

# DESIGN PATTERN RECOGNITION

Francesca Arcelli, Claudia Raibulet, Francesco Tisato

Università degli Studi di Milano-Bicocca,  
Dipartimento di Informatica, Sistemistica e Comunicazione  
Via Bicocca degli Arcimboldi, 8, I-20126 – Milano, Italy  
[arcelli@disco.unimib.it](mailto:arcelli@disco.unimib.it)

## Abstract

*The role of design patterns in forward engineering is well known, also if it's not easy to use them, since large experience is needed. Different approaches and tools have been proposed in the literature to enhance and support the design process respect to design patterns exploitation. Also the role of design patterns in reverse engineering has been largely studied. In this paper, we describe a tool for design pattern recognition to be used in forward engineering as a kind of reuse advisor. As finding the right solution, or better finding the right pattern involves modeling some kind of similarity, we explore and propose some possible similarity-based techniques that could be used for design pattern recognition.*

## 1. INTRODUCTION

The study of software patterns concerns all the phases of the software development process, from requirement engineering to process management or software architecture. Usually, the term design pattern is used to refer to any pattern concerning software architecture: design patterns can be seen as micro-architectures involved in the overall architecture of the system.

One of the principle aims of design patterns is reuse of experience. Several definitions of patterns have been given in the literature ([7]), as “each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context and a certain software configuration which allows those forces to resolve themselves”, but in general a design pattern (DP) is a documented solution to a problem that has been proven to work well in a given design solution. It is important to identify where a pattern could be applied

and then using it. Developers facing a specific problem should be able to easily find a matching pattern that provides a solution from a library of available patterns, which have been already carefully classified.

The problem of finding a matching pattern is one of the most critical for design pattern recognition.

We consider here design patterns as defined in [7], where patterns are described uniformly through an informal language by means of informal text and graphic notations, usually given through UML diagrams.

Each pattern is presented using a consistent format with the following sections: *Pattern Name* and its classification, *Intent* of the pattern, *Also known as* (other names), *Motivation* (where it can be applied), *Structure* (graphical representation of the classes in the pattern and their interactions), *Applicability* (the context in which can be applied), *Participants* (in terms of classes and objects), *Collaborations* (between participants), *Consequences* (costs and benefits of applying the pattern), *Implementations*, *Sample Code*, *Known Uses*, *Related Patterns*.

In [7] patterns are classified in terms of purpose, which could be creational, structural, behavioural and in terms of scope, whether a pattern applies to classes or to objects. Otherwise if we consider the relationships existing between patterns, we can identify the following relationships, as outlined in [18]:

- ) a pattern A *use* another pattern B in its solution: one sub-problem of A is similar to the problem addressed by B;
- ) a variant of a pattern A uses the pattern B in its solution: some variant of A may use B, but it is optional;
- ) a pattern A *is similar* to a pattern B: the patterns A and B address similar kinds of problems.

All these considerations can be useful for classification and recognition purposes: to identify basic DP, DP for typical

software problems and DP for specific application domains.

We are particularly interested in addressing the pattern recognition problem, starting from the problem description, or better from the problem specification given through UML diagrams.

As stated in [3], “the retrieval of a design pattern involves the selection of a set of components, which is much more complicated than the traditional software component retrieval”.

We describe a tool that has been developed at our University ([8]), in which starting from an UML diagram and a catalogue of design patterns, the tool tries to discover if there is a pattern “similar” to the UML diagram, at which degree of similarity and for which aspects it needs to be adapted to the new context. The mapping is determined exploiting the notion of roles associated to design patterns.

It’s difficult to be able to apply an exact matching (obviously not only in this context): finding useful ways to model a fuzzy matching, that often consists in modelling some forms of similarity, become a critical problem. Hence we explore how similarity-based techniques can be used for design pattern recognition, in general both in forward and reverse engineering.

