Chapter 8

DERIVING DESIGN ALTERNATIVES BASED ON QUALITY FACTORS

Mehmet Akşit and Bedir Tekinerdoğan

*TRESE Group, Department of Computer Science, University of Twente, postbox 217,
7500 AE, Enschede, The Netherlands. email: {aksit, bedir}@cs.utwente.nl,
www: <http://trese.cs.utwente.nl>*

Keywords: modeling design spaces, design alternatives, balancing quality factors

Abstract: Software is rarely designed for ultimate adaptability or performance but rather it is a compromise of multiple considerations. At almost every stage of the software development lifecycle, software engineers have to cope with various design alternatives. Current object-oriented design practices, however, rely mainly on the intrinsic quality factors of the object-oriented abstractions rather than considering quality factors as explicit design concerns. It is considered important to support software engineers in identifying, comparing and selecting the alternatives using quality factors such as adaptability and performance. This chapter introduces a new technique to depict, compare and select among the design alternatives, based on their adaptability and time performance factors. This technique is formally specified and implemented by a number of tools.

1. INTRODUCTION

Software development methods [6][13] provide a set of heuristic rules to guide software engineers to analyze, design and implement software systems. Although heuristic rules can be quite helpful in developing high quality software systems, it is generally difficult for software engineers to identify, compare and prioritize the design alternatives.

Software is rarely designed for ultimate adaptability or performance but rather it is a compromise of multiple considerations. At almost every stage of

the software development lifecycle, software engineers have to cope with various design alternatives. Of course while defining object models, software engineers apply their knowledge and experience. They generally compare the alternatives based on their intuition. This process, however, is rather implicit instead of explicit.

This chapter introduces a new technique called Design Algebra to depict, compare and select the alternatives of a design, based on the adaptability and time performance factors. The software engineer can identify the alternatives at various phases of the development process, and make explicit decisions. This technique is formally specified and implemented by a set of tools.

This chapter is organized as follows. The next section introduces an example and explains the problems addressed in this chapter. Section 3 presents a process for selecting the design alternatives based on the adaptability factors. Section 4 shows techniques to determine the probabilistic time performance factors of the design alternatives. Balancing the adaptability and performance factors is explained in section 5. The related work is presented in section 6. Section 7 discusses the usefulness of the proposed approach. Finally, section 8 gives conclusions.

2. THE PROBLEM STATEMENT

We will use a simple working example throughout the chapter. Section 2.1 presents the example, and section 2.2 explains the problems addressed in this chapter using this example.

2.1 An Illustrative Example: Collection Classes

Assume that we would like to design a set of collection classes, such as *LinkedList*, *OrderedCollection* and *Array* to be a part of an object-oriented library. These classes should provide the necessary operations to read and write the elements stored in collection objects. Further, we would like to define a sorting algorithm to order the items stored in collection objects according to a certain criterion.

Now assume that we analyze this requirement specification using an object-oriented method, such as OMT [13]. OMT defines a set of rules and a process to identify the necessary classes, associations, aggregations, attributes, inheritance relations and operations¹. Applying OMT will possibly result in the class diagram shown in Figure 1. Here classes *Library*,

¹ In this chapter our focus is not OMT but we use OMT as an example of object-oriented software development methods.

LinkedList, *OrderedCollection*, and *Array* are identified by searching for the nouns in the requirement specification. Since these classes share the same abstract behavior, an abstract class *Collection* is introduced, which declares the necessary operations for all its subclasses. The identification of the aggregation relation and the operations *read*, *write* and *sort*, and the attribute *collectionItems* are derived from the requirement specification using the heuristics of the OMT method.

While implementing the class diagram shown in Figure 1, the software engineer has to choose among several options. First, a suitable sorting algorithm has to be selected. Second, the structure of the attribute *collectionItems* must be determined. This structure will mainly depend on the subclass. For example, *LinkedList* will likely have a different attribute structure than *Array*. The operations *read* and *write* and if necessary other operations must be defined and implemented when the sorting algorithm and attribute structure are known. Further, the software engineer must determine if the sorting operation must be implemented in class *Collection* or in the subclasses. If the sorting operation is an abstract (virtual) operation in class *Collection*, then it has to be decided whether some parts of the algorithm must be implemented in the subclasses. Clearly, the class diagram shown in Figure 1 can be implemented in many different ways.

Dealing with the alternatives is not only a concern of implementation. The class diagram shown in Figure 1 is just an example solution for the problem. For example, the sorting operation might be defined as a part object of class *Collection*. This would allow changing the sorting operation at run-time. One might also prefer to change part of the sorting algorithm, for example the sorting criteria (e.g. the Strategy design pattern). There are, of course, a considerable number of design alternatives, depending on the granularity of the required changes, and whether these changes must be realized at compile-time or run-time.

Figure 1: Class diagram of the collection classes

2.2 Problem Description

As illustrated in the previous section, the *design space* of the collection classes is determined by many issues, such as data structures, sorting algorithms, and object-oriented modeling techniques. Although the presence of many alternative solutions indicates the richness of the object-oriented approach, the lack of tool support to compare the alternatives increases the complexity of the analysis and design process.

Consider the object model as shown in Figure 1. We notice two major problems in defining this object model using an object-oriented method like OMT. First, there is no explicit support for identifying the possible design alternatives. Second, although software engineers may prefer to compare the design alternatives based on certain quality factors such as adaptability, performance and reusability, there are no explicit rules to compare the alternatives. Of course while defining object models, software engineers apply their knowledge and experience. They generally compare the alternatives based on their intuition. This process, however, is rather implicit instead of explicit.

Software is rarely designed for ultimate adaptability, performance or reusability but rather it is a compromise of multiple considerations. In general there are many correct solutions for the same problem. Even in the simple case of sorting items in collection objects, one may identify many alternative designs, which will differ with respect to adaptability, performance and reusability factors. Providing ultimate adaptability may create too much run-time overhead. Aiming at the fastest implementation may result in unnecessarily rigid software. Aiming at the most reusable software may introduce redundant abstractions for a given problem. Software engineers must be able to explicitly compare, evaluate and decide between various alternatives based on the relative importance of the quality factors.

3. DESIGNING FOR ADAPTABILITY

In this section we will introduce a process to transform a requirement specification into adaptable object-oriented models. The objective of this process is to gradually introduce domain, design and implementation knowledge into the requirement specification, while selecting the alternatives based on their adaptability factors. This process is formally specified and implemented in a set of tools. We will apply these techniques to the example problem presented in the previous section.

The design process for adaptability consists of the following phases:

Finding the concepts in the requirement specification (section 3.1): This is a necessary step for every software development activity. The output of this phase is a set of concepts.

