Heuristics for Automatic Detection of Design Patterns in Object-Oriented Software

André Luis Castro de Freitas [1]
Ana Maria de Alencar Price [2]

[1] Centro Federal de Educação Tecnológica de Pelotas & Universidade Católica de Pelotas
Pelotas - RS
afreitas@atlas.ucpel.tche.br
[2] Universidade Federal do Rio Grande do Sul
Porto Alegre - RS
anaprice@inf.ufrgs.br

Abstract

Design patterns summarize the experience of expert designers. Patterns are not invented, rather they are extracted from existing systems. The extracting process of patterns had involved the observation of a number of systems designs, looking for patterns across those systems. A design pattern is a reusable implementation model or architecture that can be applied to solve a particular recurring class of problem. In general, it is hard to recognize a pattern use in real-world software systems, unless you know what you are looking for and go carefully and methodically searching for a particular pattern. This paper is about the problem of detecting the essence of design patterns. By pattern essence we mean those collaborations between classes that characterize each one of the patterns. Heuristics have been created to identify and apply design patterns to object-oriented programs. The rules are based on the structural realtionship between classes and objects. It is implemented a tool in Smalltalk that automatizes detection and identification of design patterns in object-oriented applications. The tool intendes to be a validation prototype for the buit in heuristics. The development of the examples allows patterns comparison, showing advantages and tendencies in using one or another type of collaboration between classes and objects. Patterns studies stimulate facilities during the building of object-oriented programs. The patterns also help in the definition of good designs. We understand that a good design is a software that follows the fundamental concepts of the object-oriented paradigm including those rules stated by recognized.

Keywords: Software Engineering, Object-Oriented Design, Design Patterns, Smalltalk.

1 Introduction

Methodologies for developing object-oriented software and programming languages has evolved during last years. Both of them aim to reduce the problems of software maintenance. However these methodologies fail in providing suitable support for understanding the addressed systems. According to [ARA 93] in general, the methodologies for object-oriented software development fail in providing suitable formal concepts to reflect the mapping between project and implementation.

It is necessary to consider that the development of good quality object-oriented software is not a task that can easily be accomplished by inexperienced designers on object-oriented technology. In general, the design of object-oriented systems is definitively more complex of being accomplished than the design of systems that are based on the procedural technology. The reason for that complexity mainly comes from the high interaction among the objects created during run time.

Possibly the best idea brought from the object-oriented design community in an attempt to solve these problems, has been the notion of design patterns. Design patterns have the potential to represent the common abstractions that experienced object-oriented designers use to solve design basic problems in a way that can be understood and utilized by a novice designer.

Formally codifying these solutions and their relationships lets us successfully capture the body of knowledge which defines our understanding of good architectures that meet the needs of their users [APP 00].

1.1 Motivation

The development of good quality software is a constant concern among researchers of the software engineering area. According to Agarwal [AGA 95], software production is considered a recent technology in comparison to other industrial activities. Software maintenance still consumes a lot of effort and it presents a high final cost.

The model defined in [AGA 95] is a help for the understanding of the process of software development guided to objects. It evaluates the maintenance efforts spend in the system and it manages an optimizing process. The work presents techniques for the defining the system structures (entities and relationships) during the stages of software development.

The proposal described in [JIA 96] is characterized by the capture and dynamic simulation of relationships among objects. The work mentions a series of steps for systems modeling. These steps are not sequential, and there is a great iteration and relationship among them. That work shows a formal representation that is mentioned to characterize knowledge domain, which turns more practical and necessary the transformation processes during the modelling phase.

More specific works in the area of design patterns are also in process. Seemann [SEE 98] shows how to recover design information from Java source code. The tool takes a pattern-based approach and proceeds in a step by step manner deriving several layers of increasing abstraction. A compiler collects information about inheritance hierarchies and method call relations. It also looks for particular source text patterns coming from naming conventions or programming guidelines. The author build up two inheritance structures for classes and interfaces and their interrelation. The work uses the term references that means a class has an attribute of the type of the second class. This relationship may be either an association or aggregation.

