*—Large software systems need to be modified to remain useful. Changes can be more easily performed when their design has been carefully documented. This paper presents an approach to quickly find design patterns that have been implemented into a software system. The devised solution greatly reduces the performed checks by organising the search for a design pattern as tree traversals, where candidate classes are carefully positioned into trees. By automatically tagging classes with design pattern roles we make it easier for developers to reason with large software systems. Our approach can provide documentation that lets developers understand the role each class is playing, assess the quality of the code, have assistance for refactoring and enhancing the functionalities of the software system.**

***Keywords*—design patterns, source code analysis, software architecture, tree traversals**

I. Introduction

DEVELOPERS find it difficult to analyse and understand the code of large software systems. Generally, there are several problems that have to be faced: lack of a well-defined and documented software architecture, architectural erosion [1], [2], mixing of concerns [3], etc. Trying to understand the software architecture of a system by only visually inspecting the whole source code can be far too complex and time consuming. Moreover, such an analysis may lead developers to several misunderstandings. Even when a software system is well designed, analysing it would still be difficult for both the huge amount of code, and the different design solutions that different teams use, and which can be difficult to assess by looking at the code alone.

Seeing that software systems need to change due to new requirements or supporting technologies, understanding their structure plays a fundamental role in keeping them useful and maintaining or improving their structure. Knowledge about the existing design solutions and overall architecture of a software system may be acquired by recognising the *design patterns* [4] that have been implemented within the system. Any information about the design patterns is an indication to maintainers about the choices of previous developers, i.e. the role of the class for the design pattern unveils the reason for the low level details, and the motivation for the class. Moreover, once a design pattern has been revealed a change on its classes may be conveniently carried out to introduce new functionalities, for the reason that how the portion of the system comprising the classes of the design pattern would be understood at once. Moreover, even knowing that some classes are not arranged according to any design pattern can be useful, since appropriate refactoring can be carried out [5], [6], [7], [8], or improvements to the modularity can be performed by adopted advanced technologies [9], [10], [11], [12].

Previous approaches that identify design patterns are slower compared to our solution, often inaccurate due to their assumptions, or not fully automated, because relaying on other external tools or human supervision [13], [14] (the detailed comparison with the related work is in Section V). This paper proposes an approach to automatically find design patterns in large software systems fast. Our approach consists of analysing the Java bytecode of a software system in order to discover the constituting classes and their relationships, and build a corresponding class graph. Then, we search design patterns by matching portions of the graph with known characteristics of the roles within design patterns; the candidate classes having some role for a design pattern are then organised as trees, which are traversed to find out whether a whole design pattern exists. Thanks to the proposed search algorithm, finding design patterns is very fast when compared with other approaches. Moreover, an exact match is sought therefore the approach is the most accurate according to the model of the known design pattern.

This paper is structured as follows. Section II gives an overview of our approach. Section III describes the details of our solution. Section IV presents the results of some experiments. Section V compares our solution with other approaches. Finally, Section VI draws our conclusions.

II. Approach Overview

The proposed approach consists of two main phases: (i) the Java bytecode *exploration* that analyses an existing software system and extracts necessary data and (ii) the *searching* for design patterns, as models known beforehand, by properly organising extracted data in order to reduce checks as much as possible.

The exploration phase scans the Java bytecode of the target software system in order to extract the name of all the classes, and the relationships existing among classes. The latter are unveiled by carefully processing the bytecode declaring classes, and the instructions within classes that rely onto other classes (such as method invocations, variable declarations, etc.). The

This work has been supported by project PRISMA PON04a2 A/F funded by the Italian Ministry of University and Research within PON 2007-2013 framework and by project PRIME within POR FESR Sicilia 2007-2013 framework.

S. Cicciarella, C. Napoli and E. Tramontana are with the Department of Mathematics and Informatics, University of Catania, Viale A. Doria 6, 95125 Catania, Italy, (email: stefano.cicciarella@tiscali.itm {napoli, tramontana}@dmi.unict.it).

class relationships we are interested in are: *inheritance*, *use*, *invocation*, *implementation*, *method parameter type*, *return type*, *override*, and *instantiation*, as defined in [15]. These are all the relationships that can be distinguished from the code alone, as e.g. *aggregation* or *composition* cannot be recognised [15]. By extracting all the relationships we can build a graph that represents classes, as nodes, and their relationships as edges. Given that there are different kinds of relationships the corresponding graph edges are labelled accordingly. In our approach, in order to explore the bytecode of a system we use *Computational Reflection* [16], [17] and the support of Javassist libraries [18].

