This volume contains the papers presented at the 7th International Conference on Object Oriented Information Systems – OOIS 2001. The conference was hosted by the University of Calgary, Calgary, Canada on 27 – 29 August 2001.

The theme of OOIS'01 was Object-Oriented and Web-Based Frameworks for Information Systems. The papers published in this volume highlight the contributions of leading researchers and practitioners in the field of Object Technology and Information Systems. The topics covered include: OO foundations, OO modeling and analysis, OOIS processes, XML-based IS, OO-based reuse, OO frameworks, OO and web testing, Use case for requirement analysis, OO CASE tools, OO virtual environments and real-time systems, IT process assessment and improvement, Industrial experience and case studies, Web-based IS, Component-based OOIS, Software engineering metrics and analysis, Production line and requirements engineering, GRIDs: the next generation technologies for the Internet, E-Business Enterprise Frameworks, and Perspectives on future development.

ISBN 1-85233-546-7

www.springer-ny.com
www.springer.co.uk
www.springer.de

ISBN 1-85233-546-7
9 781852 335465

http://www.springer.co.uk

Wang • Patel • Johnston

OOIS 2001

Y. Wang, S. Patel, and R.H. Johnston (Eds)

2001 OOIS

7th International Conference on Object Oriented Information Systems

Springer

OOIS 2001 Programme Committee

General Chair

Ronald Johnston (Canada)

Program Committee Co-Chairs

Yingxu Wang (Canada)
Shushma Patel (UK)

Programme Committee

Motoei Azuma (Japan)
Franck Barbier (France)
Zohra Bellahsene (France)
Francesco Capozza (Italy)
Elizabeth Chang (Australia)
Islam Choudhury (UK)
Alessandro d'Atri (Italy)
Sergio de Cesare (UK)
Stefano de Panfilis (Italy)
Tharam Dillon (Hong Kong)
Armin Eberlein (Canada)
Mohamed Fayad (USA)
Ian Graham (UK)
Jane Grimson (Ireland)
Rachel Harrison (UK) .
Brian Henderson-Sellers (Australia)
Mou Hu (Canada)
Keith Jeffrey (UK)
Ronald Johnston (Canada)
Graham King (UK)
Mark Lycett (UK)
Frank Maurer (Canada)
Greg McAvoy (Canada)
Martin Mintchev (Canada)
Alan O'Callaghan (UK)
Maria Orlowska (Australia)
Dilip Patel (UK)
Shushma Patel (UK)
D Janaki Ram (India)
Colette Rolland (France)
Guenther Ruhe (Germany)
Mike Smith (Canada)
Giancarlo Succi (Canada)
Yuan Sun (UK)
Jeff Sutherland (USA)
Yingxu Wang (Canada)
Alan Wills (UK)
Roberto Zicari (Germany)

Organising Chair

Frank Maurer (Canada)

Organizing committee: Camille Sinanan, Steven Leikeim, Darcy Murray

Table of Contents

Session 2.B ASERC ASE (I) – Software Engineering Metrics and Analysis

Session 3.A XML-Based Information Systems

Session 3.B OO-Based Reuse

Session 4.A OO Frameworks

Session 4.B ASERC ASE (II) – Production Line and Requirements Engineering

Session 5.A OO and Web Testing

Session 5.B Use Case for Requirement Analysis

Session 6.A OO CASE Tools

Session 6.B OO Virtual Environments and Real-Time Systems

Session 7.A IT Process Assessment and Improvement

Session 7.B Industrial Experience and Case Studies

Session 8.A Web-Based Information Systems

Session 8.B Component-Based OOIS

Principles and Patterns in the Object Oriented Design

JAVIER GARZÁS
ALTRAN SDB Consultant
Projects Engineering Research Group
ALTRAN SDB
C/ Ramírez de Arellano, 15. 28043,
Madrid - SPAIN
jgarzas@altransdb.com