Bansiya [BAN 98] proposes a tool that automates detection, identification and classification of design patterns in C++ programs. The tool can identify most structural and some behavioral patterns. Object-oriented languages such as C++ enphasize the structural relationship between classes and objects and provide such powerful capabilities as polymorphism, which lets child classes override the parent class while using and manipulating objects of the child classes as objects of the parent-class types. The proposal consists of a series of pattern algorithms illustrated through a case study of a drawing tool kit.

The use of tools, either automatic or manual, by designers during the development of object-oriented software design is of great importance to help to reduce the inherent complexity of the understanding process. These tools aid the designer in the construction of models, through analysis mechanisms, exploration and visualization of the information in different abstraction levels.

1.2 Objectives

Reverse engineering intends to extract the specification of complex software systems. A definition of what reverse engineering is about in [CHI 90] is: “Reverse engineering is the process of analyzing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form at higher levels of abstraction”.

For maintenance, reuse, and reimplementation, designers frequently need to examine source code to understand the software system. The ability to learn and understand software system from source code is greatly enhanced by visualizing the systems at higher levels of abstraction, rather than seeing them nebulous collections of classes and methods implementation. [BAN 98]

Visualizing object-oriented programs as a system of interacting patterns requires detecting, indentifyng and classifying groups of related classes in program code. These visualizations represent known patterns that perform an abstract task and are not necessarily known pattern solution. Aiming to support the development process of object-oriented software, this work proposes identifying design patterns essence. Heuristics have been created for identifying and applying design pattern to object-oriented programs. These heuristics are applied by a tool implemented in Smalltalk that automates identification of design patterns in object-oriented applications.

This paper is organized as follows: Section 2 presents characteristics of Design Patterns and class collaborations. Section 3 shows an automatic identification tool for design patterns. The tool aggregates the characteristics described in Section 2. In Section 4 there is the conclusion enfhasizing the paper contribution.

2 *Design Patterns*

A design pattern is a reusable implementation model or archicture that can be applied to solve a particular recurring class of problem. The pattern describe how methods in a single class or sub hierarchy of classes work together. More often, it shows how multiple classes and their instances collaborate [GAM 94].

2.1 Relationships Between Objects

This approach focuses on the ability of identifying structural and functional key relationships between classes and objects. The structural relationships of interest for pattern identification are inheritance, aggregation and use. While inheritance is easy to identify, aggregation and uses can have several forms of implementation. Typically, in Smalltalk, the aggregation and uses relationships are detected by looking for attribute declarations in classes only at runtime, when the objects are being created and manipulated [ALP 98].

Among the basic strategies for using the association mechanism there is a need of defining an attribute in each class that can store or make reference to all the others related objects. In order to keep these attributes updated there must exist methods of addition and removal of objects.

2.2 Heuristics for Automatic Detection

The proposed heuristics are expressed by logic clauses with the intention of presenting them in a formal way their use in a tool of design specification. There is a data dictionary which stores information regarding the classes involved in the application. The dictionary was built in the form of tables and the access to information is done by relational calculus [ELM 89].

Relational calculus is a non procedural language which allows a group of tuples to be defined as expressions of the type {t | P(t)} meaning that for each tuple t the predicate P(t) is true: t ε Result → P(t). A variable tuple (VT) represents to each instant a tuple T of a certain relationship R. The formula P(t) can present more than a variable.