As for the design patterns to be found, we use a Java implementation for each of the well-known design patterns from the GoF's catalogue [4] and, analogously to the software system exploration, we build a "small" graph for each design pattern we want to look for. Then, such design pattern graphs, i.e. *models*, are stored into a catalogue, so that they can be used during search. Moreover, new models can be easily stored in our catalogue at any time, e.g. to cater for correct variants.

After the bytecode exploration phase, the search phase begins and aims at finding any occurrence of each design pattern model into the graph representing the whole target software system. The search phase is organised into several steps. The first step is to examine for a node the edges of the "small" graph for a design pattern model and match them with the ones of a node in the "large" graph for the software system. This lets us find the classes of the examined software system that match one role for a design pattern model, as they have all the needed relationships with other "surrounding" classes as the role prescribes. Each role of the design pattern model is matched independently of other roles.

Starting from the assigned roles, the second step is to check whether all the relationships among classes having given a role correspond to the ones for the selected design pattern model. If there is full correspondence between model and graph relationships, i.e. each relationship between roles in the model exists in the graph, then we will conclude it is a match. On the contrary, if one or more relationships existing for the design pattern model can not have any correspondence, then we conclude that the examined design pattern can not be matched. In this step, in order to reduce the number of possible matches between classes and correspondent roles of a design pattern we use trees. Each tree represents one possible match, and each tree level represents a role that a class can have. During tree traversals, nodes of each level are added as children to the ones of the previous level. This happens only if all the relationships among classes in the same path match the design pattern roles. Details for such an algorithm will be given in the next section, showing design choices and optimisations aimed at obtaining fast execution even when large software systems are analysed.

III. Searching Design Patterns by Building Trees

Our solution has been implemented as a Java tool, named DPRecog, that provides several classes, and among them, classes Explorer, Graph and Recog. Class Explorer analyses the bytecode of the input application and builds the corresponding graph. Method explore() of such a class scans the bytecode instruction by instruction; then returns an instance of class Graph at the end of the task.

A. Exploration and Graph Representation

Class Graph holds a representation of an adjacency matrix that maps the classes and their relationships. This representation allows us to execute most of the needed operations in a constant time. Moreover, thanks to an encoding we avoid to incur into memory waste in case of a large number of classes (that would otherwise bring a large sized matrix). The encoding allows us to have just a byte per element. In this way, a software system consisting of n classes will take up to n^2 bytes, which is a very small amount of memory even for large sized systems.

The encoding consists of using a power of two to represent one of the eight different kinds of relationships between classes (inheritance=1, use=2, method invoke=4, implementation=8, method parameters type=16, return type=32, override=64, instantiation=128). If a given kind of relationship holds between two classes (that is to say a kind of edge connecting two nodes of the graph), then the binary representation of that element of the matrix will be 1 in the proper bit; of course zero will appear otherwise.

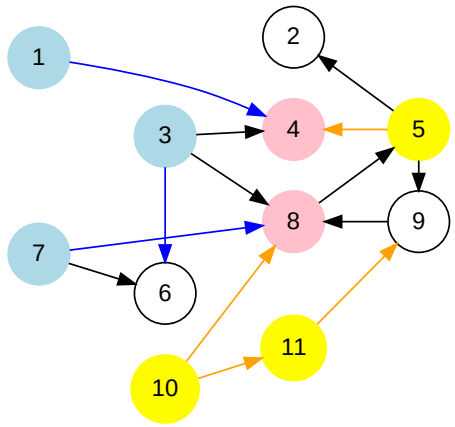
B. Building Sets of Classes for Design Pattern Roles

Each class of the analysed system is assigned to a set, according to which *role* the class could play in a design pattern. A class is recognised to play a role in a design pattern only when all of its relationships with other classes hold as in the design pattern. Enough elements to understand if these conditions are met can be retrieved by means of the graph we have built; in particular we will focus on the incoming and the outgoing edges of any vertex.

Therefore, several *sets* are filled with suitable classes, each set corresponds to a role in a design pattern. E.g. for design pattern Observer, a set will be build containing all the classes from the system that can play the role *Subject*, another set containing the classes for role *ConcreteSubject*, etc. The number of sets is the same as the number of roles prescribed by a design pattern. Of course the smaller the sets, the faster the search. In our implementation each design pattern model is handled by a dedicated Java thread, hence more design patterns at once are examined in a multi-core host. Building such sets will be instrumental for drastically decreasing the number of checks to be performed when searching for design patterns.

