Chapter 5

DESIGN OF ASPECT BASED LANGUAGE LIBRARY MODEL FOR MULTILINGUAL SOFTWARE

5.1 Introduction

Multilingual software development can be made systematic with the introduction of formal models discussed in the previous chapter. These models enable the developers to apprehend the design of multilingual components involved in the multilingual software. Suitable models can be applied to the multilingual software development.

Qualities are the essential requirement of software. Realization of qualities has to be planned from the design phase of software development process. Similarly, multilingual software has qualities from the language perspectives. Multilingual software qualities (which are defined in chapter 3) are expected from the mined models of multilingual software. But these mined models do not fully exhibit these qualities. Therefore, a new model confirming the qualities of multilingual software has to be proposed.

The requirements of multilingual software have to be analyzed using the design space approach in order to find out the factors which limit the qualities of multilingual software. Aspect-based language library model, a new model, is proposed to overcome the limiting factors. This proposed model is represented using an algebraic structure. The qualities of the proposed model are analyzed for its effectiveness.

Next section briefs about the design space approach. An analysis of multilingual software using design space approach is described in this chapter. An algebraic structure defined for the proposed model and the operations on the structures of the proposed model are stated. Discussion about the qualities of the proposed model is also presented.
5.2 Design Space Approach

According to Lane (Lane, 1990), a design space identifies the key functional and structural dimensions used to create a system design in a specific application domain. The design space is multidimensional, and each dimension describes a variation in one system characteristic or requirement. Values along a dimension correspond to different design alternatives. Rules are used to encapsulate design knowledge in the form of relationships between alternatives in different dimensions of the design space.

The dimensions of a design space may be functional or structural in nature. Functional dimensions describe aspects of the problem space as functional- and performance-related design alternatives. Structural dimensions describe the solution space as a set of alternative implementation characteristics.

Some dimensions of the design space are related to each other and some are independent. The relationships between dimensions is described as a set of rules and that can be categorized as those linking functional to structural dimensions, and those interconnecting structural dimensions. The first category allows system requirements to drive a structural design, while the latter ensures the internal consistency of the design.

The design space is useful in its own right as a shared vocabulary for describing and understanding systems. The central concept of a multidimensional design space is to classify the system architectures. Each dimension of a design space describes variation in system characteristics or design choices. Values along a dimension correspond to alternative requirements or design choices. The different dimensions are not necessarily independent; in fact, it is important to discover correlations between dimensions, in order to create design rules describing appropriate and inappropriate combinations of choices.

A key part of the design space approach is to choose some dimensions that reflect requirements or evaluation criteria (function and/or performance), while other dimension reflect structure (or other available design choice) and the correlation...
found among such dimensions. It can provide direct design guidance with which
design choices are most likely to meet the functional requirements for a new system.

The dimensions that describe functional and performance requirements make
up the functional design space while those that describe the structural choices make
up the structural design space. These groupings can be regarded either as independent
space or as sub spaces of a single large design space in the context of a stepwise
model of the software design process. The functional design space represents the
results of the requirements analysis and consolidates the functional design steps and
the structural design space represents the results of initial system decomposition.

A dimension that represents a structural choice is likely to have discrete set of
possible values, which may or may not have any meaningful ordering. For example,
methods of specifying user-interface behavior include state-transition diagrams,
context-free grammars, menu trees and many others. Each of these techniques has
many small variations. So one of the key problems in constructing the design space is
finding the most useful granularity of classification even when a dimension is in
principle, continuous (e.g. a performance number), one may choose to aggregate it
into a few discrete values (e.g. “Low”, “Medium”, “High”)

An obvious limitation of this approach is that it may not yield much clarity
about the new structures. A more serious objection about design space is that
important dimensions may be overlooked. It is difficult to demonstrate that a given
design space covers everything that may be of interest at a particular level of
abstraction. Obviously a practical design space can not cover all possible ways of
looking at a software system, so some such restrictions are necessary. The
experimental results suggest that functional and structural dimensions associated with
design methods may need to be added to the present space to overcome the limitation.

On the other hand, experience with this space suggests that leaving the
extraneous dimensions is important and also difficult. If ‘m’ functional dimensions
originally defined for the space, then ‘n’ (where ‘n’ is less than ‘m’) functional
dimensions have significant impact and the other functional dimensions have
considerably less influence. Hence, the less influence dimensions should have been
omitted entirely. In the structural dimensions, it is proved possible to omit many design choices because they turned out to be closely correlated design choices that were retained. For example, the classification of application-interface abstraction level was sufficient to predict many properties of that interface. Hence, those properties need not be represented by separate dimensions.