For example, the main method below represents a fragment of the identified structure for the Composite pattern in a design. The Composite pattern allows you to build complex objects by recursively composing similar objects in a treelike manner. The pattern also allows the objects in the tree to be manipulated in a consistent manner, by requiring all the objects in the tree to have a common superclass or interface [JON 99].

```
VerifyRelationshipComposite
level = 0
level_indicator = nil
relationship_indicator = nil

∀ Objetos_Em_Teste ∈ Dictionary
{ objetoTeste | objetoTeste ∈ Objetos_Em_Teste }
        { objetoVerificado | ∃ objeto ∈ Objetos (
                    objeto[Nome_Objeto] =
                     objetoTeste[Nome_Objeto]
                    objetoVerificado = objeto ) }
            VerifyHierarchy: objetoVerificado
                                |
            IF level > 1
              IF all (level_indicator of same attribute =
                                        Verdadeiro)
                IF all (relationship_indicator of same attribute
                                        > 1)
                          IndicateComposite
```

```
VerificaHierarchy: objetoVerificado

level = level + 1
∀ Associações ∈ Dictionary
{ objetoAssociação | ∃ associação ∈ Associações (
            associação[Nome_Objeto] = nomeObjeto[Nome_Objeto]
            objetoAssociação = associação) }

            // Call objects with relationship
                ...
```

3 The Tool

This section presents a tool which automatizes the detection, identification and classification of design patterns in Smalltalk programs. The tool identifies some structural patterns by using the proposed heuristics. The present stage of the tool shows how to get the characteristics of an object related to its relationships and functions. This object can be

visualized and on it can be identified some patterns such as Composite, Decorator and Observer when they are found in the relationships of the object being analysed. We also want to adapt the study of other patterns to the tool.

In Figure 1, the tool presents the description of the object which is being analysed in its main window. The class attributes are visualized in the same screen. The user can select which attribute he wants to evaluate. The respective content will be presented for each attribute.

```
Example application
| listaDinheiroVivo listaMonetario listaBens |
listaDinheiroVivo := AtivoComposite new.
listaDinheiroVivo adicionaAtivos: (Garantia tipo: 'reais' valor: 5000.00).
listaDinheiroVivo adicionaAtivos: (Garantia tipo: 'dolares' valor: 10000.00).
listaMonetario := AtivoComposite new.
listaMonetario adicionaAtivos: (Garantia tipo: 'aplicacoes' valor: 5000.00).
listaMonetario adicionaAtivos: (Garantia tipo: 'conta corrente' valor: 1200.00).
listaMonetario adicionaAtivos: (Garantia tipo: 'poupanca' valor: 2300.00).
listaMonetario adicionaAtivos: (listaDinheiroVivo).
listaBens := AtivoComposite new.
listaBens adicionaAtivos: (Garantia tipo: 'imoveis' valor: 150000.00).
listaBens adicionaAtivos: (Garantia tipo: 'se-moventes' valor: 45000.00).
listaBens adicionaAtivos: (listaMonetario).
listaBens teste.
^listaBens valor
```

Object sent to tool

FIGURE 1 - **First Comunication Window**

The *Visualiza* option allows to check the structure of the objects involved in the application as well as the class diagram. Only the objects which are marked as a reference in their description will be visualized in the structure. The window bellow shows the classes diagram that represents an application being analised. If a diagram follows the any predefined heuristics a pattern indication will be done. Figure 2 shows the Composite pattern where a hypothetical object is related to objects by *Garantia* or *AtivoComposite*. Therefore the realtionships happens to the *Ativo* superclass.

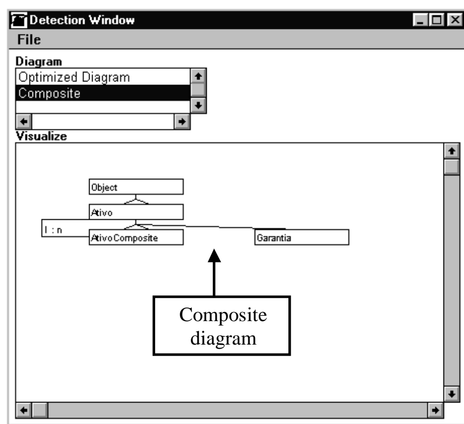

FIGURE 2 - **Composite Pattern Diagram**

4 Conclusion

The goal of this work is to give examples of how to use mechanisms for detection design patterns. Heuristics have been defined for design patterns to help in the construction of new projects because they are suitable to direct the development of activities which are based

on designers personal experience. However to use design patterns does not lead us to obtain definite answers to the problems at issue. On the other hand it has established some ideas to optimize the construction of object-oriented software.

