MAINTENANCE COST OF A SOFTWARE DESIGN
A Value-Based Approach

Daniel Cabrero
Dirección General de Tráfico, Spanish Ministry of Internal Affairs, Madrid, Spain

Javier Garzás
Kybele Consulting S.L. Madrid, Spain

Mario Piattini
Alarcos Research Group. University of Castilla-La Mancha. Ciudad Real, Spain

Keywords: Value-Based design, cost estimation, maintenance, design evaluation.

Abstract: Alternative valid software design solutions can give response to the same software product requirements. In addition, a great part of the success of a software project depends on the selected software design. However, there are few methods to quantify how much value will be added by each design strategy, and hence very little time is spent choosing the best design option. This paper presents a new approach to estimate and quantify how profitable is to improve a design solution. This will be achieved by estimating the maintenance cost of a software project using two main variables: The probability of change of each design artifact, and the cost associated to each change. Two techniques are proposed in this paper to support this approach: COCM (Change-Oriented Configuration Management) and CORT (Change-Oriented Requirement Tracing).

1 INTRODUCTION

Recently, a Value-based software engineering (VBSE) agenda has emerged (Boehm, 2005), with the objective of integrating Value considerations into the full range of existing and emerging software engineering principles and practices. One of the major elements of this agenda is Value-based architecting, which involves the further reconciliation of the system objectives with achievable architectural solutions.

Some work has been already published according to the agenda. In particular, Value considerations about requirements (Cleland-Huang and Denne, 2005, Cleland-Huang et al., 2004, Heindl and Biffl, 2005, Srikanth and Williams, 2005) and test Value-based aspects (Egyed et al., 2005, Huang and Bohem, 2006, Srikanth and Williams, 2005). However, very few proposals about Value-based design have been written. (Kazman et al., 2001) exposes an architecture-centric approach to the economic modelling of software design decision making called CBAM (Cost Benefit Analysis Method), in which costs and benefits are traded off with system quality attributes. However, this method doesn't provide any clue of how this cost should be calculated. Up to now, there is no work addressing how to calculate cost and benefit of each design decision in a Value-based context.

The way each design decision affects to the maintainability and global cost of software projects is still an open research issue. In general, each design artifact has a different relative importance. In fact, the contribution to the global design will vary depending on where and how the solution is applied. This lead us to the concept of "Value-based" Design.

2 FROM DESIGN TO VALUE

Software maintenance consumes the largest part of the overall lifecycle cost (Bennet and Rajlich, 2000, Pigoski, 1996). The incapacity to update software quickly and reliably means that organizations lose business opportunities. Thus, in recent years we

Cabrero D., Garzás J. and Piattini M. (2007).
MAINTENANCE COST OF A SOFTWARE DESIGN - A Value-Based Approach.
In *Proceedings of the Ninth International Conference on Enterprise Information Systems - DISI*, pages 384-389
DOI: 10.5220/0002358903840389
Copyright © SciTePress

have seen an important increase in research addressing these issues.

Considering the ISO 9126 standard, three important parameters for quality maintenance of Object Oriented Micro Architectural Design exist: Analyzability, Changeability and Stability:

- **Analyzability** allows us to understand the design.
- **Changeability** allows a design to be able to be altered, an important requirement at the time of extended functionality into an existing code. In our case, the element that provides changeability 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".
- **Stability** allows us reduce risk of unexpected effect of modifications.

Introducing indirections, such as abstraction layers or patterns, has a big impact on the relationship among analyzability, changeability and stability. Thanks to the fact that experienced software engineers and domain specialists develop patterns, the software community can take advantage of this reliable knowledge, available from pattern libraries (Gamma et al., 1995).

This new approach was an important milestone when talking about design techniques. Since then, however, a lot of applications have been designed. Some of them implemented no patterns at all, and on the other hand, some others are overloaded with patterns. But then, when should I introduce a new indirection?

