A Model for Quality Optimization in Software Design Processes

Joost Noppen, Pim van den Broek and
Mehmet Akşit

Trese Group, Dept. of Computer Science, University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{noppen, pimvdb, aksit}@cs.utwente.nl

Abstract. The main objective of software engineers is to design and implement systems that implement all functional and non-functional requirements. Unfortunately, it is very difficult or even generally impossible to deliver a software system that satisfies all the requirements. Even more seriously, failures in fulfilling requirements are generally detected after the realization of software systems. This is because design decisions are mostly taken based on estimations, which can turn out to be wrong at a later stage in the design process. Switching to different design alternatives at a later stage can be very difficult since this may demand drastic changes in design and also may increase project time and costs. In this paper a model is proposed for modeling and tracing design processes with respect to the selected design alternatives. Based on the model, two algorithmic definitions of design strategies are given, which enable software engineers to optimize design decisions with respect to quality and resource constraints.

1. Introduction

It is common to divide requirements into two different categories: functional requirements, which are mostly described as “what the system should do”, and non-functional requirements, mostly described as “how the system should do it”. Non-functional requirements typically describe quality aspects of a software system, such as performance, security or adaptability. The overall goal of the software engineer is to design a system that incorporates all the functional requirements, and also fulfills all the non-functional requirements. In general, while designing the system, the software engineer has to select the most appropriate solution from a large set of alternatives, each with their own quality characteristics.

The difficulty the software engineer faces is caused by the fact that the quality of software products cannot be precisely determined until the final system has been implemented completely. The intermediate choices between design alternatives before the system is implemented are mostly done based on intui-

tion and experience, rather than knowledge and measurement. Choosing any of the available alternatives does not ensure a degree of quality with certainty. This makes it difficult for the software engineer to assess the fulfillment of the non-functional requirements, especially in the early phases of software development.

The design of software can be seen as a process of steps, in which customer requirements are transformed into a software system that incorporates these requirements. In each step the current state of the design process, consisting of intermediate design artifacts, is transformed into the next state. However, the next state is not uniquely defined. Generally a large number of alternatives are possible with different characteristics. It is the task of the software engineer to identify and select the alternatives, which fulfill the requirements best.

At each step in the design process, the alternative is selected which is estimated as the one that offers the best quality characteristics while still conforming to the functional requirements. However, there is no guarantee that the final system will adhere to these quality estimations once it is completed. In retrospect, an increasing insight during the design process can offer additional information on the accuracy of the estimations. With this insight it becomes possible to reassess the selected alternative with respect to its appropriateness.

Even while the detection of unjust choices of design alternatives is possible as described above, it remains difficult to determine how the current design should be adjusted to fulfill the quality requirements. The design steps that have been taken should be traced back, until a point is reached where a more appropriate alternative can be selected. In most cases, other alternatives have hardly been considered, which greatly troubles the adjustment of the design.

To enable software designers to achieve a higher quality for their design, a better insight into quality predictions for their design choices should be given. In this paper we propose a model which traces design decisions and the possible alternatives. With this model it is possible to minimize the cost of switching between design alternatives, when the current choice cannot fulfill the quality constraints. With this model we do not aim to automate the software design process or the identification of design alternatives. Much rather we aim to define a method with which it is possible to assist the software engineer in evaluating design alternatives and adjusting design decisions in a systematic manner.

The remainder of the paper is organized as follows: in the second section non-functional requirements are explained in more detail. The third section describes the model that is used for capturing design alternatives. In section four, knowledge is described that is specific to the software engineering domain and can be used to enrich the alternative model. Section five describes a case study in which the approach is applied to an example case. In section six, related work is described, and in section seven the paper is concluded.

2. Non-Functional Requirements in Software Design Processes

The non-functional requirements are the main factor to influence the choice for one alternative over the other. This is inherent to the definition of alternative, being a functional equivalent with differing quality attributes. However, to be able to compare design alternatives, their respective quality attributes need to be quantified. By defining means to compare and trade off the quantifications of each of the design alternatives, the experience and intuition of the software engineer can be made explicit.