MARIO PIATTINI
Alarcos Research Group
Escuela Superior de Informática,
University of Castilla-La Mancha
Ronda de Calatrava, s/n. 13071,
Ciudad Real - SPAIN
mpiattini@inf-cr.uclm.es

Abstract

Nowadays, due to experience acquired during years of investigation and development of Object Oriented systems, numerous techniques and methods that facilitate their design are available to us. In this article we present a compilation of the object oriented design principles, as well as an initial analysis of the design patterns and their relationship with these principles and as this relationship can facilitate a new base for the study, comparison and application of patterns. The principles allow us to extract "good practical" OO and to observe how the patterns are based and connected with the design principles, so we will be able to learn how to apply them.

1 Introduction

Since Simula 67 up until the present day knowledge related to the construction of Object Oriented (OO) systems has evolved significantly. Nowadays, due to experience acquired during years of investigation and development of OO systems, numerous techniques and methods that facilitate their design are available to us.

However, serious difficulties are still encountered when we tackling the construction of OO systems, especially in the transition between the analysis processes and the OO design, aspect this very vague in this type of paradigm [1]. In practice designers have accumulated a body of knowledge that they apply during these processes. Up until a few years ago this knowledge was totally implicit but fortunately it is now being specified and popularized in different forms: principles, heuristics, patterns and more recently, refactoring techniques. The difference between these concepts is generally unclear and moreover not all of them have received the same amount of attention or have reached the same degree of maturity. In fact, with the exception of the contributions of Liskov [2], Meyer [3] and Martin [4] [5], a strong knowledge does not exist on design principles, being used these in an isolated way or even

being ignored*. Regarding OO design heuristics the only works to which we can refer are those of Riel [6] and Booch [7]. Patterns, however, are without doubt one of the elements that have undergone the greatest evolution and proof of this is the existence of numerous publications on the theme. The application of patterns in OO began at the beginning of this decade [8] and was consolidated by the work of Gamma *et al.* [9], Buschmann *et al.* [10], Fowler [11] and Rising [12]. Amongst the different types of patterns we can distinguish, mainly, although other categories exist (Antipatterns, specific domains, etc.):

- Architectural: these focus on the structure of the system, the definition of subsystems, their responsibilities and rules.
- Object Oriented Analysis/Design (OOAD): to support the refining of the subsystems and components as well as the relationships between them.
- Idioms: they help us to implement particular aspects of the design in a specific programming language.

As we already know, the use of patterns means that we can avoid constant reinvention, thus reducing costs and saving time. Gamma [9] points out that *"one thing that expert designers do not do is resolve each problem from the beginning [...]. When they find a good solution they use it over and over again. This experience is what makes them experts."*

In the figure 1 we can see the relationship between the previous concepts and their quality goal, according to our interpretation.

Lastly, refactoring techniques are characterized by their immaturity, although it is true to say that this topic is rapidly gaining acceptance, largely thanks to Fowler's work [13].

The problem confronting the designer is how to articulate all this explicit knowledge and to apply it in an orderly and efficient fashion in the OODA, in such a way that it is really of use to him. In fact, in practice, even such advanced subjects like patterns have this problem. Ralph Johnson comments in this sense that *"for one thing, the large number of patterns that have been discovered so far need to be organized. Many of them are competitors; we need to experiment and find which are best to use [...]. Analyzing existing patterns, or making tools that use patterns, or determining the effectiveness of patterns, could all be good topics"* [14], this situation could give rise to incorrect applications of the patterns [15].

In this paper we are going to make an in-depth analysis of the relationships between principles and patterns, as we believe that principles can be useful when

* Books such as that of James Martin (not to be confused with Robert C. Martin, previously cited) titled "Principles of Object Oriented Analysis and Design" do not in reality analyse construction principles of OO systems but rather present concepts, techniques and methods for OO analysis and design.

systematizing the application of patterns in OODA. We will look at heuristics and refactoring techniques on another occasion. In the following section we will sum up the principles that we have been able to gather from the already existing bibliography. In section 3 we analyze the relationships between principles and patterns and we classify the patterns according to the principle to which they best adjust. In section 4 we present our conclusions and future projects.