1. Finding the concepts using domain analysis (section 3.2): In this phase the fundamental abstractions are searched within the context of the solution domain. The objective of this phase is to enrich the concepts obtained from the requirement specification with the concepts that are considered essential in the solution domain.
2. Identification of the adaptable concepts (section 3.3): In this phase the software engineer decides which concepts must be adaptable or fixed. The software engineer may also consider various alternatives and assign adaptability degrees to the alternatives. The purpose of this phase is to make the software engineer conscious about his/her decisions with respect to the adaptability characteristics of the models that he/she develops.
3. Identification of the object-oriented abstractions (section 3.4): The adaptable or fixed concepts delivered from the previous phase are classified according to the object-oriented abstraction techniques. The result of this phase is a consciously selected set of object-oriented abstractions with well-defined adaptability characteristics.
4. Identification of the object-oriented relations (section 3.5): This phase aims at identifying the relations among the identified concepts. The result of this phase is a set of object-oriented relations that satisfy the adaptability requirements.

The total result of this process is a set of alternative object-oriented models that implement the requirement specification. These models may be ordered according to the desired adaptability characteristics. Using this ordering, the software engineer may consciously select one among them. The software engineer may also compare the alternative models both from the adaptability and performance viewpoints. This will be explained in section 5.

3.1 Finding the Concepts in the Requirement Specification

To develop high quality software, it is necessary for any software development method that there is a well-defined requirement specification and sufficient knowledge available about the problem domain. We define a model M as a tuple consisting of a set of concepts and a set of relations

among these concepts. Let us now assume that $M_{Library}$ is a requirement specification model of the collection library example given in section 2.1:

$$M_{Library} = (C_{Library}, R_{Library}) \quad (F1)$$

$M_{Library}$ is represented by the sets $C_{Library}$ and $R_{Library}$, which correspond to the concepts and relations of the requirement specification, respectively. After analyzing the problem description, we identify the following set of concepts in $C_{Library}$:

$$C_{Library} = (Library, Collection, LinkedList, OrderedCollection, Array, collectionItems, sort, read, write) \quad (F2)$$

Note that these elements correspond to the elements of the object model shown in Figure 1. However, in (F2) we have not yet make any assumption about the types of object-oriented abstractions that represent these concepts. The relation set $R_{Library}$ will be considered in section 3.5.

3.2 Finding the Concepts using Domain Analysis

To identify the fundamental abstractions we will now analyze the domain of the problem. This commonly involves collecting the related information from various sources, and detecting the commonalties among them through comparison. These common abstractions generally correspond to the fundamental concepts in that domain². The software engineer is responsible for combining the entities obtained from the requirement specification and the concepts from the background knowledge.

We may discover the concepts of the sorting domain by comparing some well-known sorting algorithms. A number of these sorting algorithms is given in [14]. After comparing these algorithms, we can see that they all share the following 5 concepts: the algorithm type, the range of the sorting process, reading and writing items in the collection, and the criterion to compare the items.

The algorithm type basically defines the control-flow of the sorting process, and is used in the literature to distinguish the sorting techniques from each other. In [14], for example, *selection*, *insertion*, and *bubble sort* algorithms are presented. These sorting algorithms perform different with respect to various factors. For example, the *bubble sort* algorithm is very efficient if the collection is almost sorted. The range of the sorting process

² To identify the fundamental abstractions of a software system, in chapter 5 of this book a synthesis-based approach is presented. Our approach here follows the synthesis-based approach. The reader may also refer to the various domain analysis methods presented in the literature [3] [3].

defines which items in a collection must be sorted. Although, in [14], the range in all examples is set to the full size, after reading the motivation of the sorting algorithms, we decided to introduce the range as a concept. This allows modeling a partial sorting process. To be able to compare the items in a collection, the items must be read, and to change their order, they must be re-written in the collection. Obviously, sorting must be based on a certain criterion. In a simple case, numbers can be sorted by comparing their magnitudes, or in a more complicated case, items can be compared with each other based on certain policy, such as the arrival date, price, etc. Based on these observations, we define the fundamental concepts of the sorting domain as follows:

$$C_{Sort} = (AlgT, RN, RD, WR, CR) \quad (F3)$$

Here, C_{Sort} represents a set of concepts of the sorting domain where $AlgT$, RN , RD , WR and CR are the elements that correspond to the algorithm type, range, reading, writing and the comparison criterion, respectively.

The concepts which are obtained from the requirement specification (F2) and through the domain analysis (F3) should be merged together. The concepts *sort*, *read* and *write* in $C_{Library}$ correspond to the concepts $AlgT$, RD and WR in C_{Sort} , respectively. We prefer the names used in the requirement specification. This results in the following set of concepts:

$$C_{Library} = (Library, Collection, LinkedList, OrderedCollection, Array, collectionItems, sort, RN, read, write, CR) \quad (F4)$$

Figure 2: New models can be entered by the tool *Model Definer*

We have developed various tools to implement the techniques explained in this chapter. As shown by Figure 2, the tool *Model Definer* is used to introduce new models such as *MLibrary*. Using the tool, for each model its

concepts and their relations can be defined. Every model is stored in a global repository and can be accessed by other tools in the system.

3.3 Identification of the Adaptable Concepts

Adaptability can be defined as the ease of changing an existing model to new requirements. To this aim, we have to deal with two contradictory goals: On one hand we have to fix the concepts for robustness and time performance. On the other hand, we need to make concepts adaptable for flexibility [18]. The predefined property P_{Adapt} consists of two elements FX and AD , which is used to qualify concepts as fixed or adaptable, respectively³:

$$P_{Adapt} = (FX, AD) \quad (F5)$$

Classification of concepts as fixed or adaptable creates alternative concept sets with different adaptability characteristics. The predefined properties support the techniques introduced in this chapter. These properties are implemented by the tools. We define the term *design space* to represent a set of design alternatives. Formally, design spaces are defined as function spaces that map concepts to properties. Consider, for example, the following design space:

$$S_{AdaptLibrary} :: C_{Library} \rightarrow P_{Adapt} \quad (F6)$$

The space $S_{AdaptLibrary}$ maps the concepts of $C_{Library}$ to the elements of P_{Adapt} and as such represents the total set of alternatives of library models with adaptability properties. Every alternative can be considered as a specific design decision. For example, the following tuples may be an alternative from this space:

$$CA_{AdaptLibrary} = \{(AD, Library), (AD, Collection), (FX, LinkedList), (FX, OrderedCollection), (FX, Array), (AD, collectionItems), (AD, sort), (FX, RN), (AD, read), (AD, write), (AD, CR)\} \quad (F7)$$

This alternative defines the design decision in which the range *Library*, *Collection*, *collectionItems*, *sort*, *read*, *write* and *CR* concepts have been selected as adaptable (AD) and the other concepts as fixed (FX). There are many more alternatives in $S_{AdaptLibrary}$. The total number of alternatives can be computed using the following formula:

³ Note that other adaptability models can be identified as well. For example, at this stage of design we may also distinguish between compile-time and run-time adaptability. We are currently experimenting with different adaptability models [17].

$$\begin{aligned}
 totalNoAlternatives(S_{AdaptLibrary}) &= size(P_{Adapt})^{size(C_{Library})} \quad (F8) \\
 &= 2^{11} = 2048
 \end{aligned}$$

Hereby, the function *size* returns the number of concepts of each model. The function *totalNoAlternatives* computes the number of alternatives of the design space. Obviously, the software engineer may not be interested in all the alternatives and in addition not all of them may be possible due to various constraints. The design space can be reduced by either selecting a sub-space or eliminating the set of alternatives that are not considered feasible from the perspective of the client requirements or the internal constraints. Formally, both approaches can be specified as follows:

$$S_{AdaptLibrary} :: \{C_{Library} \rightarrow P_{Adapt} \mid (cond)\} \quad (F9)$$

Here, *cond* represents the condition for reducing the design space. The condition *cond* may consist of logical connectives to specify complex conditions. Assume, for example that the software engineer is only interested in the set of alternatives in which the concepts range (RN), read (RD), write (WR) and comparison (CR) are adaptable. This can then be formally expressed as follows:

$$S_{AdaptLibrary} :: \{C_{Library} \rightarrow P_{Adapt} \mid (RN \rightarrow AD) \wedge (RD \rightarrow AD) \wedge (WR \rightarrow AD) \wedge (CR \rightarrow AD)\} \quad (F10)$$

This reduces the number of the total set of feasible alternatives to $2^7 = 128$. Other selection and/or elimination conditions may be easily specified to further reduce the set of alternatives. The elimination conditions can be defined in the same manner by using a negation connective before the specified condition.