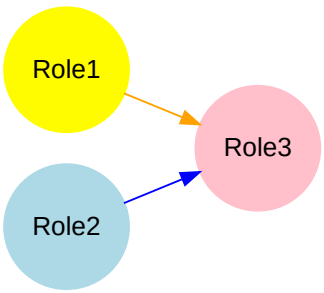
Both the procedure to fill the sets and the search itself are implemented by class Recog. Sets are represented by arrays and the method fillSets() accomplishes the task of filling them with the ids of the proper classes. For this, we use the data stored in the Graph class during the exploration phase (each vertex is associated to two bytes; one describes the types of incoming edges the other is for the types of outgoing edges).

Figure 1 left shows a simple application represented as a graph, where each node is a class and each edge is a relationship between classes. Relationships (and roles) are displayed



(a)

A graph representation of a simple application, where each class has been marked according to its possible role for the design pattern



(b)

The roles and their relationships for a design pattern model, where each role has been marked by a different colour

Fig. 1: An example of matching between roles for a design pattern and candidate classes for an application. Accordingly, Role1 (yellow) matches classes 5, 10, 11; Role2 (lightblue) matches classes 1, 3, 7; and Role3 (pink) matches classes 4, 8.

using different colours, with reference to the design pattern shown in the rightside. Hence, each role of the design pattern is marked in the application graph using the corresponding colour, and each relationship in the design pattern is also marked in the application graph. The given representation is very simplified, whereas a real system under analysis will have a much bigger number of classes and relationships. Moreover, all the sought design patterns will have to be modelled.

C. Matching Whole Design patterns

Once the sets for the roles have been filled, an element of each set will be positioned as a node of a tree. We associate every set to a tree level, so that the elements of the first set will be the possible root nodes, the elements of the second set will be the possible nodes of the second level, etc. Then for each tree, we perform a traversal, which consists of the checks that ensure the match between the edges of the positioned class and the related ones in the design pattern model. If just one check fails, then the whole branch of the tree will be truncated. If the traversal keeps going on, and all checks succeed till the leaf node, we will store a match. The search ends when all the possible paths have been explored. Figure 2 shows the above search algorithm for the simple application and for the design pattern model shown in Figure 1.

This algorithm (that is an iterative tree traversal) is implemented by the recognise() method of class Recog. For each vertex in the path, the vertexSuits() method is invoked to perform all the checks about the edges. The references to the vertices having passed all the checks at the current state are pushed into a stack structure so that, when a path has been completed, its elements can be simply stored as an occurrence of a design pattern.

D. Performance Optimisation

Before the traversals begin, the sequence of sets representing role filled with candidate classes (which in our implementation is a sequence of arrays) is ordered by ascending cardinality, so that the set containing the root nodes has the smallest number of elements and the set containing the leaf nodes has the largest one. I.e. we determine the nodes in each level by giving the highest priority to the smallest sets. While such an order does not affect the correspondences that will be found, the performances are greatly increased.

A simple observation can help us to understand the way in which the execution time could change by reordering the sets. Generally, the number of combinations for classes that are playing a role for a design pattern are much less than the total number of possible combinations for classes; this implies that most tree paths will be truncated during the search. Hence, the best strategy appears to perform the exploration by beginning from the smallest sets. In all likelihood, this choice will allow us to truncate soon the useless paths, having performed only a small number of checks. As we will show in the following, experimental data have confirmed that by using this optimisation we have sensibly improved the efficiency of the search.

IV. Results

A. Performances

Most of the effort we made during the development of our approach was focused on improving performances. Tests to estimate the execution speed have been performed using the Java software JHotDraw (version 6.0b1) as an input application. JHotDraw is an open source graphic framework, which was created as a design exercise and now it is often used as a case of study for software engineering issues. It contains many design patterns and is provided with documentation, hence it is an ideal input application to test our method.