5.3 Analyzing the Multilingual Software Using Design Space Approach

The stakeholders' requirements drive the design of multilingual software. They possess some factors which limits the qualities of multilingual software. So, the requirements of multilingual software have to be analyzed to find out the limiting factors. In order to analyze the existing multilingual software the design space approach is used. Multilingual software have to be designed based on the requirements. Each requirement has to be modeled in the required set of languages. These two aspects are considered to form the design space in order to analyze the design process.

5.3.1 Design Space for Language Components

Consider the software design space $D$ which will help to understand and design the multilingual software. Software will have a set of requirements say

$$ R = \{r_1, r_2, r_3, \ldots, r_n\}, \text{ where } 'a' \text{ is the number of requirements.} $$

Since the software is designed to work with multiple languages, the software should handle a set of languages $L$. So

$$ L = \{L_1, L_2, \ldots, L_d\}, $$

where '$d$' is the number of languages. Each requirement $R$ for a language can be met with a component $C_{RL}$, as shown in Figure 5.1.
The component $C_{RL}$ is designed to meet the requirement $R$ in the language $L$. Extending this concept to multiple languages and the requirements related to language set $L$ the design space will be as shown in Figure 5.2.

Figure 5.2: Design Space for Language Component Set $C_{RL}$

The requirements may be of functional requirements and/or non-functional requirements. Therefore the components set ‘$C$’ is represented as:

$$
C = \begin{bmatrix}
C_{r1L1} & C_{r2L1} & \cdots & C_{rL1} \\
C_{r1L2} & C_{r2L2} & \cdots & C_{rL2} \\
\vdots & \vdots & & \vdots \\
C_{r1Ld} & C_{r2Ld} & \cdots & C_{rLd}
\end{bmatrix}
$$
From the above component set $C$, it is seen that $C$ grows by the introduction of either requirements or languages. Hence the size of the component set limits the maintainability. If there is any change in the language concern of a language (like keyboard layout) it will affect the components (of that language) which use that particular language concern. Also, this scenario in the multilingual software development decreases the maintainability.

Therefore, design of language component is driven by the language concerns. These language concerns are addressing any one of the issues like language implementation issues (e.g., Coding Scheme, Keyboard Layout, etc.), linguistic issues (e.g., Character set, Grammar, etc.) and domain level language issues (e.g., Indexing, Parsing, etc.). These concerns crosscut in each language component and it is called as language aspects. Each language aspect is realized with different alternatives. These alternatives are supported by different communities and also no standard or benchmark is available to say that which alternative is better one. Designer has the option to implement either complete set of alternatives or selected set of alternatives. This decision will make the multilingual software open to either all communities or limited to few. When the designer chooses to implement the entire set of alternatives of language aspects, he ends up with the larger size of language component library. This reduces the maintainability of the multilingual software. On the other hand, if the designer limits the alternatives of language aspects to implement, then the usability of the software is restricted. Eventually, this situation makes the design process much more complex.

In order to overcome the above said issues, the language aspects have to be analyzed and they have to be considered in the design of multilingual software development. The language specific components will have complexity due to the aspects at three different levels. They are

- Linguistic level language aspects
- Implementation level language aspects
- Domain level language aspects
5.3.1.1 Linguistic level Language Aspects

Any language has its own character set in script form and phonetic form, syntax rules and semantic rules. These entities are called as linguistic level language aspects. The character set and rules may differ between language communities. Moreover modernization of language will alter the grammar rules and sometimes the character set too. Government also alters the character set and grammar rules in order to satisfy the society. These scenarios introduces complexity in designing the languages in computer with various aspects like character set, syntax rules, semantic rules, etc.

5.3.1.2 Implementation level Language Aspects

When a language is implemented, various implementation level language aspects have to be addressed. These aspects are

- Coding Schemes
- KB Layout
- Display standards
- Output standards, etc.,

Multilingual software developers follow their own choice of these aspects. Their selection is due to the familiarity and/or simplicity of the implementation aspects. Efforts have started to standardize these aspects, which ultimately has to lead to universal standards. Until the universal standards are established, these aspects will be supported by different implementations and it introduces the complexity in the design process.