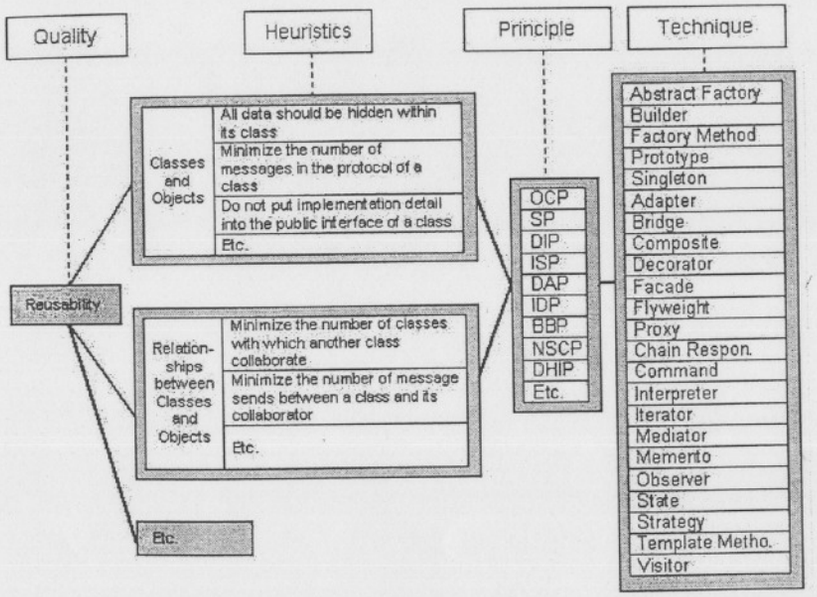

Figure 1. Heuristic, Principle, Pattern and the Reusability

2 OOD Principles

An OOD principle can be defined as a set of proposals or truths based on experience that form the foundation of OOD and whose purpose is to control this process.

This section aims to bring together some of the main object oriented design principles both at Inter-Modular and Intra-Modular design level. Although other principles apart from those described here may exist, we are limited by the length of this paper in the number we can describe. However, those mentioned here represent a good sample and are moreover the most outstanding.

2.1 Open-Closed Principle (OCP)

Statement: *A module should be open for its extension and closed for its modification.*

This is perhaps the most important and well-known principle, originating from Meyer's work [3] and meaning simply that: we should write modules that can be extended in such a way that they do not require modification. In order to do this we usually differentiate between interfaces and implementations without any problem. Both should been separated so the client only depends on the interface being able to change the implementation of the interface without problems. Several strategies exist to complete the OCP, although the use of Abstract Interfaces, is the most versatile and efficient way. An abstract supplier defines an interface for the supplier class and the client only depends on an interface that will rarely change.

2.2 Substitution Principle (SP)

Statement: *The subclasses must be substitutable by their base classes.*

The principle was proposed by Liskov [2] in her work on data abstraction and types theory. It also comes from the concept of Design by Contract of Meyer. This principle states that generalization between two classes can only be used in situations where instances of the superclass can be freely substituted by the subclass and where that substitution cannot be detected by the client module.

2.3 Dependency Inversion Principle (DIP)

Statement: *Depend upon abstractions. Do not depend upon specifications.*

The first studies referring to this principle appear in [4]. If the OCP principle provides an objective for software architecture the DIP establishes an initial mechanism for achieving this objective. DIP is the strategy of depending on abstract interfaces, functions or classes rather than on specific elements, essence of component models (CORBA, EJB, etc.).

The principle implies that each dependency in the design should have an interface or abstract class as its objective. Dependencies should not have specific classes as objectives. Such a restriction may be excessive if generalized but in fact it is more reasonable as specific things change a lot whilst abstract things change less. The abstractions are "hinges", they represent the places where design can bend or extend without being modified, which is related to the OCP.