We can quantify the design alternatives to order and compare these with respect to different criteria. To reason about the adaptability of each alternative in the acquired set we assign a natural number to each model. This can be specified as follows:

$$S_{AdaptLibrary} :: \{C_{Library} \rightarrow P_{Adapt}\} \rightarrow N \quad (F11)$$

The basic goal of this quantification of alternatives is that it helps to explicitly reason about each alternative with respect to the corresponding

quality factor. The adaptability degree for each alternative may depend on various conditions⁴.

In the following, we will describe the tools that implement the above operations for composing design spaces, quantifying design alternatives and generating design alternatives.

Figure 3 represents a snapshot of the tool *Design Space Composer* that can be used to define and depict the concept spaces. The software engineer can select a model and a property-set, and likewise can compose different design spaces. In Figure 3, the design space *AdaptLibrary* is composed from the property set *Adapt* and the model *MLibrary*. As it is displayed in the tool, *AdaptLibrary* includes 22 tuples, which have been generated by taking the Cartesian product of the sets *Adapt* and *Library*. The tool provides also means to use other set manipulation operations to refine design spaces. In this chapter, however, we will not elaborate on these and refer for a more detailed description to [17].

Figure 3: A tool for composing design spaces

Figure 4 represents a tool for quantifying the tuples of a design space. The top-right widget *Priority* displays the priority number for each generated tuple. Depending on the concept type, different priorities can be assigned to the tuples. For example, if adaptability is considered important, then a higher priority value can be given to the adaptable tuples. In Figure 4, among the adaptable tuples, the concept *sort* has the highest priority and range *RN* has the least. The tuples *read* and *write* have the same priority. The software engineer here assumes that from the perspective of adaptability, changing the

⁴ This value should be interpreted as an indication of the software engineer's preference. The assigned adaptability degree does not imply that it is also available. During the further refinement process when other concerns are introduced it may appear that the required adaptability is not possible due to for example language constraints.

algorithm type has the most impact and changing the range the least. When a priority value is changed, the tool automatically computes the new values. This allows the software engineer to experiment with the priority values.

Model	Tuple	Priority
MLibrary	Library-AD	1
	Collection-AD	1
	LinkedList-AD	1
	OrderedCollection-AD	1
	Array-AD	1
	collectionItems-AD	1
	sort-AD	5
	RN-AD	2
	read-AD	3
	write-AD	3
	CR-AD	4

AdaptLibrary represents the Library concepts together with their adaptability properties. Since adaptability is an important requirement the tuples with AD property have been assigned a higher value than the tuples with a FX

Figure 4: A tool for quantifying design alternatives

To generate alternatives from the predefined design spaces the tool *Alternative Generator* is used from which a snapshot is shown in Figure 5. Initially, the set of alternatives for the design spaces listed in the list box *Design Spaces* is not generated. The widget *no. alternatives* defines the number of alternatives that can be derived from the selected design space. The software engineer can generate the set of alternatives by pressing the *Generate* button. Since this number of alternatives can be quite large, the tool gives an error message when the number of alternatives exceeds a predefined maximum value. If the number of alternatives is smaller than the maximum default value the alternatives will be generated and listed and ordered according to their priority values. This ordering of the alternatives is also shown in the graphic below the list of alternatives. In the graphic each point represents an alternative. The graphic shows only 30 alternatives at once. To browse the other alternatives the left and right arrows at the right corner of the window can be used.

The software engineer can directly select some of these alternatives through the menu of the *alternatives list* and store this in the repository. The design space can also be reduced by either pressing the button *Matrix Selection* or the button *Rule-Based Selection* that opens tools for conditional selection of the sub-spaces and heuristic rule supported selection, respectively.

Figure 5: The tool *Alternatives Generator* is used to generate concept sets from concept spaces

Figure 6 shows the dialog window that is opened if the software engineer presses the button *Rule-Based Selection*. The window shows a dialog with questions that the software engineer needs to select the relevant tuples. In the collection library example, 11 questions are asked. As a reply, the software engineer may choose *Yes*, *No* or *I don't know*. In case *I don't know* is selected, all the alternatives for the corresponding concept are kept. The number of the sets to be generated is displayed in the dialog as well.

The reduced design spaces can be stored as a new design space and imported by the relevant tools for further refinement. After selecting the tuples, and generating the alternatives the software engineer may then compare these based on their adaptability degrees.

Figure 6: The dialogue that helps the software engineer to select the required adaptable tuples

3.4 Identification of the Object-Oriented Abstractions

In the object-oriented model, a concept can be represented either as a class, an operation or an attribute. The predefined property P_{Object} is a set of object modeling alternatives for concepts:

$$P_{Object} = (CL, OP, AT) \quad (F12)$$

Here, CL , OP and AT , represent classes, operations and attributes, respectively. We can further classify operations as virtual (Op_v) and not virtual (Op_n) operations, attributes as mutable (AT_m) and constant attributes (AT_c), etc⁵. Virtual operations can be polymorphically overridden through inheritance. Constant attributes cannot change at run-time.

The following concept space $S_{ObjectAdaptLibrary}$ maps the concepts of $C_{AdaptLibrary}$ to the elements of P_{Object} and as such represents the total set of alternatives of library models with adaptability properties:

$$S_{ObjectAdaptLibrary} :: C_{AdaptLibrary} \rightarrow P_{Object} \quad (F13)$$

Again, this design space may be reduced by selection functions:

$$S_{ObjectAdaptLibrary} :: \rightarrow \{C_{AdaptLibrary} \rightarrow P_{Object} \mid (cond)\} \quad (F14)$$

We have to extend the adaptable concept model with the property *objectModel* to store one of the selected object-oriented abstractions, which is either CL , Op_v , Op_n , AT_m or AT_c .

The tool *Alternatives Generator*, which was shown in Figure 4 for selecting the adaptable library concepts, can be used here as well. This space is generated from the property *Object* and the concept set *AdaptLibrary*. The software engineer may directly select alternatives from this space or first reduce it. When the button *Rule Assistance* is pressed, a dialog window is opened⁶ such as shown in Figure 7. During a dialog session, the following questions could be asked to the software engineer:

IF THE TUPLE IS ADAPTABLE:

(R1) IF THE CONCEPT REPRESENTS AN OPERATION AND RUN-TIME ADAPTABILITY IS REQUIRED,
THEN DEFINE IT AS A PAIR OF OPERATION (OP_v) AND CLASS (CL);

⁵ It is also possible to define derived classes and attributes.