For the quantification we will make a distinction between *non-functional behavior* and *non-functional requirements*. Non-functional behavior can be described as the way a system behaves with respect to quality aspects. For instance, the actual performance of a system can be seen as the non-functional behavior of the system with respect to performance. From this perspective a non-functional requirement can be seen as a constraint that is imposed on the non-functional behavior that is allowed. For instance, a non-functional requirement with respect to performance could state that the performance of a behavior should always adhere to a certain constraint.

3. Modeling Design Alternatives

During the design of a software system, a sequence of decisions is made, by selecting the most promising design alternatives. At each phase in the process a number of functionally equivalent alternatives can be selected. Each of the possible alternatives will eventually result in a system that fulfills the functional requirements. The resulting systems only differ in their non-functional behavior. The alternatives span a (potentially very large) search space, and it is the job of the software engineer to find a system that satisfies the non-functional requirements. Therefore design can be seen as a search problem. This search space is modeled as a *principle design tree* (see next section).

3.1. Design trees

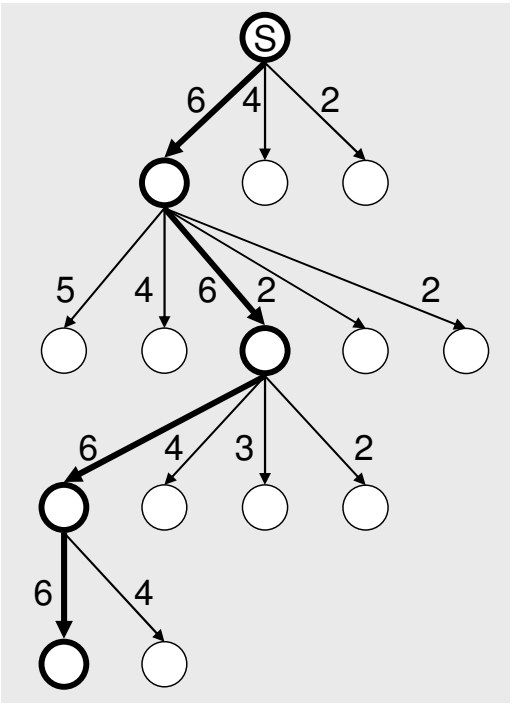

Fig. 1: design tree

The *principal design tree* is the tree where the leaves are the *completed designs* and nodes are *partial designs*. Partial designs are designs, which have at least one *design issue* to be resolved, before the design phase is completed. One of the design issues is chosen to be the principal design issue. The principle design issue to be solved has a number of functionally equivalent alternative solutions. These alternative solutions determine the (partial) designs which are the children of the given partial design. In the design process, the principle design tree is explored. The current state of the design process is given by a *design tree*, which is a part of the principle design tree, and is equal to the portion of the principle design tree which has been explored thus far. At each step in the design process a node of the design tree is expanded, i.e. its children in the principle design tree are added to the design tree. Since the principle design tree is usually too big to explore, the design tree is only expanded until a design is found of acceptable quality.

In figure 1 a design tree is depicted. This tree contains all the nodes that have been considered in the design process. The initial state of the design process is denoted by the root of the principle design tree, labeled “S”. From this state a choice needs to be made between three alternatives. The selected alternative has been depicted with a thick line. Only for the selected alternative, the next level alternatives have been unfolded. The subsequent selection and unfolding of alternatives will finally lead to a system that implements the functional requirements.

The actual selection of one of the alternatives over the others is done based on the intuition and experience of the designer. By assessing the expectations with respect to quality for each of the alternatives, an ordering can be estab-

lished between the possible options. This ordering can be added to the design tree by attaching labels to the edges originating from a specific state. In figure 1 this is done by attaching numbers to the edges of the tree. These numbers represent the expected quality of the final system, when choosing this edge. We suppose that all quality estimations are done in an optimistic manner, meaning that the estimated quality should always be greater than the actual quality that can be achieved. As a consequence, partial design with a low quality estimation needs not to be expanded, since the quality of any resulting final design will not be higher.

3.2. Quality Modeling

Since the designer wants to produce a high-quality software system, most design decisions between alternatives are made based on quality information. The quality of a system generally consists of a number of properties and how well a system performs with respect to these properties. We suppose that the quality of system can be expressed using the elements of a completely ordered set.