We are interested to explore how it’s possible to merge UML diagrams and design patterns following different similarity-based approaches involving fuzzy techniques to face in particular the problem of pattern recognition.

The paper is organized through the following Sections. In Section 2 we describe the tool Platone developed for design patterns recognition. In Section 3 we explore the relevance of similarity in design pattern recognition and finally we conclude and discuss some future developments.

## 2. DESIGN PATTERNS RECOGNITION

Several aspects can be taken into account in developing a tool for design pattern recognition: *a)* given a problem description find a set of patterns able to face and solve the specified problem; *b)* recognize a design pattern that is similar to a current problem, identify needed modifications and apply the modified design pattern to the new problem. To face problem *a)* perhaps only experience can help, while to face problem *b)* different approaches can be followed.

Retrieving the correct design pattern and if necessary customize the pattern to fit the new domain problem is not easy: the correct customization of a DP is very critical for a successful exploitation of the pattern.

We can follow the following steps in creating a new DP:

- 1) analysing the Problem

- 2) deriving more UML models
- 3) observing the similarities among them
- 4) abstract the common properties
- 5) creating a design pattern
- 6) using the design pattern

If we want to recognise a DP, step 3) consists in finding the similarities between the UML model which we have obtained and a given DP.

Our principal interest is directed to the integration of DP and UML by exploiting the different mechanisms available through UML, called General Extension Mechanisms, as stereotypes, tagged values and constraints.

### 2.1 The tool Platone

The supporting tool, exactly a decision support system, called *Platone*, developed at our University [8] has been realized to decide if given a UML class diagram and a DP, some similarities exist, also by doing interactively questions to the user, as depicted in Figure 1.

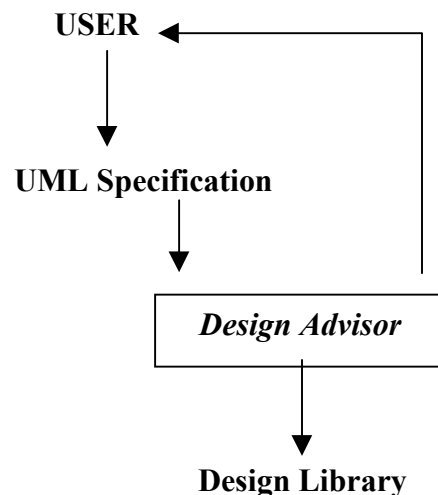


Figure 1. Platone as a Design Advisor

Two different users may interact with the tool Platone: the *UML-designer* which uses the patterns as a support instrument in the design phase and tries to find the more appropriate DP and the *Pattern-designer*, which writes the DP itself and adds it to the Design Library.

Several DP as described in [7] have been analysed (in particular the Singleton, Observer, Abstract Factory, Template Method, State, Chain of Responsibility) and proved through the tool.

The concept of *role* as introduced in [10] (different notion from the *role* as used sometimes for Associations in a class diagram) represents a key feature of Platone. The aim of the role is to assign to each participant in the description of

the DP a typed role. Then a link is created between the design pattern and the real classes through a mapping of a role to some UML element, and a fuzzy value in the interval [0,1] is assigned to this mapping function. For UML element we mean an entity as *Class*, *Interface*, *Method*, *Property* and *Association*. The definition of UML element is a subset of the definition of Element given in the UML specification provided by OMG. The roles may be of different types, as Abstract Class, RealClass, ClassesInstanceHolder, PartialMethod, MethodsGroup, Association, Aggregation, Client,.. . Sometimes it's required to know if some roles are static or not and this it is simply obtained adding a boolean flag "isStatic" (i.e. for the roles of type *Method* or *Property*).

For every pattern, we have two set of roles: *Fundamental* and *not-Fundamental*. The first ones are used to distinguish some particularly significant elements of the pattern. This choice is determined also according to the implementation features. Moreover a role may be mapped on more than one element.