2.4 Interface Segregation Principle (ISP)

Statement: *Many client specific interfaces are better than one general purpose interface.*

The main studies relating to this principle belong to Meyer [3] and Martin [5]. The essence of the principle is simple: in the case of a class with several clients, to create specific interfaces for each client and multiple inheritance from them regarding the class. If this principle is followed, the methods that each client needs are located in specific interfaces for this client.

2.5 Default Abstraction Principle (DAP)

We have found no reference to this principle in existing literature but we consider it to be interesting. Types or interfaces can in many cases belong to the domain of the problem. In other cases, for example when it is necessary to interconnect the domain with subsystems, service interfaces can appear (for example the subject interface in an Observer pattern). In this case what a domain class must implement the operations defined by the interface is not totally clear. This is usually solved by introducing between the interface and the class that implements it, an abstract class that makes the implementation in default of most of the interface operations. On the other hand, if we do not put in the abstract class that makes the implementation by default, we would repeat the code in several possible children of the parent interface. We could consider eliminating the interface and leaving only the abstract class, but this could lead to future subclasses of this abstract class not wanting this behaviour by default, making it necessary to overwrite it, which is not advisable.

2.6 Interface Design Principle (IDP)

Statement: *"Program" an interface, not an implementation.*

This principle has been taken from Gamma *et al.* [9]. When inheritance is used properly all the classes derived from an abstract class will share the same interface. This implies that a subclass adds or overwrites the operations and does not conceal operations of the parent class. All the subclasses can therefore respond to requests of the interface of this abstract class, and in so doing make themselves subtypes of the abstract class. There are two advantages to manipulating objects only in terms of the interface defined by abstract classes:

1. Clients are not aware of the specific types of the objects they use as long as the objects adhere to the interface expected by the clients.

2. Clients are not aware of the classes that implement these objects, they only know the abstract classes that define the interface.

This greatly reduces implementation dependencies between subsystems. The attributes should not offer themselves as instances of particular specific classes, but should only commit themselves to an interface defined by an abstract class; a common occurrence in patterns.

2.7 Black Box Principle (BBP)

Statement: *Favour the object composition over class inheritance.*

The two most common techniques for reusing functionality in OO systems are class inheritance ("white box") and objects composition ("black box") [9] and [16]. Inheritance and composition have their advantages and disadvantages. Class inheritance is statically defined in compilation time, is easy to use and is directly supported by the language. However it has some disadvantages: the characteristics inherited from the superclasses in execution time cannot be changed, and the superclasses frequently define part of their physical representation to their subclasses. In fact, by giving details of the implementation, the inheritance violates the encapsulation. These implementation dependencies can cause problems when one trying to reuse a subclass. Objects composition helps to keep each class encapsulated and focused on one task. Classes and class hierarchies will remain small and will probably not grow. On the other hand, a plan based on objects composition will have more objects (and less classes) and the behaviour of the system will depend on its mutual relationships instead of on class definitions.

2.8 Don't Concrete Superclass Principle (DCSP)

Statement: *Avoid maintaining concrete superclasses, with the* ***corollary****: an abstract class can never be the child of a concrete class.*

Part of the description of this principle corresponds to Gamma *et al.* [9] and Priestly [17]. A concrete class is characterized, among other things, because it contains implemented all its operations, also, many of these operations will be of a specific functionality, with a particular purpose. If a superclass is concrete we will force to all its subclass to take an implementation for defect of a very specific behavior. If some subclass wanted to vary that behavior it could rewrite class father's operations, but this would be an artificial solution and that would have as consequence code replication, failure symptom in the design. Another solution could be that the method of the superclass acts in different ways (in a way in the subclass and of another in the superclass), but this is an explicit reference to a subclass form superclass, then, when we add or eliminate subclass, the code of the method in the father will be modified... *the class is not closed* (to see OCP). The problem, in essence, is that the superclass is executing two roles:

- It is a superclass, it is defining an interface that all the subclasses must complete.
- It is providing an implementation for defect of the interface

A solution to this problem is to apply the principle DCSP and to leave in the superclass behavior the sufficiently generic as for not being varied, leaving the concrete without implementing.

