Analyzability and Changeability in Design Patterns

Javier Garzás[1] and Mario Piattini[2]

[1] ALTRAN SDB Senior Consultant
Projects Engineering Research Group
ALTRAN SDB
C/ Ramírez de Arellano, 15, 28043, Madrid - SPAIN
jgarzas@altransdb.com

[2] Alarcos Research Group
Escuela Superior de Informática,
University of Castilla-La Mancha
Ronda de Calatrava, s/n. 13071, Ciudad Real – SPAIN
Mario.Piattini@uclm.es

Abstract. It has been a long time since the appearance of the Object Oriented (OO) paradigm. From that moment, the designers have accumulated much knowledge in the design and construction of OO systems. However, at the present time the exclusive use of patterns is not sufficient to guide a design in a formal way. Two important quality parameters for Object Oriented (patterns) Design exist: Analyzability and Changeability. Principles permit to us to analyze an easier manner in which to introduce design patterns. Indirections provide changeability to the pattern. With all the former, we can obtain a metric to answer how much Changeability we can apply in order not to lose the design's Analyzability.

1. Introduction

In the late eighties, the application of patterns in OO appeared and was consolidated, among others, by the work of Coad (1992), Gamma *et al.* (1995), Buschmann *et al.* (1996), Fowler (1996) and Rising (1998). The motivation was to transfer a type of Object Oriented Design Knowledge (OODK) (Garzás and Piattini, 2001), knowledge accumulated during years of experience. Since then, designers have been reading and using patterns, reaping benefit from this experience.

However, at the present time the exclusive use of patterns is not sufficient to guide a design in a formal way, the designer's experience being necessary to avoid overload, non-application or the wrong use of patterns due to ignorance, or any other problems that may give rise to faulty and counteractive use of the patterns. When patterns are used, several types of problems may occur (Wendorff, 2001; Schmidt, 1995):

- Difficult Application.
- Difficult Learning.
- Temptation to Recast everything as a pattern
- Pattern overload.
- Ignorance.
- Deficiencies in catalogues: Search and Complex Application, High Dependence of the Programming Language, Comparatives, etc.

Copyright (c) 2002, Javier Garzás and Mario Piattini. Permission is granted to copy for the SugarloafPLoP 2002 Conference. All other rights reserved

In principle, using design patterns increments design quality. In this sense, there are many works about metrics and design quality, for example (Genero et al., 2000), (Brito e Abreu and Carapuça, 1994), (Briand et al., 1999), (Henderson-Sellers, 1996), etc. Since design quality can be measured by quality metrics, the use of design patterns should lead to better measurements. However, many common object-oriented design metrics indicate lower quality if design patterns are used. In this sense, Reibing (2001) comments that if we have two similar designs A and B for the same problem, B using design patterns and A not using design patterns, B should have a higher quality than A. However, if we apply "classic" object-oriented design metrics to both designs, the metrics tell us that design A is better – mostly because it has less classes, operations, inheritance, associations, etc. Who is wrong? The metrics or the pattern community? Do we have the wrong quality metrics for object-oriented design? Or does using patterns in fact make a design worse, not better? So what is the cause of the contradiction between the supposed quality improvement by design patterns and the measured quality deterioration? (Reibing, 2001)

First, in the following section, we will analyze the maintenance and design patterns and relationship with analyzability and changeability in more detail. Later, we will show a measurement of the impact of the patterns used. In the last sections, we present acknowledgments, our conclusions and future projects, and references.

2. Maintenance and design patterns

According to the ISO/IEC 9126 – 1999 *"Software Product Evaluation – Quality characteristics and Guidelines for their use"* standard maintainability is subdivided into Analyzability, Changeability, Stability, Testability and Compliance.

Software maintenance consumes the largest part of the overall lifecycle cost (Pigoski, 1997; Bennett and Rajlich, 2000). The incapacity to change software quickly and reliably means that organizations lose business opportunities. Thus, in recent years we have seen an important increase in research directed at addressing these issues.