A relation between roles, called *playsRole()*, has been introduced, giving rise to the *Roles&Relations Model* as a conceptual model of patterns, a model which is synthetic since it captures the essence of the pattern and general since it's able to include the greater number of DP "instances". Examples of these relations between roles are: *contained*, *called*, *used*, *declared*, *implemented*, *number of subclasses*, ...

The mapping function between a role and some UML elements is defined in this way:

$$\text{RoleX} \rightarrow u_1, u_2, \dots, u_n$$

where  $u_1, u_2, \dots, u_n$  are mapping elements (if  $n$  is greater than 1 we have multi-mapping roles). Then *Platone* gives an evaluation of this mapping function in the interval [0..1]. This is particularly useful, since a strength aspect of DP is related to their concentration on relevant aspects, not considering all the secondary details. For this reason a fuzzy evaluation seems to be more appropriate.

For the mapping, the function *playsRole* is used: with  $r.\text{playsRole}(u)$  we mean the evaluation through the *playsRole* function of the role  $r$  on the element  $u$ . The mapping succeed if and only if the relations described in a role are all satisfied by the corresponding UML elements (i.e.  $u_1, u_2, \dots, u_n$ ).

The heuristic engine defined in *Platone* takes as input a list of roles and an automatic assignment of the role to an element is performed, by evaluating the confidence of each UML element through the value obtained by the function *playsRole*. These confidences values are evaluated in a decreasing order and then a mapping is performed between the role and the element with the higher confidence value. At the end to guarantee the process termination, all the elements assigned to the role are eliminated from the set of assigned elements.

*Platone* compute two fuzzy values: the first is obtained through the simple average of the confidence values of each single role (ie. the average of the different *playsRole()* values). A second fuzzy value is computed through a *weighted sum*, where the non fundamental roles (those deduced by the tool) have a double weight respect to the fundamental ones chosen by the user.

When all the roles have been mapped, we obtain an evaluation of how the UML schema implements the DP.

Each role may assume different weights and the weighting function could be changed dynamically by the user giving higher values to the more relevant roles, on behalf of his judgement.

Hence, a DP Model (DPM) is characterized by the roles of some elements and by the more important relations existing between the other elements (other roles).

We observed that the relations between roles and the roles types are recurrent and in many cases identical in several DP. A DP can be seen in this way as a set of roles of a single element with the relations existing between these elements. We can evaluate in this way how an UML schema implements the DP.

The design patterns have been analysed in the order of difficulty expressed in terms of the number of roles involved and of the complexity of the relations in each model.

We now see an example of how a *Role&Relation Model* is obtained for a given DP, i.e. the Chain of Responsibility (pattern 225 in [7]), as given in Figure 2.

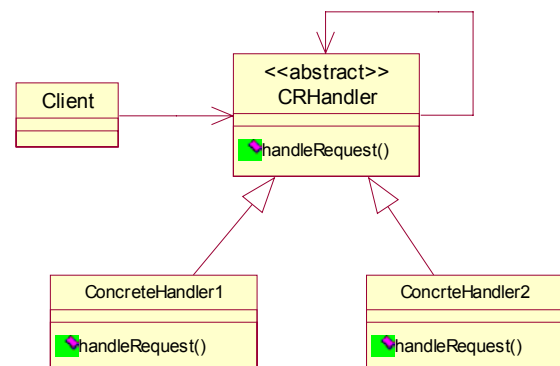


Figure 2. Chain of Responsibility

The following roles are identified:

*CRHandler*, *ConcreteHandler* and *Successor*.

For each of these role a table is produced with the following information:

- if the role is Fundamental
- if the role is mapped on more elements
- the type of the role
- the Relations
- the Semantics.
- the code of the function *playsRole* in Java.

