

Evaluating Architecture Implementation Alternatives based on Adaptability Concerns

Mehmet Aksit and Bedir Tekinerdogan
TRESE project, CTIT and Department of Computer Science,
University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.
email: {aksit, bedir}@cs.utwente.nl
<http://www.trese.cs.utwente.nl>

Abstract

Software is rarely designed for ultimate adaptability, performance or reusability but rather it is a compromise of multiple considerations. Even for a simple architecture specification, one may identify many alternative implementations. This paper makes an attempt to depict the space of implementation alternatives of architectures, and to define rules for selecting them. The applicability of this approach is illustrated by means of a simple design problem.

1. Introduction

Software systems have to cope with continuously changing requirements. If a software system is derived from a well-defined architecture specification, then the effect of changes in the requirement specification will be limited to the boundaries of the abstractions of the architecture. A well-defined architecture can be specified as a set of abstractions and relations which form a concept. A concept is a fundamental abstraction in a given domain, which is useable and inherently complex. Architectures can be defined at various phases of the software development process, for example, domain architecture refers to the fundamental abstractions in the background knowledge, product-line architecture specifies a family of products to be produced, and a system architecture refers to the software and/or hardware components.

Since architecture specifications are generally abstract, one may create various different implementations for the same architecture specification. Each alternative will have different adaptability, performance and reusability characteristics. Providing ultimate adaptability may create too much run-time overhead. Aiming at fastest implementation may result in unnecessarily rigid software. Aiming at most reusable software may introduce redundant abstractions for a given problem. Software

engineers, therefore, must be able to explicitly compare, evaluate and decide on various implementation alternatives of architectures based on the relative importance of the required quality factors.

This paper introduces a new formalism to depict the space of alternative implementations of architectures, and rules to select among the alternatives based on quality factors. This paper mainly focuses on the adaptability concern.

2. An Example Problem

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. Applying object-oriented techniques may result in the object model shown in Figure 1.

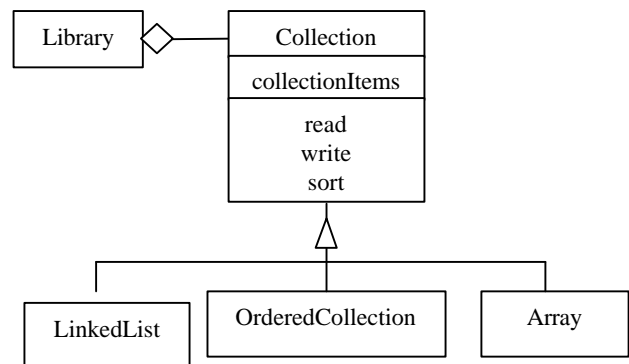


Figure 1. An object diagram for the collection classes.

Here classes *Library*, *LinkedList*, *OrderedCollection*, and *Array* are identified by scanning the nouns in the requirement specification. Since these classes share the same abstract behavior, 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 object-oriented heuristics.

The object model presented in figure 1 is one of the many possible implementations. For example, the sorting operation might be defined as a part object of class *Collection*. This would allow, for example, changing the sorting operation at run-time. One might also prefer to change part of the sorting algorithm, for example the sorting criterion. There are, of course, a considerable number of alternatives, depending on the granularity of the required changes, and whether these changes must be realized at compile-time or run-time.

To be able to reason about these possible implementations, we need to identify the concepts of the problem. Various domain analysis methods [1] have been introduced to determine the fundamental concepts of a domain. These methods derive the concepts from the common abstractions, which are discovered by analyzing the related knowledge. The architectural abstractions can be derived from the identified concepts [2].