If we obtain a correct OO design, we will obtain better maintenance. Considering the 9126 standard, two important parameters for quality maintenance of Object Oriented Design (patterns) exist:

- *Changeability* allows a design to be able to change easily, an important requirement at the time of extended functionality into an existing code.
- *Analyzability* allows us to understand the design. This is an essential requisite in order to be able to modify the design in a realistic period of time.

To be specific, at the time of applying patterns to a software design, two opposites forces appear and these forces are directly related to maintainability. On the one hand, we put together a common terminology to the applying patterns, we have proven solutions, but we have the inconvenience that the solution, once obtained, can be very complex, and this means that the design is less comprehensible, and modifying the design is more difficult (Prechelt, 2000). Thus, to continue with the previous concepts, a curious relation between Changeability

and Analyzability appears: If we increase the design's Changeability then we will decrease the design's Analyzability, and vice versa.

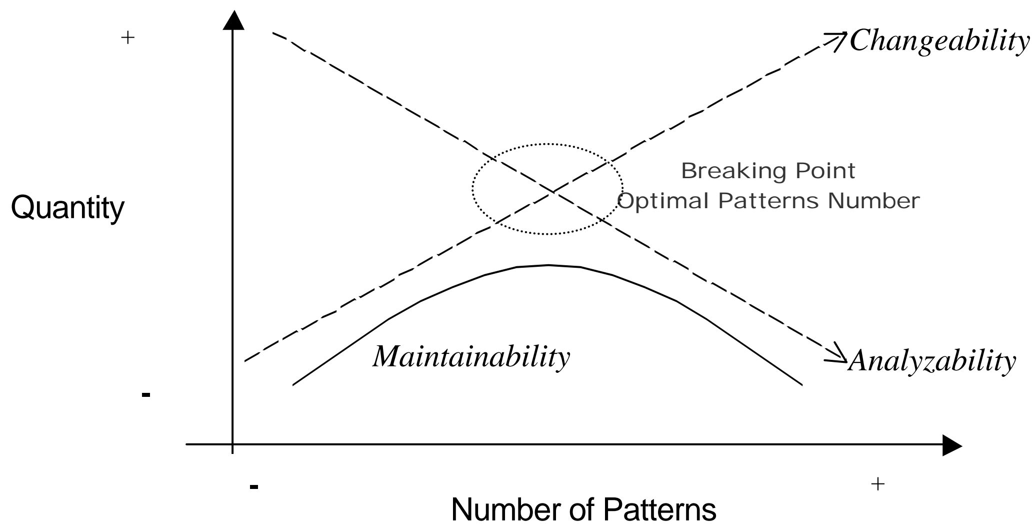

Fig. 1. The Relationship between Changeability and Analyzability. The Breaking Point determines the "Optimal Patterns Number", where the best maintenance is in relation to Patterns.

Figure 1 shows graphically the relationship between Changeability and Analyzability when patterns are applied. As the figure shows:

- If a design has a lot of design patterns this design will have a great amount of Changeability
- If a design has few design patterns this design will have a great amount of Analyzability

2.1 Analyzability issues

In general, the pattern introduction is a complex task. In this epigraph, we show how to introduce design patterns to the design from design principles. This method permits us to analyze an easier way to introduce design patterns.

Garzás and Piattini (2001) comment that 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. Some principles are the following (other principles apart from those described here may exist but we are limited by the length of this paper):

- **Open-Closed Principle (OCP):** A module should be open for its extension and closed for its modification.
- **Substitution Principle (SP):** The subclasses must be substitutable by their base classes.
- **Dependency Inversion Principle (DIP):** Depend upon abstraction. Do not depend upon specifications.