It could be thought that we would encounter the same problem when an abstract class implements a method. The problem lies in that if it is possible to instance the superclass it is because it is too concrete. It is not possible to instance an abstract class, moreover, its methods are very generic and will rarely be modified, as we commented in DIP, abstractions are more stable than concretions.

3 Relationship Between Principles and Patterns.

In general, we can state that in order for an OO system to be of a certain quality this shouldn't violate any principles. On the other hand, patterns contribute to an efficient design but in general the exact relationship between principles and patterns is unknown or more specifically we do not know which principles/s is/are ensuring each pattern.

So, for example, in order to conform to the DIP, one of the strategies could be to use the abstract Factory pattern. The purpose of other patterns such as Prototype, Factory method, etc. is more to perform the Abstract Factory than to directly conform to a principle. Therefore, we can conclude that there are patterns that directly allow a principle to be complied with, whilst other patterns are more related to patterns than to principles: *DIP ... Abstract Factory... Builder, Factory Method, etc.*

The pattern Bridge is also directly related to the DIP principle, this one solves specific problems of the DIP.

Consequently, patterns could be classified according to the principles they follow. The principles would even enable us to create a different catalogue of patterns to that currently existing (in most cases they are simply presented in alphabetical order). Checklists of principles could also be drawn up which assess the design and offer us solutions patterns that ensure that they are complied with.

We can specify more and considering their relationship with the patterns, the principles can contemplate one or several of the following types:

- *Type 1*, the pattern contributes a good solution to the resulting model of the application of the principle ("from the principle toward the pattern").
- *Type 2*, the pattern completes or contains at the principle.
- *Type 3*, the principle can improve a solution to which has been applied a pattern previously ("from the pattern toward the principle").

Table 1 shows an analysis of the principles mentioned in previous epigraphs and their relationship with each pattern of the detailed by Gamma *et al.* [9] in function of the previous types. In this table we can observe as the relationship of patterns has been ordered alphabetically, we can obtain this way an objective order, later on, and based on the principles, we will be been able to obtain analogies.

Principio	OCP			SP			DIP			ISP			DAP			IDP			BBP			DCSP		
Patrón	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
Abstract F.																								
Adapter Class																								
Adapter Object																								
Bridge																								
Builder																								
Chain R.																								
Command																								
Composite																								
Decorator																								
Facade																								
Factory Met.																								
Flyweight																								
Interpreter																								
Iterator																								
Mediator																								
Memento																								
Observer																								
Prototype																								
Proxy																								
Singleton																								
State																								
Strategy																								
Template Met.																								
Visitor																								

Table 1. Relationships between principles and patterns

From the previous table several thesis, uses and investigation lines can be extracted, some examples are the following ones:

- It allows to break down in forces of smaller grain each one of the patterns, facilitating the study of elements common to all the patterns of oneself character: "patterns in the patterns" or "meta-patterns".

- It allows to guide the use of patterns, since it is easier to know how to apply in a correct way a principle that a pattern and once applied the principle is easy to arrive to the pattern. This facilitates us the pattern's good use, in their fair measure, without abuse. For example, the use of NSCP implies us to use creational patterns and this assures us that our system is written in function of interfaces and not in function of implementations.

- It allows a formal study of micro architectures.

- It allows to obtain the forces (principles) that conform the pattern and how depending in its way of incidence in the pattern (type 1, 2 or 3) this can be of different character, examples:

  - We can observe as of the 5 creation patterns, Abstract Factory, Builder, Factory Method, Prototype and Singleton [9], the four first maintain an almost identical kernel of principles while Singelton doesn't complete any principle. The patron Singleton is not a micro architecture (it only describes a class), Singleton treats the creation of objects but he doesn't make it with the same character and the same abstraction that the other four creational patterns, in fact Buschmann *et al.* [10] an Idiom considers it. With regard to the four remaining creation patterns, we observe that they complete the same principles except Builder, since this has the same character that the previous ones but by means of a composition strategy. As we see the study of the principles that intervene in a pattern allows us, among other many things, a finer and based classification.