⁶ This dialog window is generated by the property *Object*. In this way, the tool *Alternatives Generator* remains generic. Similarly, the dialog window shown in Figure 6 was generated by the property *Adapt*.

- (R2) IF THE CONCEPT REPRESENTS AN OPERATION AND COMPILE-TIME ADAPTABILITY IS REQUIRED,
THEN DEFINE IT AS A VIRTUAL OPERATION (OP_v);
 - (R3) IF THE CONCEPT IS AN ATTRIBUTE, THEN DEFINE IT AS A MUTABLE ATTRIBUTE (AT_m).
 - (R4) IF (R1) TO (R3) ARE NOT APPLICABLE THEN DEFINE THE CONCEPT AS A CLASS (CL);
- IF THE TUPLE IS FIXED:
- (R5) IF THE CONCEPT IS AN ATTRIBUTE,
THEN DEFINE IT AS A CONSTANT ATTRIBUTE (AT_c);
 - (R6) IF THE CONCEPT IS AN OPERATION
THEN DEFINE IT AS A (NON-VIRTUAL) OPERATION (OP_n)
 - (R7) IF (R5) AND (R6) ARE NOT APPLICABLE
THEN DEFINE THE CONCEPT AS A CLASS (CL).

The rule (R1) assumes that if the concept is an operation and is run-time adaptable, then it must be represented as an operation declared at the interface of a mutable object (like the Strategy pattern). According to the rule (R2), if the concept is an operation and is compile-time adaptable, then it can be defined as a virtual (abstract) method (like the Template Method pattern). If, however, the concept is an attribute and it is adaptable, then it can be defined as a mutable attribute (R3). The rule (R4) assumes that if the concept is adaptable and the rules (R1) to (R3) are not applicable, then the concept can be defined as a class. The rule (R5) suggests that if the concept is a fixed attribute, then it can be defined as a constant attribute. According to the rule (R6), if the concept is a fixed operation, then it can be selected as a non-virtual operation. Finally the rule (R7) assumes that if the fixed concept is neither an attribute nor an operation, then it can be represented as a class. Note that the rules (R4) and (R7) are quite similar, since they both select the class abstraction. We could make distinction between these rules by assuming that (R4) and (R7) create mutable and constant classes (objects), respectively. In most object-oriented languages, however, objects are per default mutable. Additional programming effort is necessary to enforce constant objects. We therefore do not make distinction between mutable and constant objects.

These rules considerably simplify the generation of a concept set. Although the possible number of alternative concept sets is $5^{11} = 48828125$, by using the heuristics rules, a concept set can be selected only in 11 steps. Similar to the dialog window shown in Figure 6, if the software engineer selects the button *I don't know*, then the alternative concept sets are

generated. Again, these alternatives can be compared with respect to their adaptability degrees⁷.

We will now illustrate object-oriented concept identification process using the collection classes example. Assume that based on the functions (F7) and (F14), and by using the tool *Alternatives Generator* the software engineer decides on the following adaptability properties:

$$C_{ObjectAdaptLibrary} = ((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, AD, collectionItems), (OP_v, AD, sort), (CL, AD, sortClass), (AT_c, FX, RN), (OP_v, AD, Read), (OP_v, AD, write), (OP_v, AD, CR), (CL, AD, CRClass)) \quad (F15)$$

Here, the adaptable tuples $(AD, Library)$, $(AD, Collection)$, $(AD, sort)$ and (AD, CR) are considered as classes. Note that based on the rule (R1), to represent sort and CR two new tuples are introduced. To distinguish operation and class names, we use the names *sort*, *sortClass*, *CR* and *CRClass*. The adaptable tuples $(AD, read)$ and $(AD, write)$ are represented as virtual operations. The adaptable tuple $(AD, collectionItems)$ is considered as a mutable attribute. The fixed tuples $(FX, LinkedList)$, $(FX, OrderedCollection)$ and $(FX, Array)$ are considered as constant classes. The fixed tuple (FX, RN) is considered as a fixed attribute.

Figure 7: The dialogue used to generate object-oriented concepts

3.5 Identification of the Object-Oriented Relations

We will now consider the relations among concepts. As an example, consider the following table, which represents the relations among the

⁷ This tool is implemented as an expert system, which applies object-oriented heuristics to reduce the design space. Various different heuristics can be defined by using the method engineering facilities of the tool [17].

concepts of $M_{Library}$. This table is derived from the requirement specification and domain analysis⁸.

Table 1: Identification of relations among the concepts of Library

Concept	has a relation with the following concepts
Library	Collection, LinkedList, OrderedCollection, Array
Collection	LinkedList, OrderedCollection, Array
CollectionItems	Collection, LinkedList, OrderedCollection, Array
Sort	Collection, LinkedList, OrderedCollection, Array, SortClass
Sort	RN, read, write, CR
CollectionItems	RN, read, write, CR, sort

The top 4 rows are directly derived from the requirement specification. Here, it is assumed that the concept *Library* contains the collections. The second row indicates that *Collection* is an abstraction of *LinkedList*, *OrderedCollection* and *Array*. The third row indicates the relation between *collectionItems* and the collections. The fourth row represents the relation between sorting and the collections.

The rows 5 and 6 are derived from the sorting domain. The fifth row indicates the relation between the sorting algorithm and range determination, reading, writing and comparison operations. The sixth row represents the relation between *collectionItems* and range determination, reading, writing and comparison operations.

After the identification of the object-oriented abstractions (like in (F14)) additional tuples and therefore new relations may be introduced. For example, in previous section we have replaced $(AD, sort)$ and (AD, CR) by the pair of concepts $(OP, AD, sort)$ and $(CL, AD, sortClass)$ and (OP, AD, CR) and $(CL, AD, CRClass)$. By using the tool *Model Definer* new concepts can be easily introduced at any time. The other related tools are updated automatically.