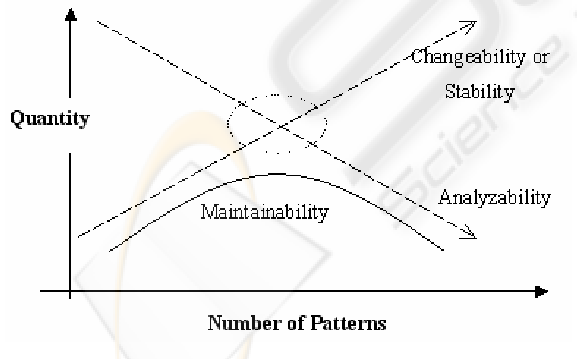

Figure 1: Impact of the number of Patterns on Maintainability extracted from (Garzás and Piattini, 2002).

For example, if a design implements a lot of design patterns this design will have a great amount of changeability and stability, but it won't be analyzable. On the other hand, if it doesn't provide point of change (indirections) it will be analyzable, but difficult to change. Figure 1 depicts this context. This problem lead us to the point that not every pattern or indirection adds the same Value to the design, it depends on where and how it's used. This will be discussed in detail in the next section.

3 COST DRIVEN DESIGN

3.1 Cost of Maintainability

A first approach for calculating the cost of a design is the generic cost model showed in the equation 1.

$$C_t = C_i + C_m \qquad (1)$$

The Total Cost of Construction (C_t) is defined by two factors: The initial development cost (C_i), and the maintenance cost of that implementation (C_m).

Extending this model, this paper proposes to decompose the problem into pieces, and to apply this model to each piece that predicts which will be the cost of development and maintenance during the whole project's life, taking into account not only the cost, but also the long-term benefit.

This can be done multiplying the probability of change of each component by the cost of change. Of course, the probability of change will depend on the estimated life expectancy of the project. Thus, the equation 2 models the maintenance cost in this scenario.

$$C_m = \Sigma_{\text{year, comp}} \text{Probability(change)} * C_{change} \qquad (2)$$

Where $\Sigma_{\text{year, comp}}$ Probability(change) is the probability of change of a component in the whole life of the project, and C_{change} is the cost of change of this component. The feasibility of this model depends on a correct estimation of probability and cost. In addition, a usable technique of decomposition of the problem must be defined.

3.2 Cost of Refactoring

(Fowler, 1999) introduced the concept of "Refactoring". To refactor an application is to change its internal design in order to make it easier to maintain but conserving the same external behaviour.

In this study, we will use the model to decide whether or not to refactor a faulty expensive-to-maintain application. In terms of viability and costs, a software company has to check if the cost of refactoring ($C_{refactoring}$) plus maintenance (C_m(refactored)) is smaller than the money spent in the long term maintenance cost without refactoring (C_m). See equation 3.

$$C_m > C_m(\text{refactored}) + C_{refactoring} \quad (3)$$

If this statement is true, the refactoring of the faulty application is viable. Otherwise, it could be better not to modify the application or to build it from the beginning.

To be able to apply this model, we must estimate several variables: the probability of change of a component ($\Sigma_{\text{year, comp}}$ Probability(change)), the cost of a change (C_{change}), and the cost of refactoring ($C_{refactoring}$). The rest of the paper will focus on this task.

4 ESTIMATING THE PROBABILITY OF CHANGE

As stated in section 2.2, the feasibility of this model depends on a correct estimation of probability of change.

Two techniques are proposed in this paper to tackle this problem: COCM (Change-Oriented Configuration Management) and CORT (Change-Oriented Requirement Tracing).

4.1 Change-Oriented Configuration Management (COCM)

This technique reviews which object has changed and when it happened. The Table 1 shows an example of data extraction.

Table 1: Example of COCM data extraction.

Object	Changes
Object 1	10/10/2006 12/10/2006 15/10/2006
Object 2	10/10/2006 12/10/2006
Object 3	15/10/2006

The historical information available on the Table 2 might be transformed into probability of change. This probability will be obtained by normalizing the number of changes.