5.3.1.3 Domain Level Language Aspects

Today the software systems are aimed at customer-centric than process-centric. The customers of business domain have their own requirements and demands. Multilingual software is one of their domain requirements. On the other hand internet-enabled and pervasiveness of the software introduces much more demands in multilingualism. Domain level language aspects are also a part of the
demands that are to be addressed. This makes the design process more complex. The domain level language aspects are

- Domain based speech recognition,
- Domain based parsing,
- Domain based indexing etc

5.3.2 Design Space with Language Aspects for Language Component

The complexity in the design of multilingual software adds up due to the aspects of multilingual software requirements. In order to address the complexity in the design of multilingual software due to aspects, a design space has been designed with aspects as the dimensions of the design space. In the design space, linguistic level language aspects, implementation level language aspects and domain level language aspects for each multilingual requirement are represented in $X$, $Y$ and $Z$ dimensions respectively. $C_{LID}$ is the component represented in the Figure 5.3 which addresses the language aspects namely $L$ – Linguistic level language aspect, $I$-Implementation level aspect and $D$- Domain level language aspect.

![Figure 5.3: Design Space for Language Component $C_{LID}$](image)
Considering the design space given in Figure 5.3 with $m$-linguistic level language aspects, $n$-implementation level language aspects and $o$-domain level language aspects, design of multilingual software components have the dimensions of design space as $m \times n \times o$ aspects which is quite large and hence complex. On the other hand, multilingual components need not have single aspect from each dimension. So, combinations of the aspects required for the design of multilingual software components make the multilingual component set quite larger. Considering the language as another dimension, design of multilingual software is still more difficult. Moreover, the high coupling between the structures of various aspects of the multilingual system ends up with restricted reusability.

To overcome the above said difficulties, the aspects of the multilingual software are separated by designing separate space for each type of aspects. Each aspect space is designed with an intent to address a set of components addressing the respective aspect requirements. Language aspect components are designed based on the aspect spaces. Further, a language component can be designed by compositing the required aspect components. This composition can be made at design-time or run-time based on the requirements. This approach reduces the coupling between the types of aspects and also simplifies the design of language components.

5.3.2.1 Linguistic level Language Aspect Space

Considering only the linguistic level language aspects a 2-dimensional design space is designed as shown in Figure 5.4.

Where $l_1$ to $l_m$ are linguistic level language aspects, $L_1$ to $L_d$ are the languages in the multilingual software and $C_{l_1, L_i}$ is the component which meets the $l_1$-th linguistic level language aspect of language $L_i$.
5.3.2.2 Implementation level Language Aspect Space

Considering only the implementation level language aspects, a 2-dimensional design space is designed as shown in Figure 5.5.

Where $l_1$ to $l_n$ are implementation level language aspects, $L_1$ to $L_d$ are the languages in the multilingual software and $C_{u,v}$ is the component which meets the $u$-th implementation level language aspect of language $L_v$. 

---

**Figure 5.4: Linguistic level Language Aspect Space**

**Figure 5.5: Implementation level Language Aspect Space**
5.3.2.3 Domain level Language Aspect Space

Considering only the domain level language aspects, a 2-dimensional design space is designed as shown in Figure 5.6.

Where $d_1$ to $d_o$ are domain level language aspects, $L_i$ to $L_d$ are the languages in the multilingual software and $Cd_{dLy}$ is the component which meets the $d_s$-th domain level language aspect of language $L_y$.

![Figure 5.6: Domain level Language Aspect Space](image)

5.4 Algebraic Structure For Language Aspects

Above described aspect spaces reduce the complexity in designing the language components which were limited by the alternatives of the language aspects. Therefore, the language aspect components have to be used for building language components. This design process is modeled with the algebraic structures.

Let two types of sets are defined to model the language aspects. Let $S$ be a simple set comprising of components only as elements.
$S=\{c_1,c_2,c_3,\ldots,c_n\}$ is an example of simple set with $n$ components

Let $C$ is the composite set comprising of both components and simple or composite sets as elements:

$C = \{ c_i, C_i, S_i \}$ is an example of composite set which comprises of $c_i$ of a simple component, $C_i$ of a composite set and $S_i$ of a simple set. This structure can be used to build the language components

5.4.1 Algebraic Structure for linguistic level Language Aspects

From the design space shown in Figure 5.4, the following set can be easily derived in a generic form

Let the simple set for linguistic level language aspects,

$$S_L = \{C_1, C_2, \ldots, C_m\}$$

Where $S_L$ is the simple set for linguistic level language aspects and $m$ is the number of linguistic level language aspects

Let $S_{1L} = \{C_{11}, C_{12}, \ldots, C_{1m}\}$

Where $S_{1L}$ is the simple set for linguistic level language aspects of language 1. Similarly there are simple sets $S_{1L}$ to $S_{pL}$ where $p$ is the number of languages, that are composited together to form a composite set

Hence, a composite set for linguistics level language aspects is represented as

Let $C_L = \{S_{1L}, S_{2L}, S_{3L}, \ldots, S_{pL}\}$.