We have executed our tests by using an Intel Core i7 Q720 with 1.60 GHz and 4 GiB RAM, both on Linux Ubuntu 11.4 and Microsoft Windows 7. The tests have been done on the

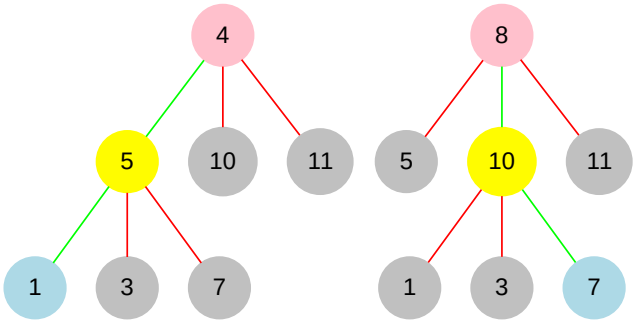


Fig. 2: Trees representing a design pattern, where each level is a described role, and each tree is a possible design pattern occurrence, filled by properly positioning classes at the different levels according to their role. The sought model is found matching classes 4, 5, 1 for the first occurrence, and classes 8, 10, 7 for the second occurrence; the link colour represents whether a check succeeds (green) or fails (red).

TABLE I: Measured Execution Times in Milliseconds

	JHotDraw	java.awt	JUnit	JEdit
Classes	600	590	252	1267
Reading catalogue	13	4	4	16
Building graph	460	366	358	1043
Searching for patterns	111	68	78	85
Total	583	448	440	1144

TABLE II: Found Design Patterns

	Adapter	Bridge	Composite	Decorator	Proxy
JHotDraw	11	12		1	1
java.awt	10		2	1	5
JUnit	1		1	2	1
JEdit	2	1	1		1

whole JHotDraw (six hundred classes), using a catalogue containing seventeen design pattern models. The outcome showed that running the full analysis took barely 700 milliseconds. By observing the execution times we can value the importance of ordering the sets by ascending cardinality. In fact, without such an optimisation, the whole analysis took a few seconds to run. A so remarkable performance improvement is certainly due to the great amount of checks we have avoided by truncating the dead ending branches at the beginning of the traversals.

We have analysed several applications, hence testing the execution time of our tool. Table I shows the execution times on JHotDraw, the package java.awt, JUnit and JEdit.

Table II contains the number of occurrences found for five design patterns, i.e. *Adapter*, *Bridge*, *Composite*, *Decorator*, *Proxy*, when searching, by executing our tool, on the said applications.

As an example of the outcomes of our tool we report the findings for the java.awt library. Such findings have been confirmed as true positive by visually inspecting the actual code. Table III shows the mapping of roles to classes for design pattern *Composite*, which has been found as two occurrences. Design pattern *Decorator* has been found as one occurrence and the mapping of roles to classes is given in Table IV. Five occurrences of design pattern *Proxy* have been found and are shown in Table V.

TABLE III: Design Pattern Composite Found in java.awt

Component	Composite	Leaf
java.awt.Window	java.awt.Dialog	java.awt.Frame
java.awt.Component	java.awt.Container	java.awt.TextComponent java.awt.Scrollbar java.awt.List java.awt.Label java.awt.Choice java.awt.Checkbox java.awt.Canvas java.awt.Button

B. Accuracy

The accuracy of our approach is related to the characterisation that we manage to give to a design pattern model, which we have defined according to the relationships that classes have (see Section II) and that can be detected by automatically processing the code of a system under analysis. However, design patterns can be found on software systems in a variant, rather than the original definition [4]. Some of the related works in pattern mining classify such variations as true positives, other ones go through a manual classification of the implementation, and then a false positive is sometimes associated. Accordingly, the accuracy measurements are different from one another. As we can see in Section V, many other approaches approximate the matching between a design pattern model an the code from the system, in order to cater for variations, and include these as findings. Hence, given that a commonly agreed definition of true positive for an occurrence of a design pattern may not be available, we can not precisely compare the accuracy of the several approaches.

Our algorithm is a novel implementation of an exact graph matching, hence we can state that only a sub-graph that is the exact copy of the model we are searching for will be given as a match. Therefore, some class relationships that are similar to models (however, not exactly the same) will be discarded, ignoring whether or not they represent a different version of the design pattern. This is not a limitation for our approach given that any number of models, hence variations of known design patterns, can be given to our tool to analyse software systems, as far as they can be described in terms of the said relationships between classes.