As an example, suppose there is a system which quality is expressed in two properties (for instance performance and adaptability). The quality can be represented by a tuple (x, y) where x and y are real numbers, x representing performance and y representing adaptability. In addition suppose that the non-functional requirements are that performance should be at least equal to x_0 and adaptability should be at least y_0. We can now depict the situation as follows:

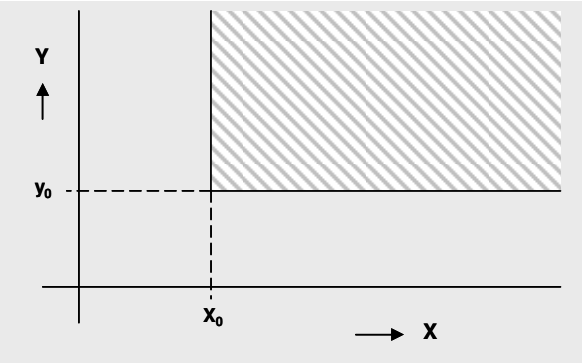

Fig. 2: feasible quality area

In figure 2 the feasible region for the final design is cross-hatched. The non-functional requirements do not allow a system which properties are located outside this area. In addition, a system which properties are located further away from the lower left corner of the feasible region can be presumed to have higher quality.

The total ordering of the set of quality estimations is for instance given by:

$$(x_1, y_1) \geq (x_2, y_2) \quad \Leftrightarrow \quad f(x_1, y_1) \geq f(x_2, y_2)$$

Here f is given by:

$$f(x, y) = \sqrt{((x-x_0)^2 + (y-y_0)^2)} \quad , \text{if } x \geq x_0 \ \& \ y \geq y_0$$
$$f(x, y) = -1 \quad , \text{otherwise}$$

3.3. Design Strategies

The software design process consists of repeatedly choosing a node from the design tree to expand, expanding it, and estimating the quality of its children. A design strategy then is an algorithm, which returns the node to expand, given a design tree and the quality estimations of its leaf nodes. The preference of one strategy over the other is based on managerial motives such as minimization of costs, or time to market. In addition it might be possible to define a number of alternative design strategies, each of which could have their own specific managerial motive.

It is a time-consuming operation to traverse the tree to find a new node to expand. For this purpose a list-based storage-and-retrieval structure will be defined to be able to access the nodes easily. A list L contains all the leaves of the design tree. The nodes are ranked based on the design strategy. The first node of L is chosen as the next node to be expanded. When a node is expanded, it is replaced in the list by all its child nodes. After this operation the list is ordered again. Note that the design strategies themselves are variants of the branch-and-bound searching algorithms, and are in particular variants of the well-known A*-search algorithm, which is for instance described in [11].

A general algorithm for the design process can be described as follows:

```
Design
{
        List L = { Root Node };
        Node N = First element of L;

        While (N is not a completed design)
        {
                Remove N from L;
                Add Children of N to L;
                Rank(L);
                N = First element of L;
        }
        Return N;
}
```

Fig. 3: general design algorithm

In this algorithm the procedure *Rank* rearranges the list L according to the chosen strategy, so the strategies are implemented in Rank. This means the design strategies only differ in the comparison criterion for two nodes. We will present two design strategies, which are considered to be relevant and applicable for software design processes.

The first strategy aims to find the best design possible. This is implemented in Rank, which orders the nodes according to their estimated quality value. Consequently the node with the highest quality estimation is selected. The process continues until a node is reached that is a completed design. When a node is selected, which is a complete design, this node is the best design possible, since all estimations were optimistic However, since the number of design alternatives can be very large, this may take a very long time.

The second design strategy aims to offer a trade-off to the software engineer with respect to the time needed to find a design. By searching for a design whose quality is not less than some given minimal quality in stead of searching for the absolute best, a satisfactory design can be found much sooner. In this strategy the ranking of the nodes is done with respect to satisfaction of the minimal quality requirement, the depth in the tree and the quality estimation, in this order. The value of a node in the design tree is represented by a tuple of the type

(Boolean, Number, Number).