The tool is still being built but it has already has implemented the identification of some patterns from a reported code. We believe that the logical consistency validation proposed by the tool is being to increased gradually with new tests. The more case studies are done for patterns of real projects the bigger the samplings will be to certify the tool.

This study is a step that represents an effort of researching towards the creation of techniques for design optimization. It is still necessary to compare this study with those tools mentioned in the section 1.1. Those tools present common objectives and therefore their results will come to contribute significantly to this research. Other aspects need to be studied such as investigating changes at the system structure level when it is necessary to include a pattern and to check if a pattern insertion is really needed.

4.1 Future Research

The approach to identifying design patterns relies on reducing known patterns to the minimum necessary and identifiable structures required by the design solution. However, an approach based solely on pattern structures is not complete because they are not suffciently unique. Several patterns tend to use similar basic structures. Thus, to reduce erroneus identifications its necessary to extend the approach by looking for design heuristics and by using empirical data for resolving the presence of patterns and pattern-like solutions. The heuristics and empirical data will be extract from design and implementation metrics, that evaluate the structure and functional characteristics of classes and relationships.

References

[AGA 95] AGARWAL R.; LAGO, P. PATHOS - A Paradigmatic Approach to High-level Object-oriented Sofftware Development. **ACM SIGSOFT Software Engineering Notes**, New York, v.20, n. 2, p.36-41, Apr. 1995.

[ALP 98] ALPERT, S. et al. **The Design Patterns - Smalltalk Companion**. Reading: Addison-Wesley, 1998.

[APP 00] APPLETON, B. **Patterns and Software: Essential Concepts and Terminology** Disponível por WWW em http://www.enteract.com/~bradapp/ (jun 2000).

[ARA 93] ARANGO, G. et al. A Process for Consolidating and Reusing Design Knowledge. In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 13., 1993, Baltimore. **Proceeding...** California: IEEE Press, 1993.

[BAN 98] BANSIYA, J. Automating Design-Pattern Identification. **Dr.Dobb's Journal.** New York, v.23, n. 6, p.20-26, Jun. 1998.

[CHI 90] CHIKOFSKY, E.; CROSS, J. Reverse Engineering and Design Recovery: A Taxonomy. **IEEE Software**, New York, v. 7, n.1, p. 13-17, 1990.

[ELM 89] ELMASRI, N; NAVATHE, S.B. **Fundamentals of Database Systems.** Redwood City: Benjamin Cummings, 1989.

[GAM 94] GAMMA, E. et al. **Design Patterns:** Reusable Elements of Object Oriented Design. Reading: Addison-Wesley, 1994.

[JIA 96] JIAZHONG, Z.; ZHIJIAN W. NDHORM: An OO Approach to Requirements Modeling. **ACM SIGSOFT Software Engineering Notes**, New York, v.21, n. 5, p.65-69, Sept. 1996.

[JOH 99] JOHNSON, R. **Design Patterns.** Departament of Computer Science at University of Illinois in Urbana Champaign. Disponível por WWW em http://st-www.cs.uiuc.edu/cgi-bin/wikic/wikic?DesignPatterns (dez. 1999).

[SEE 98] SEEMANN, R.; WOLFF JÜRGEN. Pattern-Based Design Recovery of Java Software. In: SYMPOSIUM ON FOUNDATIONS OF SOFTWARE ENGINEERING, 6., 1998, Orlando, FL USA. **Proceedings...** New York: ACM Press, 1998.