- **Interface Segregation Principle (ISP):** Many clients specific interfaces are better than one general purpose interface.
- **Default Abstraction Principle (DAP):** Introduces an abstract class that makes the implementation in default of most of the interface operations between the interface and the class that implements it.
- **Interface Design Principle (IDP):** "Program" an interface, not an implementation.
- **Black Box Principle (BBP):** Favor the object composition over class inheritance.
- **Do not Concrete Superclass Principle (DCSP):** Avoid maintaining concrete superclass.

In general, we can state that in order for an OO system to be of a certain quality it should not 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 principle(s) ensure(s) each pattern.

Therefore, 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. 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 solution patterns that ensure that they are complied with. We may specify more and consider their relationship with the patterns, so that the principles can be one or several of the following types:

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

Table 1 shows an analysis of the principles mentioned in the previous epigraphs and their relationship with each pattern of those detailed by Gamma *et al.* (1995) in function of the previous types.

Principle		OCP			SP			DIP			ISP			DAP			IDP			BBP			DCSP		
Pattern		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.																									
Adap.	Class																								
	Obje.																								
Bridge																									
Builder																									
Chain R.																									
Command																									
Composite																									
Decorator																									
Facade																									
Factory M.																									
Flyweight																									
Interpreter																									
Iterator																									
Mediator																									
Memento																									
Observer																									
Prototype																									
Proxy																									
Singleton																									
State																									
Strategy																									
Template M.																									
Visitor																									

Table 1. Principles and their relationship with each pattern of those detailed by Gamma *et al.* (1995)

We can observe that the relationship of patterns has been ordered alphabetically. In this way, we can obtain an objective order and later, based on the principles, we will be able to obtain analogies.

Several considerations, uses and investigation lines can be extracted. Some examples are the following:

- It allows us to break down each one of the patterns into smaller forces, facilitating the study of elements common to all the patterns of their own character: "patterns within the patterns" or "meta-patterns".
- It allow us to guide the use of patterns, since it is easier to know how to apply a pattern that a principle in a correct way, and once the principle is applied, it is easy to arrive at the pattern. This facilitates the pattern's good use. For example, the use of NSCP implies the use of 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 us to obtain the forces (principles) that conform the pattern and how, depending in its manner of incidence within the pattern (type 1, 2 or 3), this can be of different characteristic. For example:
  - We can observe that Abstract Factory, Builder, Factory Method and Prototype maintain an almost identical kernel of principles while Singleton does not complete

any principle. Singleton is not a micro architecture (it only describes one class). Singleton deals with the creation of objects but it does not do this with the same characteristic and the same abstraction as the other four creational patterns, we are according to Buschmann *et al.* (1996) whereon this pattern is an Idiom. With regard to the four remaining creation patterns, we observe that they complete the same principles with the exception of Builder, since this has the same character as 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 many other things, a finer and based classification.
  - We observe that any micro architecture with some hierarchy whose design pattern we want to consider should complete (type 2) at least the following principles: OCP, SP, DIP, IDP and DCSP.
  - We observe that in patterns structurally identical to State and Strategy the same principles are completed and with the same characteristics.
  - All patterns that complete OCP, SP, DIP, IDP and DCSP in type 2, ISP and DAP in type 3 and do not fulfill the BBP it is classified (according to Gamma's book) as of behavior.
- We will be able to look for and/or to validate new design patterns observing whether they complete certain 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.

2.2 Changeability issues

The element that provides changeability to the pattern is what it is called indirection. Nordberg (2001) comments that "at the heart of many design patterns is an indirection between service provider and service consumer. With objects the indirection is generally via an abstract interface". Unfortunately each level of indirection moves the software farther from the real world or analysis level view of the problem and deeper into relatively artificial mechanism classes that add overhead to both design comprehension and implementation debugging. With respect to the previous factors, we have observed the following:

- Every time that a pattern is introduced at least one indirection appears in the design and these elements are not of dominion or business logic, such as notifications, observer classes, updates methods, etc.
- Every time that we add an indirection the software moves around further away form the analysis. Upon adding indirections or design classes the design becomes less semantic, less comprehensible or less analyzable.
- Every time that an indirection is added it increases the design changeability.