Rank computes an ordering among nodes by means of the following selection criterion based on the standard comparison operator for tuples:

$(b_1, n_1, m_1) \quad > \quad (b_2, n_2, m_2) \quad \Leftrightarrow$

$(b_1 > b_2) \ \vee \ ((b_1 = b_2) \wedge (n_1 > n_2)) \ \vee \ ((b_1 = b_2) \wedge (n_1 = n_2) \wedge (m_1 > m_2))$

The first element of the tuple is a Boolean that represents that truth-value of the statement “The quality estimation of the system is not smaller than the minimal requirement for the quality”. The second element is the number that represents the depth of the node in the tree. The third element is a number that represents the actual quality estimation of the node. The final design that is found by this strategy satisfies the minimal quality requirement (if such a design exists), but it need not be the design with the highest quality. However, this strategy will need less time, giving the user the opportunity of a trade-off between time and quality.

4. Software Engineering Domain Knowledge

The estimations of the quality of the design alternatives are based on knowledge from the software engineering domain and practical experience. In this

section we give an overview of the relevant software engineering domain knowledge.

During the last 20 years, a considerable number of design methods have been introduced, such as Structural design [13], Rational Unified Process (RUP) [6] and OMT [10]. These approaches generally differ from each other with respect to the adopted models (functional, data-oriented, object-oriented, etc.). These methods propose a process which is guided by a large set of explicit and implicit heuristics rules. A method may distinguish itself from the others by introducing and emphasizing its own design heuristics. In [1], based on their heuristics, architecture design methods are classified as *artifact-driven*, *use-case driven* and *domain-driven*. In the artifact driven approaches, software is designed from the perspective of the available software artifacts. For example, in the OMT method, a class is identified using the rule: *"If an entity in the requirement specification is relevant then select it as a tentative class"*. In the use-case driven approach, use cases are applied as the primary artifacts in designing software systems. For example, in RUP, analysis packages, which are the primary means to decompose software, are identified based on the rule: *"Identify the analysis packages if use cases are required to support a specific business process"*. In the domain-driven approaches, the fundamental software components are extracted from the concepts of the domain model.

An extensive number of software engineering environments have been proposed to support software engineering methods. Most environments provide model editing, consistency checking, version management and code generation facilities. Despite a considerable amount of research on process modeling [7][4], only a few environments provide a process support. Formalizing design heuristics and providing some sort of expert system support during the design process is not exploited well. We expect more rule-based heuristics support in environments in the form of intelligent wizards. Initial work on this topic is performed in for instance [2], based on the application of fuzzy logic concepts.

5. Case Study: PDA Input & Storage System

For a number of non-functional requirements quantification can be achieved without too much effort, for others this is more difficult. In this section we will restrict ourselves to the quality attributes *performance* and *adaptability* to illustrate our model.

To illustrate our approach, we will present and analyze a very small example. This example is part of the implementation of the software for a Personal Digital Assistant (PDA) operating system. In this particular example we will focus on the means of inputting information into the PDA, and storing this information. The customer asks for a system that is able to take inputs and store them in text file format. The requirements in addition contain two restrictions with respect to non-functional behavior.

FR_1 The system should be able to accept textual input from the user

FR_2 The system should be able to store the given input in text format on a local disk

NFR_1 The system should be able to adapt the file storage format at compile time

NFR_2 The system should able to store an input within x milli-seconds

By application of a generic design process, the requirements are in turn transformed into problems, which are in turn are transformed into solution concepts [9]. Analysis of the functional requirements has lead to the definition of the following design problems:

Table 1. Design Problems

Design Artefact	Description
P_1	How do we get the textual input from the user?
P_2	How do we convert to a predefined format?
P_3	How do we store textual information?

The choice of solution concepts for these problems consists of a variety of options, equivalent in their functional behavior but different in their quality characteristics. For the example case the following alternative solutions were identified:

Table 2. Design Solutions

Problem	Solution	Description
P_1	S_1	KeyBoard Reader
	S_2	Text Transfer Protocol
	S_3	Text File Reader
P_2	S_4	Input Formatter
	S_5	Require Predefined Format
	S_6	Ignore Formatting Issues
P_3	S_7	File Writer
	S_8	XML Processor
	S_9	Text Dump
	S_{10}	Data Base

The principle design issue is chosen to be the design issue with the lowest index. This results in the following design tree:

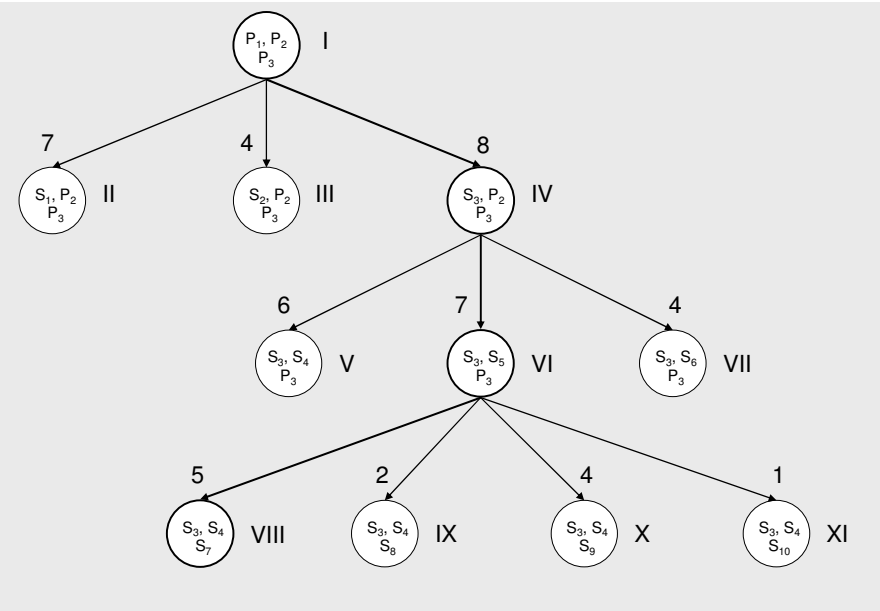

Fig. 4: design tree for example case

In the design tree at the start node the design state consists of the three design problems P_1, P_2 and P_3. P_1 is identified as the principal design issue and for this problem the design alternatives are unfolded (II, III and IV). The three alternatives each have different quality estimations (7, 4 and 8 respectively). The list L will be [IV, II, III]. Using the optimal design strategy, IV is selected, aiming for a quality of 8. However, after examining the next design states a new insight is gained. At the next level there are quality estimations of 6, 7 and 4. However, at this point there is no node with a better quality estimation therefore VI is selected. The list L at this time is [VI, II, V, VII, III].

After unfolding this node again new insight is gained. The best node now has a quality estimation of 5. At this point there are two nodes in the tree that clearly have better estimations: II with value 7 and V with value 6. This strategy will logically continue with II, since L will be [II, V, VIII, X, VII, III, IX, XI].

The other design strategies for this tree will behave differently. Suppose the minimal desired quality is 6. When the same traversal is done over the design tree, the unfolding of VI will not result in a child node of satisfactory quality. The second design algorithm selects V since this is the deepest node in the tree with a satisfactory quality estimation, as opposed to the alternative node II. L in this case will be [V, II, VIII, X, IX, XI, VII, III].

6. Related Work

In [3] the concept of design spaces is introduced, which is an algebraic representation of the possible system design a software engineer can achieve. In this approach quality factors such as adaptability and performance are used to identify and separate relevant from non-relevant design alternatives. Our approach should be seen as complementary to this approach. Where the aforementioned approach is aimed at identifying and ordering the relevant design alternatives, our approach is aimed at the mechanisms that are used for navigating through these design spaces. One of the key issues is the traceability of design processes with respect to deciding amongst design alternatives.

Research has been done on design processes in general before. For instance, in [5] a model for design processes is represented based on advanced Petri nets. In this model a distinction is made between design decisions that are procedural and decisions that are contextual. Procedural decisions are related to project specific issues such as resources and time planning. Contextual decisions relate to the actual state and context of the design process, such as identification of design alternatives or compatibility of solution concepts. These two decision types are modeled and analyzed using advanced Petri nets. The goal of the approach is to simultaneously analyze and consider both types of decisions. Our approach differs in this respect, since our model is aimed at tracing design decisions, and optimization with respect to quality issues.

The model presented in this paper can be classified under the so-called “AI-based problem solving techniques” [8][12]. These techniques generally implement a problem solution strategy and a set of heuristics to guide the engineers in implementing their designs. Most of the work in this area, however, is in designing mechanical or electronic systems.