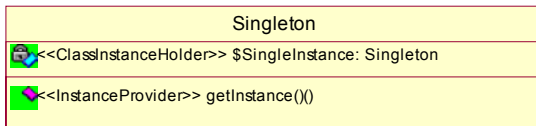
In the case of the DP in Figure 2, the role is Fundamental, it's not mapped on more elements, the type is AbstractClass, the Relations are according to the two subclasses and this Semantics: the role has a reference of name Successor, private<Noame>CRHandler successor, the role has at least an AbstractHandleRequest (i.e handleRequest).

The code of the function playsRole in Java is provided for the roles ConcreteHandler and Successor.

We now see a simple example on how *Platone* works.

*Platone* takes as input a class diagram or a subset of a class diagram selected by the user and gives as output two values about the mapping of the class diagram with one or more DP in the catalogue of DP together with a report of the details on the performed mapping.

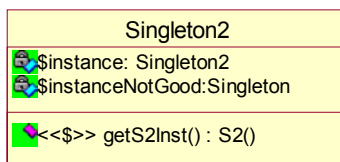
Consider for example the Singleton DP as given in [7]:



where the stereotype denotes the name of the role.

If the above diagram is introduced, *Platone* provides a degree of confidence equal to 1, that is the maximum similarity value.

If the following diagram has to be evaluated:



*Platone* provides a value of 0.8999 (through the weighted sum) and of 0.9166 (simple average).

The mapping role is:

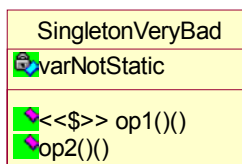
-Singleton <\*Fundament\* > -- [1.0] → Singleton2

-Instance Holder – [1.0] → instance

-Instance Provider – [0.75] → getS2Inst

We observe that *getS2Inst()* is a static method (\$) and *Platone* gives a high degree (0.75), since S2 could be also a list with several Singleton2 classes.

If the following diagram has to be evaluated:



*Platone* provides a value of 0.4200 (through the weighted sum) and of 0.5166 (simple average).

The mapping role is:

-Singleton<\*Fundament\* > -- [1.0] → SingletonVerybad

-Instance Holder – [0.4] → publicInstance

-!!Instance Provider – [0.1499] → op2

In this case both *op1()* and *op2()* don't return any value and *Platone* chooses *op2()* for the mapping of the instance provider with a very low degree (<15%), but for the heuristic engine, the choice between the two methods is indifferent. Moreover the system notices that also if the simple average is greater than 50%, the instance provider value is too low (!!) to declare that this class is similar to the Singleton DP.

The tool has been implemented in Java, supporting OLE and interfaced with Rational Rose, using the Rose Extendability Interface (REI).

### 3 SIMILARITY FOR DESIGN PATTERN RECOGNITION

As we saw before, the principal aim of a design pattern is to solve similar problems, which turn out in several and different projects; by observing the similarities, it is possible to abstract the central concepts, which characterize the different problems, it's possible to create a new DP or find the best model to solve a given problem. Hence, we observe that similarity plays a central role in this context, as in many other fields of software engineering. Similarity-based techniques have been used as a paradigm to support all the phases of the software development process, in particular in relation to the complex task of "reuse".

The application of computational intelligence techniques in software engineering has significant potential (as outlined in [14]), not yet completely discovered and exploited and we consider "similarity" central in computational intelligence, since processing data allowing tolerance for imprecision, uncertainty and partial truth are relevant aspects of both similarity and computational intelligence.

In the literature, similarity has been modelled through different techniques and used for different aims; however the problem of capturing, formalizing similarity in a suitable way, and computing and exploiting the similarities discovered has not yet been solved with success. Both fuzzy logic and case-based reasoning techniques provide a number of ways to deal with similarity notions and measures, as deeply analysed in [5]. In several cases ([13]) similarity is measured through a distance measure and many definitions of distance can be used: conceptual distance, Manhattan distance, Euclidean distance or other

similarities measures as Cosine measure (contextual relevance) and frequency measure, where usually both symmetry and reflexivity properties and some kind of transitivity property hold. While in a huge number of cases similarity is simply captured through fuzzy weighting techniques or fuzzy relations.