3. Metrics for Optimal Patterns Number

With all the former, we have a problem: how much Changeability can we apply in order not to lose the design's Analyzability? Obtaining metrics to answer the previous question would be a great contribution.

We can define a parameter that quantifies how changeable a design is in relation to indirections:

Changeability Number (CN) = Indirection Classes Number (ICN) **(1)**

On the other hand, a value that measures the design's analyzability must consider the number of design classes introduced: these are classes that simplify, reusing (such as the subject class into observer pattern) or indirection (such as the observer class into observer pattern). Thus:

Analyzability Number (AN) = Domain Classes Number (DCN) – Indirection Classes Number (ICN) – Simplify Classes Number (SCN) **(2)**

We may observe in the last formula, that when we have an analysis diagram its analyzability is maximum. When we introduce artifacts into the designing phase on the analysis diagram the model's analyzability decreases.

We also may observe, as certain patterns will have a larger impact in the analyzability than other ones, depending on the classes that the patterns introduce.

Now, we may calculate the Optimal Patterns Number (OPN) as follow:

Changeability Number (CN) = Analyzability Number (AN) **(3)**

Indirection Classes Number (ICN) = Domain Classes Number (DCN) – Indirection Classes Number (ICN) – Simplify Classes Number (SCN) **(4)**

Indirection Classes Number (ICN) = [Domain Classes Number (DCN) – Simplify Classes Number (SCN)] / 2 **(5)**

Considering in the previous formula that the DCN parameter is a fixed value in design phase, the rest of parameters depend of the kind of pattern or design artifact introduced.

Shortly, we will add to the measures a bigger refinement level considering aspects such as the quality of methods.

3.1. Example

The following example (figure 2) shows us the Metrics for Optimal Patterns Number application (Changeability Number (CN) and Analyzability Number (AN)).