Table 2: COCM probability of change.

Object	Estimated probability of change
Object 1	3/3 = 100%
Object 2	2/3 = 66%
Object 3	1/3 = 33%

The main advantage of this technique is that it can be easily automated, but there are two pre-requisites for its utilisation. The first is that we need to have all this information available in a Configuration Management Tool. The other is that this tool must have been already used for a long enough period of time to receive a representative amount of change requests.

4.2 Change-Oriented Requirement Tracing (CORT)

Using final user input to estimate probability of change of a design artifact is a difficult task for two reasons. First, among the existing types of software artefacts (requirement, design, code, and so on), requirement artifacts are the only items that can be understood by the stakeholders. For this reason, tracing techniques need to be used to identify where those requirements corresponds to design artifacts. Second, in most cases, the only link between designers and final users is the gathered requirements, which have no mention to the probability of change and earned Value of each one.

To solve this problem, we propose to extract and document some additional information concerning the probability of change for each requirement. Later, this information will be traced into design artifacts to find out which requirement will change and how, according to the final user.

4.2.1 Requirement Elicitation Approach

The aim of this new approach is to identify which requirement will change. Our technique is inspired by a case study presented by (Srikanth and Williams, 2005), that used a method called VBRT (Value Based Requirement Tracing), which sets a requirement priorization based on the risk and the relative Value of each requirement. In our case, we will use a similar approach but focusing on changeability.

In this way, after identifying stakeholders of the project, they are asked to assign a "changeability" variable to each requirement and use case. This will generate a matrix of requirement/stakeholders with a number from 1 to 10 expressing the "estimated variability" of each requirement from each stakeholder point of view, as showed in the Table 3.

Table 3: "Estimated Variability" matrix.

	User 1	User 2	User 3
Req. 1	3	5	0
Req. 2	2	7	1
Req. 3	0	1	0

This variable is then normalized in order to obtain a probability. The Table 4 depicts the process.

Table 4: Probability of change per user of each requirement.

	User 1	User 2	User 3
Req. 1	3/7 = 28%	5/7 = 71%	0
Req. 2	3/7 = 28%	7/7 = 100%	1/7 = 14%
Req. 3	0	1/7 = 14%	0

The next step is to calculate the average or probability (or adjusted average assigning weights to different users if each stakeholder is not equally important). In this case, for simplicity, we will consider that all users have the same relative importance, and we won't use weights. The Table 5 shows probability of change of the each requirement.

Table 5: Probability of change of each requirement.

	Probability of Change
Req. 1	33%
Req. 2	47%
Req. 3	5%

The main advantage of this approach is that it can be easily used by many software requirement tools, which already have variables associated with requirements, such as importance or frequency. But on the other hand, we have the problem that we need the software requirements specification and direct contact with stakeholders, which is not always available.

4.2.2 Tracing Requirements to Design Artifacts

In software development projects there are interdependencies between all kinds of artifacts, e.g. requirements, design, source code, test cases. Requirements tracing is the ability to follow the life of a requirement in a forward and backward direction (Gotel and Finkelstein, 1994).

An interesting summary of tracing oriented to improve the return of investment is provided by (Cleland-Huang et al., 2004). The detailed study of each option is out of the scope of this research. Anyway, all of them finally establish a relationship between a set of requirements and an object or group of objects.

This relationship will allow us to analyse which objects will change if a given requirement changes, and to translate that "probability of change" from the requirement to objects. For example, if requirement 1 has a 33% of probability of change and this change will affect to the object A and B, we could say that the object A and B have a 33% of probability of change.

5 ESTIMATING THE COST OF CHANGE

It's very difficult to estimate what we don't know. For this reason, the first logical step is to understand which kind of changes may be needed. Fortunately, there is a great amount of previous work on Object Oriented Design Knowledge (OODK). In particular, we are interested in "design rules" (Garzás and Piattini, 2005) exposed in the next subsection.