We may discover the concepts of the sorting domain by analyzing and comparing the well-known sorting algorithms. After comparing the 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. Typical examples are *selection*, *insertion*, *bubble* and *quick* sort algorithms. In the following, we list the concepts of a sorting process:

$$M_{Sort} = (AlgT, RN, RD, WR, CR) \quad (EQ 1)$$

Here, M_{Sort} is a model of the sorting domain and $AlgT$, RN , RD , WR and CR are the fundamental concepts and correspond to the algorithm type, range, reading, writing and the comparison criterion, respectively. We will now combine these concepts with the entities of the requirement specification. The result is shown in the following:

$$M_{Library} = (Library, Collection, LinkedList, OrderedCollection, Array, collectionItems, Sort, RN, READ, WRITE, CR,) \quad (EQ 2)$$

Here, the entities $SORT$, $READ$ and $WRITE$ correspond to $AlgT$, RD and WR , respectively. The important questions

to be answered are: what are the possible implementations of these concepts, and how do we compare the implementations based on their quality factors?

3. Modeling Adaptability

Adaptability is the ease of changing an existing model to new requirements. M_{Adapt} is a model of the concern *adaptability* and is defined as a set consisting of the elements of type fixed (FX) and type adaptable (AD):

$$M_{Adapt} = (FX, AD) \quad (EQ 3)$$

We can now apply the adaptability model to the elements of the requirement specification. For example the space $S_{AdaptLibrary}$ is the set of all elements of an adaptable collection library and can be computed as the Cartesian product of the sets M_{Adapt} and $M_{Library}$:

$$M_{Adapt} \times M_{Library} = (FX, Library), (FX, Collection), (FX, LinkedList), (FX, OrderedCollection), (FX, Array), (FX, collectionItems), (FX, Sort), (FX, RN), (FX, READ), (FX, WRITE), (FX, CR), (AD, Library), (AD, Collection), (AD, LinkedList), (AD, OrderedCollection), (AD, Array), (AD, collectionItems), (AD, Sort), (AD, RN), (AD, RD), (AD, WR), (AD, CR), \quad (EQ 4)$$

We now introduce the operation *adapt*, which is used to determine whether the concepts of the sorting model must be fixed or adaptable:

$$adapt: S_{AdaptLibrary} \otimes M_{AdaptLibrary} \quad (EQ 5)$$

Here, $M_{AdaptLibrary}$ is a model of an adaptable sorting domain and is a selection from the space $S_{AdaptLibrary}$. The sign “ \rightarrow ” represents the selection process, which may generate $11 \times 2 = 22$ different adaptable sorting models. The function *degreeAdapt* assigns an integer value to a model based on the priorities of that model.

$$degreeAdapt(M_{AdaptLibrary}) \otimes value \quad (EQ 6)$$

It is possible to order the adaptable library models based on their adaptability degrees. For example, the adaptability degrees can be computed by adding up the priority values of the tuples. This means that the model with all its tuples tagged as adaptable has the highest adaptability degree. The least adaptable model is the one with all its elements are fixed.

4. Object Models

Let us assume that M_{Object} represents the object model:

$$M_{Object} = (CL, OP, AT) \quad (EQ 7)$$

Here, M_{Object} consists of the elements CL , OP and AT , which represent classes, operations and attributes, respectively. We can further classify classes as mutable (CL_m) and constant classes (CL_c), operations as virtual (OP_v) and not virtual (OP_n) operations, attributes as mutable (AT_m) and constant attributes (AT_c), etc. Objects created from the mutable classes and mutable attributes can change at run-time. Virtual operations can be overridden through inheritance. However, constant classes and attributes cannot change at run-time.

The space of an object-based model of the adaptable library $M_{AdaptLibrary}$ can be computed by using the following equation:

$$S_{ObjectAdaptLibrary} = M_{Object} \times M_{AdaptLibrary} \quad (EQ 8)$$

An object-based model of adaptable collection library can be derived from the space $S_{ObjectAdaptLibrary}$ using the operation *objectify*:

$$objectify: S_{ObjectAdaptLibrary} \textcircled{R} M_{ObjectAdaptLibrary} \quad (EQ 9)$$

In this example, the space $S_{AdaptLibrary}$ has 11 elements and therefore there are $11 \times 3 = 33$ possible tuples and $3^{11} = 177147$ adaptable object collection library models. Since there are many possible object models, the software engineer needs heuristic rule assistance. The following heuristic rules are then applied:

IF THE TUPLE IS ADAPTABLE:

- (R1) **IF** RUN-TIME ADAPTABILITY IS REQUIRED, **THEN** DEFINE IT AS A MUTABLE OBJECT (CL_m);
- (R2) **IF** COMPILE-TIME ADAPTABILITY IS REQUIRED, **THEN** DEFINE IT AS A VIRTUAL OPERATION (OP_v);
- (R3) **IF** THE TUPLE HAS A VALUE, **THEN** DEFINE IT AS A MUTABLE ARGUMENT (AT_m).