  - We observe as any micro architecture with some hierarchy that we want to consider design pattern should complete (type 2), at least, the following principles: OCP, SP, DIP, IDP and DCSP.

  - We observe that in patterns structurally identical as State and Strategy the same principles are completed and with the same character.

  - All pattern that completes OCP, SP, DIP, IDP and DCSP in type 2, ISP and DAP in type 3 and BBP don't contemplate it is classified (according to [9]) as of behavior.

- We will be able to look for and/or to validate new design patterns observing if they complete certain of meta-patterns.

The principles allow us to extract good practical OO, observing how the patterns are based and how they are connected with the design, we can apply this way these good practical to other situations of the design, even avoiding to have to use the whole pattern to obtain some of their benefits, practice habitual many times and that makes complex the design.

4 Conclusions and Future Projects

Although over recent years different areas of knowledge related to the construction of OO system such as principles, heuristics, patterns and refactoring techniques have been consolidated, we believe that there is a lot of work still to be done in order to systematize and offer this knowledge to OO designers in such a way that it can be easily used in practical cases. In fact, up until now the different studies that have been published present these elements in a disconnected way which at times makes their application more difficult. This very problem occurs when choosing a pattern and incorporating it into an existing model.

In this article we have presented a compilation of the OO principles that we consider to be most important, as well as an initial analysis of the OO design patterns proposed by Gamma *et al.* [9] and their relationship with these principles. However,

we still have a considerable amount of work to do both in the compilation of more principles and in the study of other patterns.

Our final aim is to offer a detailed systematization of principles, heuristics, patterns and refactoring techniques (together with their respective interrelationships) which will facilitate their application for the designer.

5 Acknowledgements

This research leaves of the DOLMEN project supported by CICYT (TIC 2000-1673-C06-06). On the other hand, we want to thank to ALTRAN SDB the support shown in all moment to this investigation.

References

1. Henderson–Seller B, Eduards JM. The Object-Oriented System Life Cycle. Communications of the ACM 1990; Vol. 33, N° 9: 142-159
2. Liskov BH, Zilles SN. Programming with Abstract Data Types. Computation Structures Group, Memo N° 99, MIT, Project MAC, Cambridge Mass, 1974
3. Meyer B. Object Oriented Software Construction. Prentice Hall, 1988
4. Martin RC. Object Oriented Design Quality Metrics: An analysis of dependencies. ROAD 1995; Vol. 2, N° 3
5. Martin RC. Engineering Notebook. C++ Report 1996; Aug-Dec (published in four parts)
6. Riel AJ. Object-Oriented Design Heuristics. Addison-Wesley, 1996
7. Booch G. Object Solutions. Managing the Object-Oriented project. Addison-Wesley, 1996
8. Coad P. Object-Oriented Patterns. Communications of the ACM 1992; Vol. 35, No 9: 152-159
9. Gamma E, Helm R, Johnson R and Vlissides J. Design patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995
10. Buschmann F, Meunier R, Rohnert H, Sommerlad P and Stal M. A System of Patterns: Pattern-Oriented Software Architecture. Addison-Wesley,1996
11. Fowler M. Analysis Patterns: Reusable Object Models. Addison-Wesley, 1996
12. Rising L. The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, 1998
13. Fowler M. Refactoring improving the design of existing code. Addison Wesley, 2000
14. Johnson R. Personal communication to the authors of this work, 2000
15. Wendorff P. Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. Procedings of the Fifth European Conference on Software Maintenance and Reeingineering, CSMR 2001, IEEE Computer Society.
16. Prieto-Diaz R. Status Report: Software Reusability. IEEE Software 1993; May: 61-66
17. Priestley M. Practical Object Oriented Design with UML. McGrawHill, 2000