The relation among the concepts can be represented as a pair of concepts. Consider, for example the following lists of tuples which are derived from Table 1 after the adaptability and object-oriented properties of the concepts have been selected:

⁸ The relations can be derived using any object-oriented analysis method.

$$\begin{aligned} \mathcal{R}_{ObjectAdaptLibrary} = & (((CL, AD, Library), (CL, AD, Collection)), ((CL, AD, Library), (CL, FX, \\ & LinkedList)), ((CL, AD, Library), (CL, FX, OrderedCollection)), ((CL, \\ & AD, Library), (CL, FX, Array)), ((CL, AD, Collection), (CL, FX, \\ & LinkedList)), ((CL, AD, Collection), (CL, FX, OrderedCollection)), ((CL, \\ & AD, Collection), (CL, FX, Array)), ((ATm, AD, collectionItems), (CL, AD, \\ & Collection)), ((ATm, AD, collectionItems), (CL, FX, LinkedList)), ((ATm, \\ & AD, collectionItems), (CL, FX, OrderedCollection)), ((ATm, AD, \\ & collectionItems), (CL, FX, Array)), ((OPv, AD, sort), (CL, AD, \\ & Collection)), ((OPv, AD, sort), (CL, FX, LinkedList)), ((OPv, AD, sort), \\ & (CL, FX, OrderedCollection)), ((OPv, AD, sort), (CL, FX, Array)), ((OPv, \\ & AD, sort), (ATc, FX, RN)), ((OPv, AD, sort), (OPv, AD, read)), ((OPv, \\ & AD, sort), (OPv, AD, write)), ((OPv, AD, sort), (CL, AD, CR)), (OPv, AD, \\ & sort), (CL, AD, sortClass), ((ATm, AD, collectionItems), (ATc, FX, RN)), \\ & ((ATm, AD, collectionItems), (OPv, AD, read)), ((ATm, AD, \\ & collectionItems), (OPv, AD, write)), ((ATm, AD, collectionItems), (CLm, \\ & AD, CR)), ((OPv, AD, CR), (CL, AD, CRClass))) \quad (F16) \end{aligned}$$

Note that the new concepts *sortClass* and *CRClass* are added to the tuple-set as well.

The predefined property $P_{ObjectRelation}$ is a set of object modeling alternatives for relations:

$$P_{ObjectRelation} = (AG, IN, WS, CS, RS, WS) \quad (F17)$$

Here, *AG*, *IN*, *WS*, *CS*, *RS* and *WS*, represent *aggregation*, *inheritance*, *owns*, *calls*, *reads* and *writes* relations. These relations are explained in table 2.

Table 2: Object-oriented relations

TO \ FROM	A. <i>CL</i>	B. <i>OP_v</i>	C. <i>OP_n</i>	D. <i>AT_m</i>	E. <i>AT_c</i>
1. <i>CL</i>	aggregates/ inherits from	owns	owns	owns	owns
2. <i>OP_v</i>	owned by/ inherited from	calls (direct/inh erited)	calls (direct/inherited) /in-lines	reads -writes	reads
3. <i>OP_n</i>	owned by	calls (direct/inh erited)	calls (direct/inherited) /in-lines	reads -writes	reads
4. <i>AT_m</i>	owned by/ inherited from	read- written by	read-written by	derived from	derived from
5. <i>AT_c</i>	owned by	read by	read by	derived from	derived from

In this table, the columns and rows correspond to the object-oriented constructs in a relation set. The elements of the table represent the possible

object-oriented relations. Since most object-oriented relations are directed, we define the direction of the relations from the concepts of the first column to the concepts of the first row. For example, the relation 2D should read as OP_v reads-writes AT_m .

The relation 1A indicates that the possible relations between two classes are aggregate and inheritance relations. The relations 1B to 1E are all *owns* relations because operations and attributes must always belong to a class. In the second row, the relations 2B and 2C specify that OP_v can call on OP_v or OP_n . Sometimes, the calls on fixed operations can be in-lined in the implementation of the calling operation. Similarly, relations 3D and 3E mean that OP_v can read and write on AT_m but can only read from AT_c . The other relations are self-explanatory.

The following relation space $S_{ObjectRelationAdaptLibrary}$ maps the relations of $R_{AdaptLibrary}$ to the elements of $P_{ObjectRelation}$ and as such represents the total set of alternatives of the object-oriented relations in the adaptable library:

$$S_{ObjectRelationAdaptLibrary} :: \rightarrow \{ R_{AdaptLibrary} \rightarrow P_{ObjectRelation} \mid (cond) \} \quad (F18)$$

Here, *cond* represents the restrictions on the possible relations. The restrictions must be derived from the semantics of the application, as defined in Table 1. In addition, as shown in Table 2, in the object-oriented model only a certain kinds of relations are possible.

Figure 8: A tool for generating object-oriented relations

The tool *Object Relation Generator* as shown in Figure 8 helps the software engineer to select the appropriate relation. The top widgets *Relation Sets* and *Object Concept Sets* list the relations and concept sets of the models. The selected concept sets must have their P_{Object} values defined. It is also possible to consider alternative concept sets by selecting an item from the widget *Object Concept Spaces*.

Consider the following example. According to the relation 1.A from Table 2, the first relation ((CL, AD, Library), (CL, AD, Collection)) may have 4 possible implementations: These are "Library aggregates Collection", "Collection Aggregates Library", "Library inherits from Collection" and "Collection inherits from Library". We assume that the software engineer selects the "Library aggregates Collection" relation. If the relations were defined as directed relations, then the tool would only propose 2 object-oriented relations. The tool iterates through all the relations.

Now assume that based on the selections of the software engineer, the following relation set is generated:

$R_{ObjectRelationAdaptLibrary} = (Library\ aggregates\ Collection), (Library\ aggregates\ LinkedList),$
 $(Library\ aggregates\ OrderedCollection), (Library\ aggregates\ Array),$
 $(LinkedList\ inherits\ from\ Collection), (OrderedCollection\ inherits\ from\ Collection),$
 $(Array\ inherits\ from\ Collection), (collectionItems\ owned\ by\ Collection),$
 $(collectionItems\ owned\ by\ LinkedList), (collectionItems\ owned\ by\ OrderedCollection),$
 $(collectionItems\ owned\ by\ Array), (Collection\ owns\ sort), (LinkedList\ owns\ sort),$
 $(OrderedCollection\ owns\ sort), (Array\ owns\ sort), (sortClass\ owns\ sort), (sort\ reads\ RN),$
 $(sort\ calls\ read), (sort\ calls\ write), (sort\ calls\ CR), (RN\ reads\ collectionItems),$
 $(read\ reads\ collectionItems), (write\ writes\ collectionItems), (CR\ reads-$
 $writes\ collectionItems), (CRClass\ owns\ CR) \quad (F19)$

The relations in (F19) can be largely simplified by using object refactoring rules [11]. For example, common aggregation relations, attributes and methods can be moved to their super class, if any. In (F19) the aggregation relations between *Library* and *Collection*, *LinkedList*, *OrderedCollection* and *Array* can be reduced to "Library aggregates Collection" because all other classes inherit from *Collection*. Similarly, the aggregation relations between the collection classes and *sort* can be reduced to "Collection aggregates sort". The "owned by" relations between the *collectionItems* and the collection classes can be reduced to "*collectionItems* owned by *Collection*". Further if an operation is owned by multiple classes and if there is an inheritance relation between these classes, then the

operation can be moved to the super class. If these classes do not inherit from each other, then the method can be replicated in every class. The re-factoring of the object diagram can be advised or largely realized by a tool. In the appendix, the class diagrams of the six selected alternative implementations of *MLibrary* after the re-factoring process are shown.