TABLE IV: Design Pattern Decorator Found in java.awt

Component	ConcreteComponent	Decorator	ConcreteDecorator
java.awt.image.ImageConsumer	java.awt.image.PixelGrabber	java.awt.image.ImageFilter	java.awt.image.RGBImageFilter java.awt.image.ReplicateScaleFilter java.awt.image.CropImageFilter java.awt.image.BufferedImageFilter

TABLE V: Design Pattern Proxy Found in java.awt

Proxy	RealSubject	Subject
java.awt.image.BufferedImage	java.awt.image.ColorModel	java.awt.Transparency
java.awt.Polygon	java.awt.Rectangle	java.awt.Shape
java.awt.Frame	java.awt.MenuBar	java.awt.MenuContainer
java.awt.MenuBar	java.awt.Menu	java.awt.MenuContainer
java.awt.GradientPaint	java.awt.Color	java.awt.Paint

In a different work we have considered how variations of a design pattern can be detected and provide the developers with means to determine which characteristics is fundamental for a correct implementation, hence definition, of a design pattern [2].

V. Related Work

Tsantalis et al. [19] proposed an approach aimed to automatically discover design patterns, that shares some common elements with ours. First of all, the input application bytecode is scanned in order to build a graph representing its structure. Next, inheritance hierarchies are detected in order to create the so called sub-systems (namely portions of the main class graph), which are used to improve performances, i.e. the research will be performed on these sub-graphs rather than on the main structure. Both sub-systems graphs and design pattern graphs are represented by adjacency matrices. The search algorithm consists of computing similarity scores for matrices, in order to obtain an inexact graph matching. The authors chose not to use an exact graph matching algorithm as they state that otherwise the modified versions of design patterns will not fit the used model. Using the similarity score algorithm should allow to find both the standard version of design patterns as they are defined by models, and the possible variations that designers could use.

Guéhéneuc and Antoniol [20] introduced an approach consisting of three layers, each one related to a model used to represent the input program. The first layer model is produced by inspecting the source code and is very similar to a UML class diagram. Next, it is transformed by adding information about particular features of classes and relationships in order to obtain the so called idiom layer model. The third layer, namely the design-level layer, consists of describing a model of a design pattern and transforming it in a constraint system. This will be used to look for possible design patterns into the previous layer model. Eventually a new model provided with information about design patterns is built. It is important to note that users are allowed to relax constraints during research. This is useful to detect design pattern versions which do not strictly fit the given model.

Similarly to other approaches, De Lucia et al. [21]. first step is the construction of an UML class diagram, that is performed by scanning the source code of the input application. Next, a visual language based on this model is defined, introducing a formalism that is similar to context free grammars. The actual design pattern search is accomplished by the means of a LR parsing process. The final operation is a new source code analysis aimed to reduce the number of false positives among the results (relationships are verified by specific algorithms).

Dong et al. [14] introduced an approach and a tool named DPMiner to recover design patterns based on a matrix. This approach uses a pre-built representation of the input software system that is the UML class diagram produced by IBM Rational Rose (the actual design pattern search is performed by firstly parsing the XMI files generated by a plug-in). A matrix is filled with a set of weights that represent structural information about systems and patterns, i.e. the relationships among classes are stored into the matrix elements, while weights are useful for attributes and operations. If a pattern matrix matches a system matrix, it means that the same relationships hold for the pattern and the system. If even the weights are the same, it is considered a pattern structure match. Additionally, a behavioral analysis is performed aiming at reducing the possible false positives that could have been found previously. In order to do this, each design pattern is formally defined using a set of predicates that provide information about methods (such as invocations, parameters or return types). Outcome of the structural analysis are then filtered according to such definitions. Eventually, a semantic analysis is performed in order to retrieve information about possible design patterns from the names of involved methods (hoping that developers have followed certain conventions) and refine the results in this way.

Compared with the previous approach, our solution retrieves a more detailed information, since UML modeling often generates rather abstract representations of software systems and it is strongly influenced by the designer will. Moreover, it is important to point out that the mere fact our tool does not rely on an external support is a big advantage in terms of use flexibility. Moreover, the previous approach relies on hard-coded design pattern structures, while our tool models are fully customizable. An important feature claimed by the authors of DPMiner is performance. Since a big time gap lies between the development of the last approach and ours, we have reported in Section IV the performances of our tool on a much more advanced hardware. However, for comparison, we measured the average execution time by running our tool on an

Intel Pentium with 1.4 GHz frequency and 256 MB RAM, with JHotDraw as input, and the output was given in about 2464 milliseconds. DPMiner was reported to run in about 24000 milliseconds by using a better hardware (Pentium processor with 3.4 GHz frequency and 1 GB RAM), hence we can state with reasonable confidence that performances of our tool are far better.