5.1 Classification of Changes Using OODK Rules

In OODK, rules are the "what", patterns are the "how", and refactorings are the "how to apply" design practices. In this case, we are interested in the "what". A high quality design must be compliant with design rules. Table 6 shows some design rules.

Table 6: Some of the OODK rules extracted from (Garzás and Piattini, 2005).

If there is any software design element (class, methods, code, and so on) duplicated, then eliminate the duplication
If there are dependencies on concrete classes then these dependencies should be on abstractions.
If there are unused or little-used items then eliminate them
If a super class knows any of its subclasses then eliminate it.
If a class collaborates with too many others then reduce the number of collaborations.
If a change in an interface has an impact on many clients then create specific interfaces for each client.
If a service has many parameters then create various methods, reducing the list, or put these into an object.
If the attributes of a class are public or protected then make them private and access them through services.

Thus, when we have a software design and we need to improve it, the most intuitive way is to detect the deficiencies (in this case, violated rules) and fix a percentage of deficiency resolutions. Note that we will use the probability of change to know when will be profitable to make the change.

5.2 Estimating Cost of Each Type of Change

The last step in our research is to estimate how much time will be needed to make a maintenance update in the code where a OODK rule is violated (we will call it Cost A), how much will cost to fix that rule violation (Cost B) and to make the change once the design has been improved (Cost C). The cost will be expressed in "hours of work".

Table 7: Example of cost associated to a change.

Rule	Cost A	Cost B	Cost C
Rule 1	4 hours	4 hours	2 hours
Rule 2	3 hours	6 hours	1 hour
Rule 3	8 hours	12 hours	6 hours

The improvement is worth it if the cost A multiplied by the number of modifications (n) is bigger than the cost B plus the cost C multiplied by n. The equation 4 shows the concept.

$$\text{Cost}(A) * n > \text{Cost}(B) + (\text{Cost}(C) * n) \qquad (4)$$

At first sight, if the change must be done only once, probably it will be better in terms of cost not to improve the design. This confirms the importance of the "probability of change" variable estimated in the precedent section.

The most direct and reliable way to estimate how many hours will be necessary to change the code is to plan an experiment where a group of developers accomplish this task. This experiment must be repeated for each design rule. The output of this experiment could be similar to the data exposed in the Table 7.

Note that this experiment still hasn't been carried out in the context of this research, as noted in the "Future Work" section.

6 GATHERING ALL TOGETHER

A company maintaining a faulty and expensive-to-maintain software may wonder whether or not to redesign some parts of the application, in order to make the application easier to maintain. To assess the potential cost of this application the following steps are proposed.

First, we have to analyse the application in order to identify violated rules and to estimate the cost to modify those designs. This process is explained in detail in section 5. Then, we must estimate the potential cost of maintenance of the code associated to that violated rule. To be able to calculate this cost (C_m) presented in the equation 2, we need to estimate the probability of change and the estimated cost of change. This process is reviewed in the sections 4 and 5.

Later, we use the same equation to estimate the cost of maintenance after improving the code (C_m(refactored)), and estimate the cost of refactoring ($C_{refactoring}$) using techniques exposed in section 5. Only then we will be able estimate if the modification is profitable verifying if the condition exposed in the equation 3 is true.

In this way, it is possible to perform a cost-based guide of design decisions.

7 CONCLUSION AND FUTURE WORK

For future work, it would be very important to carry out an empirical validation of results. Several case studies could be conducted in order to verify empirically the suitability of the proposal.

In addition we plan to carry out an experiment that estimates the cost of correcting a violation of each rule, and the cost of modification of code containing violated rules. This experiment must be planned with a group of developers that accomplish this task, and repeated for each design rule.

We have presented a method for guiding a software design improvement through value and design knowledge. The estimation has been based on how effective the solutions are, instead of measuring functionality or size. This will be achieved estimating the maintenance cost of each solution.

To achieve this goal, we have developed a Value-based model oriented to maintenance costs, due to the fact that the main goal of a design is to make applications maintainable (stable, changeable and analyzable).