IF THE TUPLE IS FIXED:

- (R4) **IF** THE TUPLE IS AN AUTONOMOUS CONCEPT, **THEN** DEFINE IT AS A CONSTANT OBJECT (CLASS (CL_c));
- (R5) **IF** THE TUPLE HAS A VALUE, **THEN** DEFINE IT AS A CONSTANT ATTRIBUTE (AT_c);
- (R6) **ELSE** DEFINE IT AS AN OPERATION (OP_n).

The rule (R1) is used to transform a run-time adaptable tuple to a mutable object, so that it can be changed dynamically. According to the rule R(2), if the tuple being considered has to be compile-time adaptable, then it can be defined as a virtual (abstract) method. If, however, the tuple has a value and it should be adaptable, then it can be defined as a mutable attribute (Rule(3)). If the tuple is a fixed autonomous concept, then the rule (R4) suggests that this tuple can be represented as a constant. If the tuple has a fixed value, then it can be defined as a constant attribute (R(5)). If the tuple is fixed and the rules R(4) and R(5) are not true, then it can be selected as an operation. Note that these rules considerably simplify the

generation of a model. Assume that using EQ 5, the software engineer selects the following model:

$$M_{AdaptLibrary} = (AD, Library), (AD, Collection), (FX, LinkedList), (FX, OrderedCollection), (FX, Array), (AD, collectionItems), (AD, Sort), (AD, RN), (AD, RD), (AD, WR), (AD, CR) \quad (EQ 10)$$

Further, using EQ 9, the software engineer makes the following decisions: The adaptable tuples ($AD, Library$), ($AD, Collection$) and (AD, CR) are considered as run-time adaptable entities. The adaptable tuples (AD, RD) and (AD, WR) are compile-time adaptable entities. The adaptable tuples ($AD, collectionItems$) and (AD, RN) have values. The fixed tuples ($FX, LinkedList$), ($FX, OrderedCollection$) and ($FX, Array$) are considered as autonomous concepts. The adaptable tuple ($AD, Sort$) is considered as a run-time adaptable entity. By applying the rules R(1) to R(6), the following model is selected:

$$M_{ObjectAdaptLibrary} = (CL_m, AD, Library), (CL_m, AD, Collection), (CL_c, FX, LinkedList), (CL_c, FX, OrderedCollection), (CL_c, FX, Array), (AT_m, AD, collectionItems), (CL_m, AD, Sort), (AT_m, AD, RN), (OP_v, AD, RD), (OP_v, AD, WR), (CL_m, AD, CR) \quad (EQ 11)$$

5. Modeling Relations

Relations can be defined at various abstraction levels. For example, the following list represents the relations among the elements of $M_{Library}$, which is derived from the requirement specification and domain analysis:

Library	Collection, LinkedList, OrderedCollection, Array
Collection	LinkedList, OrderedCollection, Array
CollectionItems	Collection, LinkedList, OrderedCollection, Array
Sort	Collection, LinkedList, OrderedCollection, Array
Sort	RN, RD, WR, CR
CollectionItems	RN, RD, WR, CR

Table 1. Relevant relations of the example problem.

In Table 1 the first 4 rows are directly derived from the requirement specification. The element *Library* contains *Collection*, *LinkedList*, *OrderedCollection*, *Array*. 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.

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 comparing operations. The sixth row represents the relation between *collectionItems* and range determination, reading, writing and comparing operations.

Object relation types can be summarized as *inheritance*, *aggregation* and *message passing* relations. Once the elements are known, the object relation types can be inferred. Consider now the collection library example. As shown by Figure 2, we have now enough information to infer an implementation that fulfils the model defined by EQ 11 and Table 1.