5.4.2 Algebraic Structure for Implementation level Language Aspects

From the design space shown in Figure 5.5, the following set can be easily derived in a generic form.
Let the simple set for implementation level language aspects,

\[ S_I = \{C_1, C_2, \ldots, C_i, \ldots, C_n\} \]

Where \( S_I \) is the simple set for implementation level language aspects and \( n \) is the number of linguistic level language aspects

Let \( S_{II} = \{C_{1I}, C_{12}, \ldots, C_{iI}, \ldots, C_{nI}\} \)

Where \( S_{II} \) is the simple set for implementation level language aspects of language \( I \). Similarly there are simple sets \( S_{II} \) to \( S_{pl} \), where \( p \) is the number of languages, that are composited together to form a composite set. So, a composite set for implementation level language aspects is represented as

Let \( C_I = \{S_{1I}, S_{2I}, S_{3I}, \ldots, S_{pI}\} \)

5.4.3 Algebraic Structure for Domain level Language Aspects

From the design space shown in Figure 5.6, the following set can be easily derived in a generic form

Let the simple set for domain level language aspects,

\[ S_D = \{C_1, C_2, \ldots, C_i, \ldots, C_o\} \]

Where \( S_D \) is the simple set for domain level language aspects and \( o \) is the number of domain level language aspects.

Let \( S_{I_D} = \{C_{1I}, C_{12}, \ldots, C_{iI}, \ldots, C_{1o}\} \)

Where \( S_{I_D} \) is the simple set for domain level language aspects of language \( I \). Similarly there are simple sets \( S_{I_D} \) to \( S_{pD} \), where \( p \) is the number of languages, that forms a composite set. So, a composite set is represented as

Let \( C_D = \{S_{1D}, S_{2D}, S_{3D}, \ldots, S_{pD}\} \)
5.5 Operations

For the above defined structures, a set of operations are defined to maintain it. These operations will help in implementing the proposed structure comprehensively. These operations also help in demonstrating the construction of the proposed structures which will increase the understandability about the language aspect structures.

5.5.1 Operations on Basic Structure

In order to maintain the proposed structure, following operations are defined.

Add a component

\[ S' \leftarrow \text{add}(S,c) \]

Where \( S \) and \( S' \) are the simple sets before and after the addition of the component \( c \), respectively

Delete a Component

\[ S' \leftarrow \text{delete}(S,c) \]

Where \( S \) and \( S' \) are the simple sets before and after the deletion of the component \( c \), respectively

Modify a component

This can be implemented using add and delete operations

Member Check

\[ b \leftarrow \text{IsMember}(S,c) \]

Where \( S \) is simple set, \( c \) is the component and \( b \) is a Boolean value
5.5.2 Operations on Composite Structure

Similarly, the following operations on the composite structure are defined.

Add an element

Component or set can be added in a composite set and is presented below

- Add a component

  \[ C' \leftarrow \text{add}(C, c_i) \]

Where \( C \) and \( C' \) are the composite sets before and after the addition of the component \( c_i \) respectively.

- Add a set

  \[ C' \leftarrow \text{add}(C, X) \]

Where \( C \) and \( C' \) are the composite sets before and after the addition of the simple or composite set \( X \) respectively.

Delete an element

Similarly, a component or set can be deleted from the composite structure and is presented below

- Delete a component

  \[ C' \leftarrow \text{delete}(C, c_i) \]

Where \( C \) and \( C' \) are the composite sets before and after the deletion of the component \( c_i \) respectively.
• Delete a set

\[ C' \leftarrow \text{delete}(C, X) \]

Where \( C \) and \( C' \) are the simple sets before and after the deletion of the simple or composite set \( X \) respectively.

Modify an element

A component or set can be modified from the composite structure and is presented below.

• Modify a component

It can be implemented using component add and delete operations

• Modify a set

It can be implemented using set-add and set-delete operations

Membership of an element

In the same way, the membership of a component or set of the composite structure can be checked and is presented below

• Membership of a component

\[ b \leftarrow \text{IsMember}(C, c_i) \]

Where \( C \) is composite set, \( c_i \) is the component and \( b \) is the Boolean value

• Membership of a set element

\[ b \leftarrow \text{IsMember}(C, X) \]

Where \( C \) is composite set, \( X \) is the simple or composite set and \( b \) is the Boolean value.
5.6 Aspect based Language Library Model

Based on the structure defined above, a new model, namely aspect based language library model is proposed. Language aspect components are used to form the language components by composition. This relationship takes care of the complexity involved in the multilingual software development due to language aspects. The language components are used to build the multilingual components. The multilingual components are used in the domain software development. This aspect based language library model is represented in Figure 5.7.