We can see that each approach mainly consists of three phases. First an analysis of the input software system is performed by inspecting the source code or the bytecode. Next, a structure representing the system is built: the most common choice is the adjacency matrix. The last phase is the search procedure, performed according to a known model of a design pattern. Most of the approaches do not look for a precise model of a design pattern. This choice is aimed to avoid that some design patterns are missed, since they are a variation of the model. On the contrary, our research algorithm is an exact graph matching, hence sub-graphs have to fully match the models to result as true positives.

We have adopted an exact match for the uncertainty that can derive from relaxing the rules determining the characteristics of a design pattern. Such relaxation could put at risk the fundamental structure and definition of a design pattern. Instead in our approach the developer is free to define own variations, as additional model, to be looked for in an application.

VI. Conclusions

This paper has proposed an approach to quickly find design patterns inside a software system. The devised algorithm consists of building trees by carefully positioning in them candidate classes, then check that all the characteristics of a design pattern model are matched into the tree. The outcomes have shown a very short processing time for large systems under analysis. The usefulness of the approach lies on the ability to automatically document a system and opens up the possibility to further improve the implementation of the analysed system. Future work aims at assessing the benefits of a massively parallel processor (e.g. as in [22]) for finding many variants of design patterns.

References

[1] A. Calvagna and E. Tramontana, "Delivering dependable reusable components by expressing and enforcing design decisions," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 493–498.

[2] E. Tramontana, "Detecting extra relationships for design patterns roles," in *Proceedings of AsianPlop*, Tokyo, Japan, March 2014.

[3] ——, "Automatically characterising components with concerns and reducing tangling," in *Proceedings of IEEE Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, Kyoto, Japan, July 2013, pp. 499–504.

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

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] J. Kerievsky, *Refactoring to patterns*. Addison-Wesley, 2005.

[7] C. Napoli, G. Pappalardo, and E. Tramontana, "Using modularity metrics to assist move method refactoring of large systems," in *Proceedings of IEEE ICLS workshop at CISIS*, Taichung, Taiwan, 2013.

[8] G. Pappalardo and E. Tramontana, "Suggesting extract class refactoring opportunities by measuring strength of method interactions," in *Proceedings of IEEE Asia Pacific Software Eng. Conference (APSEC)*, Bangkok, Thailand, December 2013, pp. 105–110.

[9] R. Giunta, G. Pappalardo, and E. Tramontana, "Using Aspects and Annotations to Separate Application Code from Design Patterns," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, March 2010.

[10] ——, "Aspects and annotations for controlling the roles application classes play for design patterns," in *Proceedings of IEEE Asia Pacific Software Engineering Conference (APSEC)*, Ho Chi Minh, Vietnam, December 2011, pp. 306–314.

[11] ——, "AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Riva del Garda, Italy, March 2012, pp. 1243–1250.

[12] ——, "Superimposing roles for design patterns into application classes by means of aspects," in *Proceedings of ACM Symposium on Applied Computing (SAC)*. Riva del Garda, Italy: ACM, March 2012, pp. 1866–1868.

[13] G. Pappalardo and E. Tramontana, "Automatically discovering design patterns and assessing concern separations for applications," in *Proceedings of ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006.

[14] J. Dong, Y. Zhao, and Y. Sun, "A matrix-based approach to recovering design patterns," *Trans. on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 39, no. 6, 2009.

[15] M. Fowler, *UML distilled*. Addison-Wesley Professional, 2004.

[16] P. Maes, "Concepts and experiments in computational reflection," in *Proc. OOPSLA*, vol. 22 (12). ACM, 1987.

[17] I. Forman and N. Forman, *Java Reflection in Action*. Manning Publications, 2005.

[18] S. Chiba, "Load-time Structural Reflection in Java," in *Proceedings of ECOOP*, ser. Lecture Notes in Computer Science, vol. 1850. Springer-Verlag, 2000.

[19] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 896–909, 2006.

[20] Y.-G. Gueheneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.

[21] 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.

[22] C. Napoli, G. Pappalardo, E. Tramontana, and G. Zappalà, "A cloud-distributed gpu architecture for pattern identification in segmented detectors big-data surveys," *Computer Journal*, 2014. [Online]. Available: http://dx.doi.org/10.1093/comjnl/bxu147