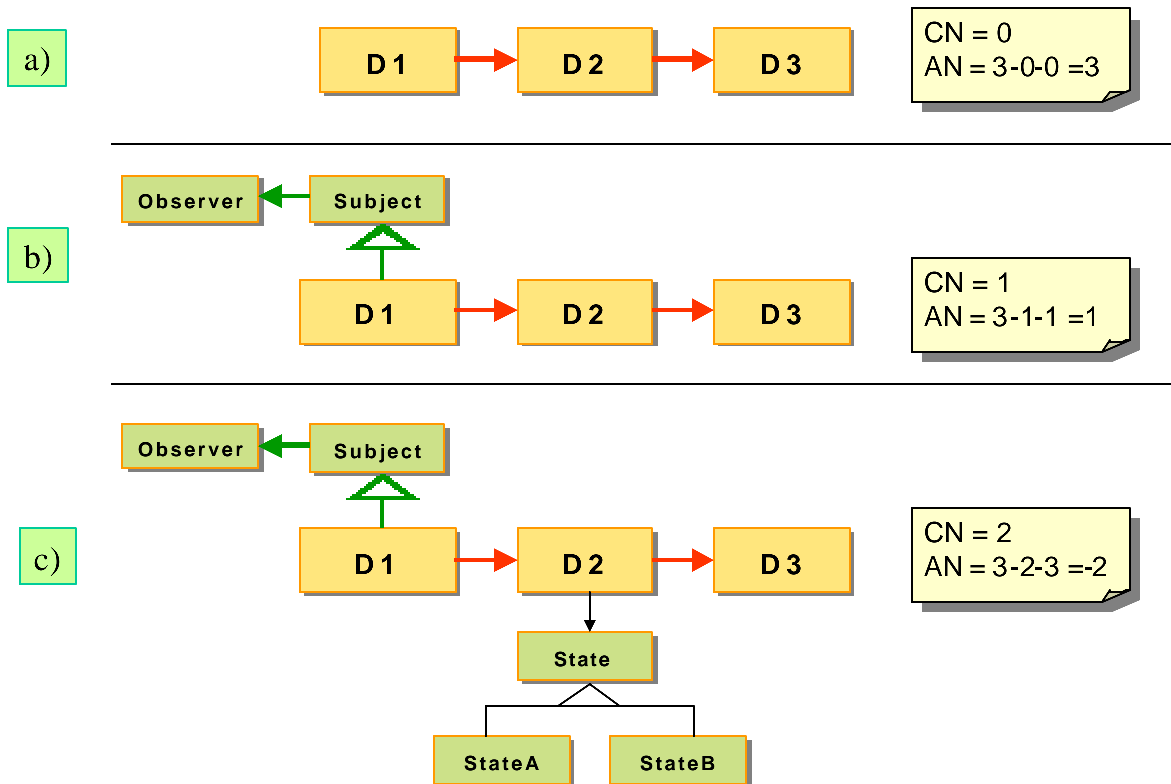

Fig. 2. This example shows the CH and AN variation at the moment of applying patterns

At first, we have three Domain entities (D1, D2 and D3). At this point we have not introduced some patterns, we do not have indirection classes, therefore, CH = 0. With three Domain Classes we have AN = 0 (we do not have Indirection Classes Number (ICN) or Simplify Classes Number (SCN), we only have Domain Classes Number (DCN)).

We introduce the Observer Pattern at a later moment (b in figure). This pattern has an indirection class (observer class), and this introduces one Simplify Class (subject class). With the previous CH = 1 and AN = 3 – 1 – 1 = 1.

Finally, we introduce the State Pattern and this has an indirection class (State class) and, for our example, it introduces two Simplify Classes (StateA and StateB). With the previous CH = 2 and AN = 3 – 2 – 3 = -2.

At this moment the CH number is bigger than the AN number (figure 3). According to the former, at this moment we should not introduce more patterns if we want to maintain the analyzability of design. Perhaps at a later date it may be essential to add more patterns, but at this moment, if we are only improving the design, in a preventive phase, we do not have an explicit need to introduce a pattern. The relationship between CH and AN gives us a rational way to control the patterns' use.

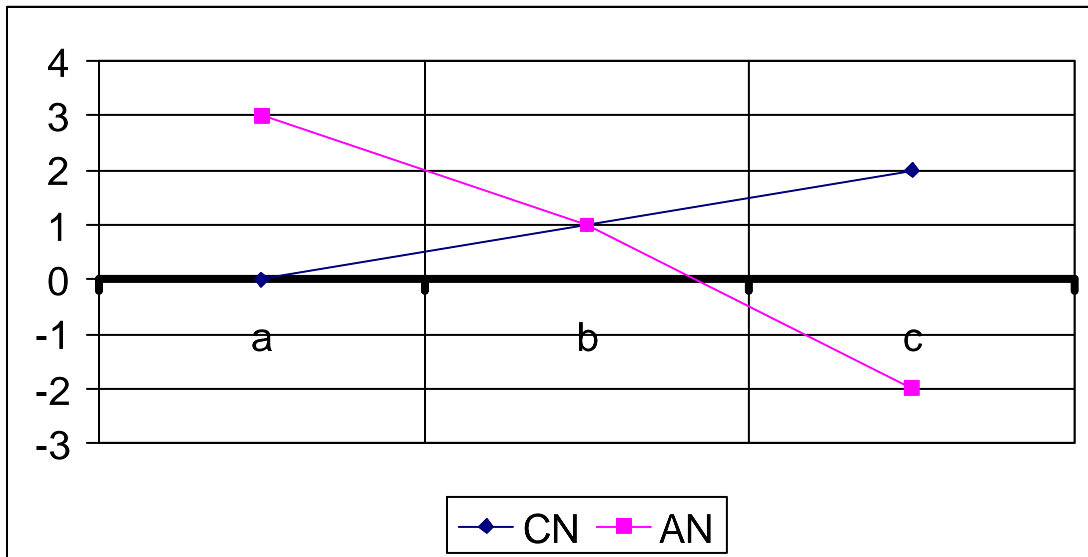

Fig. 3. The graph shows CN and AN evolution

4. Acknowledgments

This research is a part of the DOLMEN project supported by CICYT (TIC 2000-1673-C06-06). We would also like to thank to ALTRAN SDB for the support shown towards this investigation at all times.