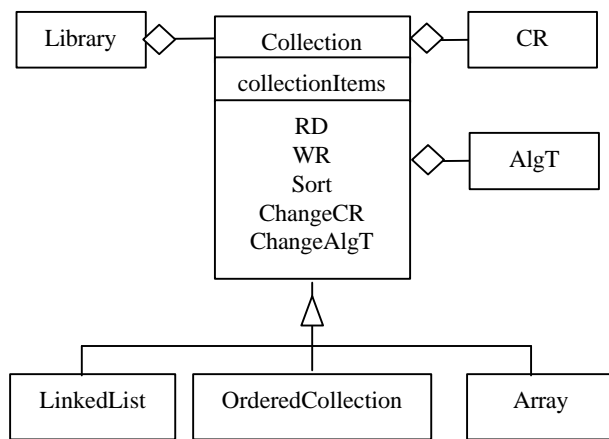


Figure 2. A modified object model.

In Figure 2, class *Collection* has two part classes *CR* and *AlgT*, which implements comparison criterion and algorithm type, respectively. According to EQ 11, these two elements had to be implemented as mutable classes. To change these implementations, the operations *ChangeCR* and *ChangeAlgT* are defined. Of course, the implementation shown in Figure 2 is only one of the many possible implementations of the example problem. This model is a result of series of decisions where the adaptability concern is explicitly taken into account. This model implements a combination of 2 Strategy patterns, where classes *CR* and *AlgT* are the strategies. The operation *Sort* forwards the call to *AlgT*, which makes a series of calls on classes *Collection* and *CR*.

6. Tools for Selecting Alternatives

To be able to support the techniques presented in this paper, we have developed a number of tools. As an example, in Figure 3, the tool *ModelGenerator* is shown. This tool is used to generate and compare models based on their adaptability characteristics. The top-left window

Model Space shows the names of the spaces in the repository. In Figure 3, the space *AdaptSort* is selected. Once a space is selected, then its tuples are displayed in the window *Tuple Space*. The tool checks the consistency of the selections. For example, the software engineer is not allowed to select (FX, AlgT) and (AD, AlgT) at the same time. In the windows *Model no*, *Model Elements* and *Degree*, model labels, tuples of the selected model, and the adaptability degrees of the models are shown. The model numbers are generated by the tool for the labeling purpose. The adaptability degrees are computed by the tool. As displayed by the bottom window, the models are ordered based on their adaptability degrees. The software engineer may refer to this window to select one of the alternatives.

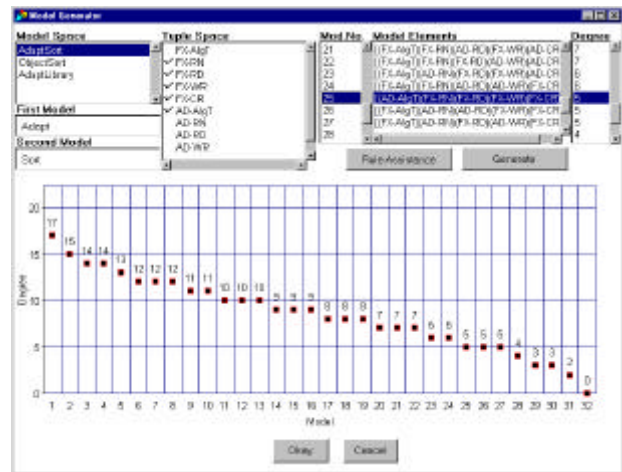


Figure 3. Comparing models based on their adaptability characteristics.

7. Conclusions

This paper presented a technique to generate and compare the implementations of an architecture based on their adaptability characteristics. This technique can be considered as a form of Relational Algebra (which we term as Design Algebra). Currently we are extending this technique for expressing other quality factors such as speed performance and reusability.

References

- [1] 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.
- [2] M. Aksit, B. Tekinerdogan, F. Marcelloni, & L. Bergmans. *Deriving Object-Oriented Frameworks from Domain Knowledge*. To be published as chapter in M.

Fayad, D. Schmidt, R. Johnson (eds.), Object-Oriented Application Frameworks, Wiley, 1998.