![Diagram](image)

**Figure 5.7: Aspect based Language Library Model**

In the Figure 5.7, the bottommost level is having $LA_{ij}$, language aspect component set, which is of composite set type $C$ (as per the structure defined in the previous section) comprising of the aspect components for the language $i$ and aspect type $j$. The $LA_{ij}s$ are composited into $L_i$, language library for the language $i$, which is a composite set containing the complete language components built, based on the language aspect components. These language sets are aggregated into the set $M$, which is a multilingual component set having the multilingual functionalities. This set $M$ is used with domain set $D$ to build the multilingual software.
5.7 Discussion

Aspect based language library model is expected to achieve the non-functional qualities which are the common demands of the stakeholders of multilingual software. Maintainability and reusability are the demanding non-functional qualities of multilingual software and the level of these qualities in the proposed model are discussed below based on (Goulao and Abren, 2004; Bertoa and Vallecillo, 2002; Gill and Grover, 2003; Boxall and Araban, 2004). In the discussion, the proposed model is compared with language library model which is a better model among the mined models.

**Maintainability**

Maintainability, in this context, is measured by the customizability of the multilingual components. If a multilingual component has more writeable operations to configure its internal aspects then, that multilingual component needs less maintenance. On the other hand, if a multilingual component has less writeable operations to configure its internal aspects and most of its internal aspects are hardcoded then the multilingual component needs more maintenance. In case of multilingual software, modification of language component is frequent due to the change requests from the stakeholders. Also most of the change requests demands change in the language aspects. The following discussion will explain how the maintainability is achieved in aspect based language library model when compared to language library model.

Consider $R$ to be a change request from the customer which affects the multilingual component $M$. Assume that $M$ is modeled as $M_L$ and $M_A$, the multilingual components modeled in language library model and aspect based language library model respectively. The component $M_L$ is designed with language components. These language components contain language aspects which are hardcoded. So this component has few writable operations and it is defined as $C_L$, customizability factor of the multilingual component $M_L$. But the component $M_A$ is designed with language components and these language components are, in turn, designed with language aspect components. So, both the language components and language aspect level
components have more writable operations and it is represented as $C_A$, customizability factor of the multilingual component $M_A$. So, $C_A$ has the maximum value and $C_L$ will be less than $C_A$. So, maintainability of language component $M_A$ is higher than language component $M_L$. Therefore, the maintainability quality in aspect based language library model is higher than the mined models.

Reusability

Reusability, in the multilingual context, of a component is measured by two factors, namely, 'Component Reusability' ($CR$) and 'Component Reusability Level' ($CRL$). A component is used in the design of multilingual software directly as a part of the software is called as 'Component Reusability' ($CR$). The second factor is the level of reusability that is achieved by the component in building components and they are in turn used for developing multilingual software applications. This is called as 'Component Reusability Level' ($CRL$). In the case of proposed model, separated aspects are made as components and they are directly reused in the development of language components. So, the level of $CR$ in aspect based language library model is higher, defined as $CR_A$, which is the Component Reusability factor of the proposed model. But in the language library model, language aspects are hardcoded. So, the level of $CR$ in language library model is lower, defined as $CR_L$, which is the 'Component Reusability factor' of the language library model.

Therefore $CR_A > CR_L$

The second factor 'Component Reusability Level' ($CRL$) of the aspect based language library model is defined as $CRL_A$ which is relatively higher than the $CRL$ of language library model, which is defined as $CRL_L$, due to dynamic composition of aspect components in language components.

Therefore $CRL_A > CRL_L$

This clearly shows that aspect based language library model has the higher reusability quality compared to other models.
5.8 Summary

Like any other domain, multilingual software also needs models which will be helpful in designing multilingual software. The mined multilingual software models have inadequacies and the inadequacies are analyzed using design space approach. A new model, named as aspect based language library model for multilingual software is proposed and formalized with implementation independent representation. Also the inadequacies are addressed by separating the aspects and design them as separate component sets. They can be used and/or reused in designing the multilingual software components. Expected multilingual software qualities can be achieved with the help of the proposed model. As a next step, a reference model for the multilingual software has to be developed and it is presented in the next chapter.