Capturing Coupling Information

In Domain Engineering

By

Rocky Zhenfeng Wu

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
December 2003

© Copyright 2003, Zhenfeng Wu

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89871-7

Our file Notre référence

ISBN: 0-612-89871-7

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Generative approaches to software development aim at maximizing the reuse of existing software artifacts while allowing the creation of customized high-quality software that meets the specific needs of its stakeholders. To do so, these approaches capture variability across a family of systems, called a domain, using feature models. Such models typically define a partial grammar of valid and invalid combinations of features, which ultimately leads to a generator. A large semantic gap can be observed between such highly abstract feature models and the components reused by a generator. In particular, we observe that coupling information is only marginally addressed at the level of features and, instead, must ultimately be extracted from the code of a generator. Yet, it is commonly accepted that highly-coupled systems greatly hinder reuse and lead to catastrophic maintenance costs, and thus that coupling must be addressed as early as possible during development. Our thesis is that coupling *can* be explicitly captured at the level of features, that is, in the problem space (i.e., the domain) rather than in the solution space (of reusable components). Restricting ourselves to features viewed as logical units of behavior, we propose in this work a textual documentation format to address two facets of feature coupling : *feature interactions* deal with combination rules between features treated as black-boxes, whereas *feature behavioral dependencies* detail which responsibilities of a feature are required by other features. Through small examples and a case study, we contrast the semantics captured in the proposed format with the information typically documented in a state-of-the-art generative approach.

Acknowledgements

The research process has been completed with the support of many individuals without whom this thesis would not have been possible.

My sincere gratitude goes to Dr. Jean-Pierre Corriveau for his valuable suggestions and his profound guidance during my research. I also want to thank him for his numerous comments on drafts of this work, and longstanding commitment to make this thesis a reality.

I would like to express my sincere thanks to Voja Radonjic for his valuable suggestions, numerous comments and technical discussions, and for his help with the editing. I would like to thank all the members in my research group for their great help.

Contents

1. Introduction

1.1 The Context	9
1.2 The Problem	11
1.3 The Proposal	17
1.4 The Validation Method	18
1.5 The Contributions	20
1.6 Road Map for the rest of the thesis	22

2. Background

2.1 The Two Spaces	23
2.2 About Coupling	25
2.3 About the approach of Czarnecki and Eisenecker	27
2.4 Capturing Complex Behavioral Specifications	32

3. Feature Contracts

3.1 The Feature Contract Approach	38
3.1.1 The Feature Coupling Problem Analysis	38
3.1.1.1 Feature Interaction and Behavior Dependency Model	39
Feature Interaction	41
Feature Behavioral Dependency	49
Features and Aspects	52
3.1.2 Feature Contracts	57
3.1.2.1 Descriptive Model	57
3.1.2.2 Operators	67
3.2 Applying Feature Contracts: Examples	72
3.2.1 Abstract Example	72
3.2.2 List Example	74
3.2.3 Product Distributor Example	82

4. A Case Study	
4.1 Game Plan	91
4.2 About Use Case Maps	92
4.3 Getting Started with the Elevator Control System	93
4.4 Scenario Analysis	94
4.5 Selecting Features	95
4.6 Scenario Refinement	107
4.7 Feature Interactions	111
4.8 Documenting Feature Coupling	113
4.9 A Comparison to a Start-of-the-Art Approach	126
5. Conclusions	
5.1 Recapitulation	137
5.2 Future work	138
6. References	140
7. Appendix	
A. Features Description For Elevator System	145
B. Features Versus High-level Responsibilities For Elevator system	
C. Grammar For Proposed Model	