ACKNOWLEDGEMENTS

This work has been partially supported by the ESFINGE project (Ministerio de Ciencia y Tecnología (TIN 2006-15175-C05-05) and ENIGMAS (Entorno Inteligente para la Gestión del Mantenimiento Avanzado del Software), supported by the Department of Education and Science of the Junta de Comunidades de Castilla-La Mancha (Regional Government of Castilla-La Mancha) (PBI-05-058).

REFERENCES

Antoniol, G., Lokan, C., Caldiera, G. & Fiutem, R. (1999) A Function Point-Like Measure for Object-Oriented Software. *Empirical Software Engineering,* 4**,** 263 - 287.

Bennet, K. H. & Rajlich, V. T. (2000) Software Maintenance and Evolution: a Roadmap. *ICSE (Track* *on The Future of Software Engineering).* Limerick, Ireland, Finkelstein A.

Boehm, B. (2005) Value-Based Software Engineering: Overview and Agenda. *Value-Based Software Engineering* Springer.

Boehm, B., Horowitz, E., Madachy, R., Reifer, D., Clark, B. K., Steece, B., Brown, A. W., Chulani, S. & Abts, C. (2000) *Software Cost Estimation with Cocomo II* Prentice Hall PTR.

Briand, L. C., Emam, K. E., Surmann, D., Wieczorek, I. & Maxwell, K. D. (1999) An assessment and comparison of common software cost estimation modeling techniques. *International Conference on Software Engineering* Los Angeles, California, United States IEEE Computer Society Press.

Cleland-Huang, J. & Denne, M. (2005) Financially informed requirements prioritization. *International Conference on Software Engineering* St. Louis, MO, USA ACM Press.

Cleland-Huang, J., Zemont, G. & Lukasik, W. (2004) A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. *Requirements Engineering Conference, 12th IEEE International (RE'04).* IEEE Computer Society

Egyed, A., Biffl, S., Heindl, M. & Grünbacher, P. (2005) A value-based approach for understanding cost-benefit trade-offs during automated software traceability. *3rd international workshop on Traceability in emerging forms of software engineering* Long Beach, California ACM Press.

Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code,* Menlo Park, California, Addison Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design Patterns,* Reading, MA, Addison-Wesley.

Garzás, J. & Piattini, M. (2002) Analyzability and Changeability in Design Patterns. *SugarLoafPLoP. The Second Latin American Conference on Pattern Languages of Programming.* Itaipava, Río de Janeiro, Brasil.

Garzás, J. & Piattini, M. (2005) An ontology for micro-architectural design knowledge. *IEEE Software Magazine,* 22**,** 28-33.

Gotel, O. C. Z. & Finkelstein, A. C. W. (1994) An analysis of the requirements traceability problem. *1st International Conference on Requirements Engineering.* Colorado Springs, CO, USA.

Heindl, M. & Biffl, S. (2005) A Case Study on Value-based Requirements Tracing. *10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* Lisbon, Portugal ACM Press.

Huang, L. & Bohem, B. (2006) How Much Software Quality Investment Is Enough: A Value-Based Approach. *IEEE Software,* 23**,** 88- 95.

Kazman, R., Asundi, J. & Klein, M. (2001) Quantifying the Costs and Benefits of Architectural Decisions. *Proceedings of the 23rd International Conference on Software Engineering.* Toronto, Ontario, Canada, IEEE Computer Society.

Nordberg, M. E. (2001) Aspect-Oriented Indirection – Beyond OO Design Patterns. *OOPSLA 2001, Workshop Beyond Design: Patterns (mis)used.* Tampa Bay, Florida, EEUU.

Pigoski, T. M. (1996) *Practical Software Maintenance. Best Practices for Managing your Investements,* NY. USA, John Wiley & Sons.

Srikanth, H. & Williams, L. (2005) On the economics of requirements-based test case prioritization. *7th international workshop on Economics-driven software engineering research* St. Louis, Missouri ACM Press