4. DESIGNING FOR TIME PERFORMANCE

This section introduces a simple process to determine the time performance factors of models at various abstraction levels. These factors can be used to compare the alternative models. The performance analysis process is based on simulation. For this purpose, we have developed a simulation environment and a set of tools. This environment involves a set of random generators, time measurement and display units⁹. The performance analysis process involves the following steps:

1. Construction of the model: To determine the probabilistic time performance-value of a model, first its concepts and relations must be identified. It is possible to determine the relative time performance of a model even before its adaptability and object properties are determined.
2. Identification of the behavioral concepts: We need to identify the concepts that contribute to the behavior of the model. For example in *MLibrary*, the concept *sort* is the most significant concept for the sorting process.
3. Identification of the interaction diagram: An interaction diagram specifies a call pattern among the related concepts. For example, when the operation *sort* is invoked, depending on the sorting algorithm, *sort* calls on other concepts to realize the sorting process. To simulate a model it is necessary to define a sub-graph, which represents the interaction diagram of that model. For example, the sub-graph shown in Figure 9 is the interaction diagram of *MLibrary* for sorting the items in the collection objects. For simplicity, here the direction of calls is not shown.

⁹ The emphasis of this section is not to introduce a new performance analysis technique but adopt an existing technique to compare various design alternatives. In the literature, many performance analysis techniques have been published [15][8].

Figure 9: The interaction of concepts in *MLibrary* for sorting the items

4. Specification of the behavior: The concepts identified in step 2 must be specified. For example, for *MLibrary*, we have searched for various sorting algorithms in the sorting domain [14]. We have then selected the *bubble* and *selection* sort algorithms as the two possible implementations of the sorting process.
5. Implementation of the simulation graph: We have developed a framework to implement the simulation graphs such as the one shown in Figure 9. The nodes of this graph must be specialized within the simulation context. For *MLibrary*, we created 6 nodes and then specialized these nodes according to the graph shown in Figure 9¹⁰.
6. Determination of the simulation parameters: This involves parameterization of the random generators, the range of simulation, etc. For example, we have simulated the interaction diagram of *MLibrary* in 2 different environments. In the first setting, we fixed the number of items in the collection objects to 100, we run the sorting process 100 times, and we randomly generated the items in the collections using the *Linear Congruential Generator* algorithm [12]. In the second setting we changed the number of items in the collections randomly.
7. Determination of the probability values of calls: Using the facilities of the simulation environment, the number of calls per relation must be counted and normalized with respect to the total number of calls. These values represent the probability of calls. Figure 10 shows the tool, which displays the probability values of calls of *MLibrary*. Here, figures 10(a) and 10(b) display the simulation environment for the first and second settings, respectively¹¹.

¹⁰ Depending on the application, various simulation languages can be used [8][15].

¹¹ The purpose of this phase is to obtain reliable probability values. The detailed analysis of this topic is considered beyond the scope of this chapter.

Note that the probability of calls to RN (range calculation) is **very** small and therefore can be neglected. The relations between classes do **not** directly influence the performance and therefore they are set to zero. We **can** now determine the relative performance values of alternative implementations of *MLibrary*, provided that the interaction patterns remain the **same**. As illustrated in the previous sections, a model can be implemented **in** many different ways, and the relative performance factors of models will **depend** on the type of the object-oriented relations used¹². We would like to reason about the alternatives by assigning a time performance value to each **model**.

Figure 10: The frequency of calls in *MLibrary*: (a) with 100 items, (b) with randomly varying number of items.

The relative time performance value of a model is computed in the following way:

$$performance = 100 / \sum p_i r_i \quad (F21)$$

Here, i is the index of the set, where p_i and r_i represent the probability and relative cost values of the indexed element of the set, respectively. The cost calculation must be iterated through the range of the set. We multiply the result with 100 for the scaling purpose. The relative cost values of relations are language dependent. Since our experimental environment is implemented in the Smalltalk language, the following average cost values are measured from our Smalltalk system. These values are normalized with respect to the cost of an in-lined call:

¹² The concepts of a model influence the call relations indirectly, because they determine the type of relations used (Table 2).

Table 3: The relative cost values in the Smalltalk system

Relation type	Cost
message call	5
inherited call	4
inline	1
attribute read	5
attribute write	16

In the previous sections, the probability of calls was assumed to be the same for every alternative model. This assumption is valid if the behavioral concepts of the models remain the same. If these concepts are adaptable, then we need to build a new simulation model for each interaction pattern. For example, in addition to the *bubble sort* algorithm, we also simulated the *selection sort* algorithm. As published in the literature [14], the *selection sort* algorithm generally performs better than the *bubble sort* algorithm. However, if the items in the collection are almost sorted, then the *bubble sort* is faster. Therefore the choice of an algorithm depends on the context of execution. A more detailed discussion about simulating multiple alternative algorithms is given in [17].

5. BALANCING ADAPTABILITY AND PERFORMANCE FACTORS

The appendix of this chapter shows the adaptability and performance values of the six design alternatives of the collection library. We will now compare these alternatives. In Figure 11 the Y-axis and X-axis show the performance and adaptability degrees of the alternative models, respectively. The models are represented as diamonds in the graph. The model numbers are shown at the right of the corresponding diamonds. We would like to emphasize that these models cannot be compared by considering their relative performance and preferred adaptability degrees only. The definition of these models as shown in the appendix must be considered as well.

Model 1 has the highest performance and lowest preferred adaptability value. This is of course an expected result since the implementation of the sort algorithm is in-lined. For the total scale we can observe that a higher preferred adaptability degree eventually results in a lower performance degree. This conclusion, however, is not necessarily valid for models with close preferred adaptability degrees. As it can be derived from Figure 11, Model 2 provides a slightly higher preferred adaptability degree than model

4 but still has a higher performance degree. In Model 2 the comparison criterion is the only adaptable operation whereas the other operations are in-lined. In model 4 two operations are compile-time adaptable, that is read and write, and the comparison operation is in-lined. Thus, in model 4 more components are adaptable than in Model 2. This results in a lower performance for model 4. From this we can conclude the following: To fulfil the flexibility requirements, sometimes a concept must be made run-time adaptable. In this case, it may be still possible to obtain a high performance, if all other concepts are fixed.

Figure 11: The relative cost values of the selected design alternatives

Model 3 is similar to Model 4. In Model 4, the comparison criterion was made run-time adaptable but to compensate the performance loss, all other operations were fixed. In Model 3, however, no such compromise is made. The performance difference between Model 3 and Model 4 shows the penalty paid for making the comparison criterion run-time adaptable without making any compromise.

Model 6 and Model 5 have both four operations adaptable. In Model 6 all the four selected operations are run-time adaptable whereas in Model 5 two of the four operations are compile-time adaptable. Although Model 6 has thus a higher preferred adaptability degree than Model 5 it performs slightly less. The main reason for this is that in Model 5 the operation *sort* is made adaptable whereas in Model 6 this is made fixed. Since this operation has the highest adaptability priority (5) it substantially increases the preferred adaptability degree of model 5. Nevertheless, this operation has a minor effect on the performance degree. This means that if an operation is not executed frequently, making that operation adaptable will cause practically no performance degradation. We can also conclude that run-time adaptability increases the preferred adaptability degree but has a minor additional effect on the performance degree with respect to compile-time adaptability.