List of figures

FIGURE 1-1:	FEATURE DIAGRAM OF A SIMPLE CAR	12
FIGURE 1-2:	FEATURE DIAGRAM OF A SIMPLE LIST CONTAINER	14
FIGURE 1-3:	GRAMMAR FOR THE LIST CONTAINER FAMILY	15
FIGURE 2-1:	PROBLEM AND SOLUTION SPACE	23
FIGURE 2-2:	GRAMMAR FOR A SIMPLE QUEUE	28
FIGURE 2-3:	GENVOCA LAYERS	29
FIGURE 2-4:	SUBJECT-VIEW EXAMPLE	33
FIGURE 2-5:	SUBJECT-VIEW CONTRACT	34
FIGURE 2-6:	REFINEMENT	36
FIGURE 2-7:	INCLUSION	37
FIGURE 3-1:	AN INTEGRATED VIEW OF FEATURE COUPLINGS	40
FIGURE 3-2:	A CLOSE LOOK TO FEATURE COUPLINGS	41
FIGURE 3-3:	A SUB FEATURE EXPRESSION VERSUS A MODEL WITH CONDITION	42
FIGURE 3-4:	AN EXAMPLE TO ILLUSTRATE THE ISSUE IN TREE-LIKE REPRESENTATION	44
FIGURE 3-5:	STATIC FEATURE MODEL VERSUS DYNAMIC FEATURE MODEL ON ALTERNATIVE FEATURES	45
FIGURE 3-6:	STATIC FEATURE MODEL VERSUS DYNAMIC FEATURE MODEL ON OR FEATURES	46
FIGURE 3-7:	STATIC FEATURE MODEL VERSUS DYNAMIC FEATURE MODEL ON MANDATORY AND OPTIONAL FEATURES	47
FIGURE 3-8:	AN EXAMPLE TO ILLUSTRATE THE RANDOM WALK-THROUGH AMONG FEATURES	49
FIGURE 3-9:	AN EXAMPLE FOR AN ASPECT	53
FIGURE 3-10:	A LIST EXAMPLE	55
FIGURE 3-11:	HIERARCHY OF CONTENTS IN MODEL	58
FIGURE 3-12:	THREE FEATURE NODE INTERACTIONS	72
FIGURE 3-13:	LIST FEATURE MODEL	75
FIGURE 3-14:	CUSTOMER ORDER PROCESS SCENARIO	83
FIGURE 3-15:	A SIMPLIFIED MANUFACTURING FEATURE MODEL	85
FIGURE 4-1:	GOMMA'S ELEVATOR SYSTEM	91
FIGURE 4-2:	STUBS IN UCMS	93
FIGURE 4-3:	AN INITIAL UCM	94
FIGURE 4-4:	DECENTRALIZED ARCHITECTURE	99
FIGURE 4-5:	PARTIALLY DECENTRALIZED ARCHITECTURE.	99
FIGURE 4-6A:	ASYNCHRONOUS BROADCAST	101
FIGURE 4-6B:	SYNCHRONOUS BROADCAST	101
FIGURE 4-7A:	A REQUEST THAT IS ALONG WITH THE CURRENT DIRECTION CAN BE ADDED INTO THE CURRENT CYCLE.	102
FIGURE 4-7B:	A REQUEST THAT IS NOT AHEAD IN THE CURRENT DIRECTION HAS TO BE ADDED INTO THE NEXT CYCLE. (I.E. NEXT DIRECTION)	102
FIGURE 4-8:	ELEVATOR SYSTEM BEHAVIOR DECOMPOSITION USING FEATURES	108
FIGURE 4-9:	A PLUG-IN FOR PROCESS FLOOR REQUEST	110
FIGURE 4-10:	A PLUG-IN FOR PROCESSING APPROACHING FLOOR MESSAGE	110
FIGURE 4-11:	A REFINED SCENARIO MAPPING BEHAVIORS WITH FEATURES.	111
FIGURE 4-12:	A FEATURE DIAGRAM FOR ELEVATOR SYSTEM	128
FIGURE 4-13:	A LAYER DEPENDENCY FLOW FOR ELEVATOR SYSTEM	133

List of Tables

TABLE 2-1:	THREE FEATURES FOR A SIMPLE QUEUE	28
TABLE 3-1:	A SUMMARY OF VARIATION AND PARAMETERIZATION	55
TABLE 3-2:	A SUMMARY OF DISTICTIONS BETWEEN FEATURES AND ASPECTS	56
TABLE 3-3:	A SAMPLE OF CUSTOMER ORDERS	83
TABLE 3-4:	A SAMPLE OF INVENTORY	84
TABLE 4-1:	THREE ARCHITECTURES FOR ELEVATOR SYSTEM	97
TABLE 4-2:	COMPONENT REALIZATION IN THREE ARCHITECTURES OF ELEVATOR SYSTEM	100
TABLE 4-3:	A SUMMARY OF FEATURES INTERACTIONS IN ELEVATOR SYSTEM	112
TABLE 4-4:	THE CONTRACT IN “ALLOCATION-DECISION-TIME” FEATURE	129
TABLE 4-5:	THE CONTRACT IN “ALLOCATION-SCORING-ALGORITHM” FEATURE	129
TABLE 4-6:	INITIAL GENVOCA MODEL FOR ELEVATOR SYSTEM	130
TABLE 4-7:	FINAL GENVOCA MODEL FOR ELEVATOR SYSTEM	135

1. Introduction

1.1 The Context: Generative Domain Engineering

A current trend in software engineering is reuse: reuse of analysis, reuse of design, reuse of code, and reuse of applications. Software reuse is driven by the desire to reduce software development costs and to leverage existing high quality artifacts. Meanwhile, stakeholders are demanding specialized software that fits their specific needs. Software must therefore be easily customizable. This creates conflicting forces on software development: reusing existing software artifacts while creating customized high-quality software that meets the specific needs of its stakeholders.