Several approaches based on similarity can be followed for design pattern recognition: DP can be recognized according to the similarity captured respect to some given features, according to the context or better respect to the problem domains.

If we consider the classification of DP based on relationships [18] as briefly outlined in Section 1, we can observe how it is important to discover similarity between DP in relation to the similarity in the problems addressed by the DP. From a different point of view, suppose to have a catalogue of DP, where each pattern is characterized by a set of properties (a design pattern descriptor) and we have to find a DP able to solve a problem. Perhaps we are able to find a DP satisfying one requirement, but not all of them. Some attributes are more relevant than other and so we can assign to them higher relevance. We can consider the features of DP, (as outlined in [7]), taking into account only those not involving graphical notations, as Intent of the pattern, Motivation,...and then we can define and use several similarity fuzzy relations [1] and combine these similarities according to a suitable t-norm to gain the best result able to model our requirements.

Another approach is the following: consider for example to have different objects as geometrical figures of different format, size-area, colour,...and we have to search for figures with the same area, or with the “more similar” area, or search for figure of a given colour. In this scenario, where we have to retrieve two objects according to similarity on one particular feature, we could again exploit a fuzzy similarity-based approach to combine the different similarities involved between the objects, which are in our case the design patterns descriptors. We may consider the more relevant attributes of the DP and the similarity could be computed through the inferential engine of the similarity-based logic programming language Likelog [2], which uses fuzzy similarity relations and fuzzy matching.

Finally, we could also follow the approach proposed in [4] for classification of reusable modules. DP can be described for example by the function they performs (their intent), the way they perform it and their implementation details (What, Where and How). These descriptors fall naturally into facets which can be ordered by their relevance. Thus a DP descriptor can be seen as a t-uple of terms where each term is an attribute value of a selected facet. Then the conceptual closeness between terms in a facet can be measured through a weighted conceptual graph, where

leaves in the graph are terms and nodes are super-types that denote general concepts relating to two or more terms.

A design pattern provides “a common solution to a common problem in a given context”. Also the notion of context ([17]) is strictly connected to similarity and we can use several t-norms in order to model different contextual-based knowledge [1].

Incorporating approximate pattern matching techniques can be useful also in tools for design pattern recognition in reverse engineering, as outlined for example in [9]. Several tools for patterns-based design recovery model similarity in different ways. For example, in [11] fuzzy weights are used in a rule to define thresholds for accepting only some sub-patterns instances. While in [16] flexibility is gained, not through fuzzy techniques, but through  $\rho$ -calculus used to formally state the relationships embodied in design patterns, without locking them into any one particular implementation.

#### 4. DISCUSSION AND FUTURE DEVELOPMENTS

In this paper we described the tool *Platone* and we gave some simple examples on the different possibilities in using it. Through *Platone* we provide a framework to assist design engineers in choosing and evaluate design alternatives. Obviously the user of *Platone* need some knowledge and experience on DP, but the tool can be in any case useful since it's well known from the literature that DP are very powerful, but it's not easy to use and apply them properly. The UML designer takes advantages, during the design phase, in finding the right DP for his specifications and he can evaluate between more alternatives. On the side of the DP creator, it's possible to evaluate if a DP similar to the one we want to create already exists.

A lot of future works and extensions are needed as using more sophisticated evaluation methods and considering also the problem to recognise not a single DP, but the combination of two or more DP.

The introduction of our ongoing research directed to explore how similarity can be used to recognize and classify design patterns is particularly interesting and opens many new developments. A large amount of knowledge and problems exist that suggest to deeply explore similarity-based reasoning and techniques for many software engineering tasks.

Exploiting similarity for design patterns classification and retrieval can be considered as a kind of case-based reasoning where cases are design patterns, and solutions to current problems are derived from the solutions of prior cases. Hence we would like to better explore the different techniques proposed for case-based reasoning, as the ones

in [5, 13] and define a kind of design paradigm, by exploring how similarity can be absorbed into existing approaches of software design.