The presented results are computed for the *bubble* sort algorithm and the distribution of the collection items to be sorted is the same for all the models. The *bubble* sort algorithm, however, is faster than the *selection* sort algorithm if the items in the collection objects are largely sorted. Model 5 allows run-time substitution of the sorting algorithm and the comparison criterion. The performance degree of this model is 12. If the number of the items to be sorted continuously changes, then this model may have a better average time performance¹³. Obviously, if the probabilistic behavior of the change is known, it is possible to compute when implementations that allow behavioral change provide a better time performance.

To verify the validity of the relative time performance values of the selected models, we have implemented the models 1, 2 and 6. We have generated the collection items using the same simulation environment as shown in Figure 10. The time performance values of the models 1, 2 and 6 were measured as 585, 910 and 1310 milliseconds, respectively. In the following, the ratio of the measured values is compared with the estimated values. We can conclude that the estimated values are reasonably accurate.

measured	1310/585 = 2.2	1310/910 = 1.4	910/585 = 1.6
computed	40/16 = 2.5	40/27 = 1.5	27/16=1.7

From Figure 11, the software engineer can select on the Y-axis an acceptable time performance, say 18, and can determine the affordable degree of preferred adaptability from the X-axis. Note that by using the table above, it is also possible to convert the relative values to the actual time performance factors.

6. RELATED WORK

Adaptability is generally considered as an important and desired characteristic of software systems and a number of research groups have been active in this area. For example, to improve the adaptability characteristics of software systems, the Demeter method [34], Composition-Filters [1], Aspect-Oriented Programming [7], and Reuse Contracts [16] are proposed as extensions to the object-oriented model. We consider these contributions important and complementary to our work. Our emphasis, however, is different. We do not propose an extension the to object-oriented

¹³ Part of the performance penalty in this model is due to run-time adaptability of the comparison criterion.

model, but introduce a technique to compare the design alternatives from adaptability and performance viewpoints.

In [6], the concept of variation point is introduced to specify locations at which variation will occur. The variation points are generally expressed using variants, which are type-like constructs. Although our adaptability modeling approach is intuitively similar, we propose an adaptability model, which can be applied along the software development process for comparing the design alternatives.

Several publications have been made on object-oriented software metrics [4]. Software metrics is quantitative measurements about any aspect of a software project. This may include *project, process and product metrics*. Product metrics aim to determine the properties of the software product, such as the amount of coupling, cohesion, code complexity, etc. Most product metrics as published in the literature are generally determined after the software system is built and there is no clear relation between the quality demands of requirements, compromises being made, and the quality of systems being built.

Simulation techniques have been used in analyzing the performance of software systems for many years [8]. In our example method, we applied a simple simulation technique to determine the relative performance characteristics of the design alternatives. In this chapter our intention is not to introduce a new sophisticated performance analysis method, but rather adopt an existing suitable technique.

During the last decade, the so-called Software Performance Engineering (SPE) discipline has emerged for combining the performance analysis techniques with software engineering methods [15]. This discipline aim to construct performance models of software systems by using data about envisioned software processing. These models are used to compare software and hardware alternatives for solving performance problems. The techniques used within the context of SPE research are relevant to our work, and can be applied together with the techniques presented in this chapter. Our emphasis is to compare the design alternatives both from performance and adaptability viewpoints, whereas the SPE research mainly emphasized the performance factors of the design alternatives.

The techniques presented in this chapter can be considered as a special form of Relational Algebra [5]. Our tools implement operations that are similar to the *union, product, select* and *join* operations of Relational Algebra. The select operation in our case is based on design heuristics. We therefore term our technique as Design Algebra. We are currently applying and formalizing Design Algebra within the context of a large transaction system design [17].

Generally speaking, our work can be classified under the so-called "AI-based problem solving techniques" [9][19]. 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. EVALUATION

For several years, we have been applying Design Algebra and the related tools in various projects [17]. Further, we tutored Design Algebra in various conference tutorials¹⁴ and professional courses. Based on these experiences, we will now evaluate Design Algebra from the perspective of scalability, complexity of the design space, complexity of the process, adaptability and performance analysis.

Scalability: In this chapter we use a rather simple example for illustrative purposes. In practice, however, we applied Design Algebra and the related tools in larger projects such as atomic transaction system design [17]. Currently we are experimenting with Design Algebra in designing quality-aware middleware systems. With respect to scalability, we observe that the techniques presented in this chapter do not necessarily increase the complexity of current object-oriented practices. For example, the only required extensions to the UML models are the introduction of additional attributes for storing the adaptability properties and preferred adaptability values. From methodological point of view, in addition to applying familiar object-oriented design rules, in our case the software engineers have to determine whether a concept should be adaptable or fixed. If desired, the preferred adaptability values for concepts may be defined as well. This does not necessarily complicate the process, since only a few questions have to be answered. In addition, this process is supported by the tools. The performance analysis process is limited to a comparative performance analysis and therefore does not require detailed performance analysis models.

Complexity of the design space: The design space concepts as formulated in (F6), (F8), (F13) and (F18) are purely conceptual. The software engineers do not deal with the total number of design alternatives. In practice, only a relevant number of alternatives, say up to 15, are

¹⁴ The techniques presented in this chapter were partially tutored in ECOOP'98, OOPSLA'99 and ECOOP'2000 conference tutorials.

considered at a time. One may claim that even a limited number of alternatives increase the complexity of design. We think that in any realistic project alternative models are always defined. In general, these models are not managed properly and kept in various files and/or repositories. Our tools keep track of the differences between the alternative models and provide means to compare, select and if necessary eliminate them. We think that our approach may reduce the complexity in dealing with alternatives, which may be hidden in the project environment.

Complexity of the process: The refinement process as presented in this chapter, such as problem understanding, domain analysis, object modeling and implementation is not fundamentally different from the advises of most object-oriented methods. The only additional work is to consider the adaptability values of concepts. For performance analysis we also consider the probability values of interactions.

The software engineer may introduce new concepts at any stage of the process. The tools keep track of the changes. For example, if a new modeling element is introduced in later stage, the tool asks the software engineer if its adaptability characteristics have to be considered. We consider rule-base support as shown in figures 6 and 7 extremely useful during the design process. Further, while gathering the input for rules, the system creates an automatic documentation of the design decisions.

Adaptability analysis: The main purpose of the preferred adaptability values is to express the wishes of the software engineer, to label the alternative models and/or to order the models, if appropriate. The software engineer has always an access to the definition of the priority values and the corresponding object models. The software engineer may select various schemes for calculating the adaptability value of a model such a summing up the values of tuples, giving a weighting value per model, etc. What is more important is the explicit and relative consideration of the adaptability factors of concepts and their effect to the overall model. If the software engineer cannot decide on the adaptability of a concept, he/she may leave it undefined.

Performance analysis: In this chapter we have presented a simple technique to analyze the relative performance of the alternatives. There are a great number of successful techniques for performance analysis and the software engineer should adopt the appropriate one. However, relative performance analysis is generally simpler than a detailed analysis of the system, since only the differences among the models have to be considered.