7. Conclusions

Design processes have proven to be difficult to execute. Decisions that should be taken in the early stages with respect to the possible design alternatives require information that might not be available until (much) later in the design process. The information therefore needs to be estimated, which can be a very difficult task. In case of incorrect estimations, the design process should be halted and re-evaluated with respect to the quality of the resulting system. However, stepping back through the design process to find an optimal point to continue requires a complete tracing of the design history and an understanding of the objectives of the designers and managers.

In this paper we have proposed a model with which it is possible to describe software design processes and in particular the design alternatives that have

been selected. By tracing the design process as a design tree, which is part of a hypothetical tree that represents all possible design executions, it becomes possible to define design strategies in an algorithmic manner. The model can be used as a means to capture the design history, and in addition the model assists in tracing back design decisions whenever estimations with respect to quality have been updated with new information.

Currently the model is still very rudimentary, since it is aimed to be the starting point for reasoning models for design processes in general, and software design in particular. By identifying and modeling additional knowledge with respect to specific areas of expertise, such as software engineering, the effectiveness and versatility of the model can be upgraded. The future work will focus on modeling the non-functional requirements and their evaluation, making estimations and development of a prototype tool for demonstration and experimentation.

8. References

1 Akşit, M., *Introduction and Overview*, In: Aksit, M. (ed), Software Architectures and Component Technology, pp. 1-56, 2002, Kluwer Academic Publishers, ISBN: 0-7923-7576-9

2 Akşit, M., Marcelloni, F., *Deferring Elimination of Design Alternatives in Object-Oriented Methods*, In: Concurreny and Computation: Practice and Experience, Vol. 13, pp. 1247-1279, 2001, Wiley, Online ISSN: 1532-0634, Print ISSN: 1532-0626

3 Akşit, M., Tekinerdogan, B., *Deriving design alternatives based on quality factors* In: Akşit, M. (ed.), Software architecture and Component Technology, pp. 225-257, 2002, Kluwer Academic Publishers, ISBN: 0-7923-7576-9

4 Finkelstein, A., Kramer, J., Nuseibeh, B. (eds.), *Software Process Modelling and Technology*, Taunton, UK, 1994, Research Studies Press, ISBN:0-86380-169-2

5 Horváth, I., Vergeest, J., van der Vegte, W., *Modeling Design Processes and Designer Decisions with Advanced Petri-Nets* In: Lehoczky, L., Kalmar, L. (eds.): International Computer Science Conference (MicroCAD 2000), pp. 81-90, 2000, University of Miskolc, ISBN: 963-661-422-9

6 Jacobson, I., Booch, G., Rumbaugh, J., *The Unified Software Development Process*, 1999, Addison Wesley, ISBN: 0-201-57169-2

7 Kaiser, G., Popovich, S., Ben-Shaul, I., *A Bi-Level Language for Software Process Modeling*, In: Tichy, W. (ed.): Configuration Management, Trends in Software 2, pp. 39-72, 1994, John Wiley & Sons, ISBN: 0471942456

8 Lee, C-L., Iyengar, G., Kota, S., *Automated Configuration Design of Hydraulic Systems*, In: Gero, J. (ed.), Artificial Intelligence in Design, pp. 61-82, 1992, Kluwer Academic Publishers, ISBN: 0792363531

9 Noppen, J., van den Broek, P., Akşit, M., *Dealing with Fuzzy Information in Software Design Methods*, In: Dick, S., Kurgan, L., Musilek, P., Pedrycz, W., Reformat, M. (eds.), Proceedings 2004 Annual Meeting of the North American Fuzzy Information Processing Society, pp. 22-27, 2004, IEEE Press, ISBN: 0-7803-8376-1

10 Rumbaugh, J., Blaha, M., Premerlani, F., Lorensen, W., *Object-Oriented Modeling and Design*, 1991, Prentice Hall, ISBN: 0-13-630054-5

11 Russel, S., Norvig, P., *Artifical Intelligence A Modern Approach*, 1995, Prentice Hall, ISBN : 0-13-360124-2

12 Tong, C., Sririam, D., *Introduction* In: Tong, C., Sririam, D. (eds.), Artificial Intelligence in Engineering Design, Vol. 1, pp. 1-53, 1992, Academic Press, ISBN: 0-12-660563-7

13 Yourdon, E., Constantine, L., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*, 1979, Prentice-Hall, ISBN 0-13-854471-9