Another direction in which we are particularly interested is to analyse the possible integration of tools for design patterns recognition in forward engineering, as Platone, with those for reverse engineering, always paying attention to the different techniques which can be used to model the “matching” problem.

To represent DP, often there is the need of abstraction expressions to account for the generalisations. In [5] it's outlined that notations like UML/OMT are not adequate to represent DP. We would like to explore if the new definition of the UML2 language could be more useful for this purpose.

**Acknowledgement.** This research has been partially supported by the MAIS research project financed by MIUR – “Ministero dell’Istruzione, dell’Università e della Ricerca” in the context of the FIRB program – “Fondo per gli Investimenti della Ricerca di Base”.

## 5. REFERENCES

- [1] F.Arcelli and F.Formato, User Adaptive Models based on Similarity. *Proc. of Symposium on Applied Computing*, SAC 2000, ACM Press, Como, Italy, March 2000.
- [2] F.Arcelli, Likelog: for flexible query answering. To appear in *Soft Computing Journal*, Springer Verlag, 2002.
- [3] W.Chu, C.Lu, C.Chang and Y.Chung, Pattern-based Software Re-engineering. *Handbook of software engineering and knowledge engineering*, 2001.
- [4] P.Diaz, Classification of reusable modules. *IEEE Software* 4(1), pp.6-16, 1987.
- [5] G.Finnie and Z.Sun, Similarity and Metrics in Case-based Reasoning. *International Journal of Intelligent Systems*, vol.17, 2002, pp.273-287.
- [6] M.Fowler. *Analysis Patterns: reusable object models*. Addison Wesley, 1997.
- [7] Gamma, Helm, Johnson and Vlissides. *The Design Patterns*, Addison Wesley, 1994.
- [8] G.Giorgi, Towards new object-oriented design methodologies: integrating UML and design patterns. Thesis, Dep. Computer Science, University of Milano, 1999.
- [9] G.Y.Guo, J.Atlee, R.Kazman. A software architecture reconstruction method. *Proceedings of WISCAI*, track on techniques on techniques and methods for software architecture, 1999.
- [10] M.Meijers, *Tool Support for Object-oriented Design Pattern*, Master Thesis, Utrecht University, CS Dept., INF-SCR-96-98, 1996.
- [11] Niere J., Wilhelm S., Wadsack J.P., Wendehals L., Welsh J. *Improving Design Pattern Instance Recognition by Dynamic Analysis*, Wendehals L., University of Paderborn, 2003.
- [12] H.Osborne and D.G.Bridge, Similarity metrics: a formal unification of cardinal & non cardinal similarity measures. *Case-based Reasoning Research and Development*, Ed D.B.Leake, E.Plaza, Springer, 1997.
- [13] H.Osborne and D.Bridge, Models of Similarity for Case-based Reasoning. *Proceedings of the Interdisciplinary Workshop on Similarity and Categorisation*, 1997.
- [14] W.Pedrycz and J.F.Peters, *Computational Intelligence in Software Engineering*. Advanced in Fuzzy Systems-Applications and Theory, vol.16, World Scientific, 2001.
- [15] D.J.Ram et al., An approach for pattern oriented software development based on a design handbook. *Annals of Software Engineering*, vol.10, 2000, pp.329-358.
- [16] J.McSmith, D.Stotts. SPQR: Flexible automated design pattern extraction from source code. Tech.Report, TR03-016, University of North Carolina, May 2003.
- [17] M.Theodorakis, A.Analyti, P.Constantopoulos, N.Spyratos, A theory of contexts in information bases. *Information Systems*, vol.27, n.3, 2002, pp.151-191.
- [18] W.Zimmer, Relationships between design patterns. *Pattern Languages of Program Design*, Ed.J.O.Coplien and D.C. Schmidt, Addison Wesley, 1995.