5. Conclusion and future projects

The experts have always used proven ideas. It is in recent years when these ideas, materialized into the pattern concept, have reached their greatest popularity and diffusion, thanks to the concern of the community to discover, to classify and to diffuse all types of patterns.

The patterns are useful elements but there are still many elements to be studied if we want to apply them in a rational manner. The first thing that we must make clear is the word quality for design patterns; this word must be used with great care when we apply it to patterns. An appropriate notion of quality should result if the quality definition includes many views. In maintenance quality two appropriate views could be changeability and analyzability.

On the other hand, more knowledge exists apart from that related to patterns, although it would be true to say that this other knowledge is frequently "hidden". We denominate, distinguish and classify the following categories in OODK: principles, heuristic, patterns and refactorings (Garzás and Piattini, 2001). But there is much uncertainty with regards to the previous elements. In fact, the previous knowledge elements have never been studied as a whole, neither its compatibility has been studied nor does a method based on this knowledge exist. There is still a lot of work to be done in order to systematize and offer this OO Design knowledge to designers in such a way that it can be easily used in practical cases.

6. References

Bennet K. H. and Rajlich V. T. Software Maintenance and Evolution: a Roadmap, in Finkelstein A. (Ed.) The future of Software Engineering, ICSE 2000, June 4-11, Limerick, Ireland, pp 75-87.

Brito e Abreu F. and Carapuça R. Object-Oriented Software Engineering: Measuring and controlling the development process. 4th Int Conference on Software Quality, USA, 1994.

Briand L., Morasca S. and Basili V. Defining and Validating Measures for Object-Based high-level design. IEEE Transactions on Software Engineering, 25(5), 722-743, 1999.

Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M., A System of Patterns: Pattern-Oriented Software Architecture, Addison-Wesley, 1996.

Coad P., "Object-Oriented Patterns", Comm. ACM, Vol. 35, No 9, Sep. 1992, pp. 152-159.
Fowler M. Analysis Patterns. Addison Wesley, 1996.

Gamma E, Helm R, Johnson R and Vlissides J. Design patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.

Garzás J., Piattini M. Principles and Patterns in the Object Oriented Design, OOPSLA 2001 - Workshop "Beyond Design: Patterns (mis) used". Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Genero M., Piattini M. and Calero, C. Early Measures For UML class diagrams. L´Objet. 6(4), Hermes Science Publications, 489-515, 2000.

Henderson-Sellers B. Object-oriented Metrics - Measures of complexity. Prentice-Hall, Upper Saddle River, New Jersey, 1996.

Nordberg M. E. Aspect-Oriented Indirection – Beyond OO Design Patterns. OOPSLA 2001 - Workshop "Beyond Design: Patterns (mis)used". Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Prechelt L., Unger B., Tichy W. Bossler P. A controlled Experiments in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Transactions on Software Engineering, September 2000.

Pigoski, T. M. Practical Software Maintenance. Best Practices for Managing your Investements. Ed. John Wiley & Sons, USA, 1997.

Reibing R. *The impact of Pattern Use on Design Quality.* OOPSLA 2001 - Workshop "Beyond Design: Patterns (mis)used". Octubre 14-18, 2001. Tampa Bay, Florida, USA.

Rising L., The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, 1998.

Schmidt D. C., Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software, Communications of the ACM 38,10, October 1995, pp 65-74.

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 Reengineering , CSMR 2001, IEEE Computer Society.