8. CONCLUSIONS

There are, in general, many correct implementations of a software design problem, and each implementation may differ from the other with respect to its quality factors. Software is rarely designed for ultimate quality, but it is a compromise of multiple considerations. For example, generally the adaptability and performance factors of a software system have to be balanced. To achieve these objectives, in this chapter the following four requirements were considered important: First, to be able to compare the design alternatives, the space of the alternatives must be determined. Secondly, the alternatives must be ordered with respect to their quality factors. Thirdly, the software engineers must be able to select among the alternatives based on the requirements. Finally, the quality factors must be balanced with respect to each other.

In section 3, a process has been presented to explicitly reason about the adaptability factors of the design alternatives at various abstraction levels. In section 4, a simulation technique was used to determine the relative time performance factors of the design alternatives. To this aim, it was found sufficient to build a single simulation model. In section 5, we have analyzed six design alternatives from their adaptability and performance viewpoints. We have shown that the adaptability and time performance factors of the software systems can be balanced with respect to the requirements. Comparing these quality factors was also educational for us and our findings were summarized in section 5.

The techniques presented in this chapter can be considered as a special form of Relational Algebra, which we termed as Design Algebra. We think that the algebraic techniques provide a formal foundation and enable implementation of suitable tools. We also think that the proposed technique is practical since it can be easily integrated with current object-oriented methods. Rule-based heuristics are particularly useful in selecting and evaluating the alternatives. The algebraic techniques can be extended to other quality factors as well. We are currently working on reuse factors of design alternatives.

ACKNOWLEDGEMENTS

This research has been supported by various organizations including Siemens-Nixdorf Software Center, the Dutch Ministry of Economical affairs (SENER), the Dutch Organization for Scientific Research (NWO, 'Inconsistency management in the requirements analysis phase' project), the AMIDST project, and by the IST Project 1999-14191 EASYCOMP.

9. REFERENCES

1. M. Akşit, *Separation and Composition of Concerns*. ACM Computing Surveys 28A(4), December, 1996.
2. M. Akşit, B. Tekinerdoğan, F. Marcelloni, & L. Bergmans. *Deriving Object-Oriented Frameworks from Domain Knowledge*. In Building Application Frameworks: Object-Oriented Foundations of Framework Design, Fayad et al. (eds), Wiley, 2000.
3. G. Arrango. *Domain Analysis Methods*. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49, 1994.
4. S.R. Chidamber, & C. F. Kemerer. *A Metrics Suite for Object-Oriented Design*. IEEE Transactions on Software Engineering 20(6): 476-93, 1994.
5. C. Date. *An Introduction to Database Systems*. Addison-Wesley, 1986.
6. I. Jacobson et al. *Software Reuse*. ACM Press, New York, 1997.
7. G.Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier & J. Irwin, *Aspect-Oriented Programming*. ECOOP '97 Conference Proceedings, LNCS 1241, , pp. 220-242, Springer-Verlag, 1997.
8. A. M. Law & W. D. Kelton. *Simulation Modeling & Analysis*. Second Edition, McGraw-Hill, Inc., 1991.
9. C-L. Lee, G. Iyengar & S. Kota. *Automated Configuration Design of Hydraulic Systems*. In: Artificial Intelligence in Design'92, (Ed) J. S. Gero, pp. 61-82, Kluwer Academic Publishers, 1992.
10. K.J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
11. W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. University of Illinois, Urbana Champaign, 1992.
12. W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling. *Numerical Recipes*. Cambridge University Press 1986, pp. 191-199, 1986.
13. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, & W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.
14. R. Sedgewick. *Algorithms*. Addison-Wesley, 1988.
15. C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley, 1990.
16. P. Steyaert, C. Lucas, K. Mens, & T. D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. OOPSLA '96 Proceedings, ACM SIGPLAN Notices, pages 268-285, ACM Press, 1996.
17. B. Tekinerdoğan. *Synthesis-Based Software Architecture Design*. PhD Thesis, Dept. of Computer Science, University of Twente, March 23, 2000.
18. B.Tekinerdoğan & M. Akşit. *Adaptability in object-oriented software development: Workshop report*, In M. Muhlhauser (ed), Special issues in Object-Oriented Programming, Dpunkt, Heidelberg, 1997.
19. C. Tong & D. Sriram. *Introduction*. In: Artificial Intelligence in Engineering Design, Vol. 1, (Eds) C. Tong & D. Sriram, pp. 1-53, Academic Press, 1992.

APPENDIX MODELS

MODEL 1

All tuples are fixed; no practical use of inheritance.

Tuple:

$CoobjectAdaptLibrary = ((CL, FX, Library), (CL, FX, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, FX, collectionItems), (OP_n, FX, sort), (AT_c, FX, RN), (OP_n, FX, Read), (OP_n, FX, Write), (OP_n, FX, CR))$

Adaptability degree: 0

Performance Degree: 40 (measured value 585ms)

MODEL 2

CR run-time adaptable, operations are non-virtual.

Tuple:

$CoobjectAdaptLibrary = ((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, AD, collectionItems), (OP_n, FX, sort), (AT_c, FX, RN), (OP_n, FX, Read), (OP_n, FX, Write), (OP_v, AD, CR1), (CL, AD, CRClass))$

Adaptability Degree: 11

Performance Degree: 27 (measured value 910ms)

MODEL 3

CR run-time adaptable, sort is fixed but inherited.

Tuple:

$C_{ObjectAdaptLibrary} = ((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array),$
 $(AT_m, AD, collectionItems), (OP_m, FX, sort), (AT_o, FX, RN), (OP_v, AD, Read), (OP_v, AD, Write),$
 $(OP_v, AD, CR), (CL, AD, CRClass))$

Adaptability Degree: 17**Performance Degree: 18** **MODEL 4**

Sort is fixed but inherited. All other methods are virtual.

Tuple:

$C_{ObjectAdaptLibrary} = ((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array),$
 $(AT_m, AD, collectionItems), (OP_m, FX, sort), (AT_o, FX, RN), (OP_v, AD, Read), (OP_v, AD, Write), (OP_m, FX, CR))$

Adaptability Degree: 9**Performance Degree: 19**

MODEL 5

Sort and CR run-time adaptable. All others methods are virtual

Tuple:

$C_{ObjectAdaptLibrary} = ((CL, AD, Library), (CL, AD, Collection), (CL, FX, LinkedList), (CL, FX, OrderedCollection), (CL, FX, Array), (AT_m, AD, collectionItems), (OP_v, AD, sort), (CL, AD, sortClass), (AT_c, FX, RN), (OP_v, AD, Read), (OP_v, AD, Write), (OP_v, AD, CR), (CL, AD, CRClass))$

Adaptability Degree: 27

Performance Degree: 12

MODEL 6

RN, CR, read and write run-time adaptable. Sort is fixed.

Tuple:

$C_{ObjectAdaptLibrary} = ((CL, AD, Library), (CL, AD, Collection), (CL, AD, LinkedList), (CL, AD, OrderedCollection), (CL, AD, Array), (AT_m, AD, collectionItems), (OP_v, FX, sort), (OP_v, AD, RN), (CL, AD, RNClass), (OP_v, AD, Read), (OP_v, AD, ReadClass), (OP_v, AD, Write), (OP_v, AD, WriteClass), (OP_v, AD, CR), (CL, AD, CRClass))$

Adaptability Degree: 30

Performance Degree: 16 (measured value 1310ms)