Recent developments in software engineering have attempted to address various aspects of this issue. In particular, domain-driven software development methods, such as software product line engineering ([Atk01], [CN01], [BFKLMSD99], [Bos00]), system-family engineering ([CE00], [Par76], [WL99]), and generative programming ([CE00]), all promote a *generative* approach to software production and reuse. More specifically, these approaches address variability in a *family* of systems¹. This family forms a domain, also called a *problem space* ([BV02]). Modeling software variability across a domain facilitates software reuse in a *solution space* [Ibid.]: a highly flexible generic architecture and a set of domain components can be reused in order to produce a new software product *within* the given family. Intuitively, a problem space is defined implicitly by a set of (system-level functional and non-functional) requirements. Conversely, a solution space is constrained by the set of reusable components available to be assembled together to form a solution, that is, a member of the family. *Domain engineering* addresses the modeling of the problem space, whereas *application engineering* focuses, in the solution space, on the selection of the reusable components that come to form a new family member.

¹ Distinguishing between software product lines and system families is of little relevance in the context of this dissertation.

Given that except for small, if not trivial systems, it will generally be impractical to enumerate all possible systems in a family, we need a *generative* approach to the capturing of variability, one that *implicitly* specifies family members as valid possible variations. In order to achieve this, the use of *feature models* ([FFB02], [BFKLMSD99], [Bos00], [Gey00], [KCHNP90], [CE00]) is generally advocated for domain engineering. Here, a feature is a "variation point" and a feature model purports to partially define a grammar of valid and invalid combinations of features, and ultimately (albeit indirectly) leads to a generator². This generator is then used in an application-engineering process to produce a particular executable system. More precisely, the generator is fed a particular configuration (consisting of a selection of features), which triggers the generation of its corresponding family member.

Before going any further, it must be emphasized that "modifiability" is different from variability. Modifiability is the ability for a system to be changeable. It constitutes a non-functional requirement, often referred to as "ease-of-modification", which can be measured (e.g., with respect to the number of hours required on average to implement a particular type of change in an existing system). Conversely, variability is the ability for a system to vary in its functionality across a domain. For example, a security component offers a variable encryption strategy if this component can use distinct encryption strategies. On the other hand, this component is highly modifiable if it is trivial for a designer to modify the set of encryption strategies this component handles. From this viewpoint, variability pertains to a domain, whereas modifiability is specific to a particular application (i.e., a particular member of a family of applications). Indeed, some family members may optimize modifiability at the expense of performance, whereas others (i.e., other variants in the family) may do exactly the opposite.

² We refer here to a "generator" as defined in ([CE00]) as "a program that takes a higher-level specification of a piece of software and produces its implementation". In application engineering, the higher-level specification takes the form of a selection of features. This so-called *configuration* of features leads the generator to try to assemble together the corresponding reusable components of the solution space into a

1.2 The Problem

Feature modeling is at the heart of generative approaches to software development. Yet authors differ on the exact nature of a feature and what purpose a feature model serves ([CE00], chapter 4). Regardless, it is commonly accepted that the generation of a member within a family (i.e., of a specific application within a domain) is rooted in the selection of a specific set of features. In general, these features are *not* independent of each other. That is, the selection of a particular feature may restrict the selection of others. For example, choosing the feature **Manual Transmission** for a car could rule out also selecting the **Cruise Control** feature or the **Pull trailer** feature. Czarnecki and Eisenecker ([CE00]) add:

Feature interdependencies are captured using composition rules [such as]:

1. Requires rules: Capture implications between features[.]
2. Mutually-exclusive-with-rules: Model constraints on feature combinations.
An example of such a rule is “[feature] *manual* is mutually-exclusive-with [feature] *automatic*... In general, mutually-exclusive-with rules allow us to exclude combinations of features where each feature may be seated in quite different locations in the feature hierarchy.

It is important to notice that the typical visual notations used for feature modeling can directly capture only some of these composition rules. Let us elaborate by considering Figure 1-1 (from [CE00]). In most notations, features are classified into the following categories:

- ? Mandatory – supported by all members of the family. (black dot)
- ? Optional – supported by some members of the family. (white dot)
- ? Alternative – mutually exclusive alternatives. (arc between alternatives)
- ? Or – any non-empty subset of the alternatives is allowed. (filled arc)

Figure 1-1: Feature diagram of a simple car (see pg 103 in [CE00])

Figure 1-1 shows a feature diagram of a simple car. Feature **Car** has three mandatory sub-features: **car body**, **transmission**, and **engine**, and one optional feature value: **Pulls trailer**. The **Transmission** feature has two alternative feature values: **Automatic** and **Manual**. The **Engine** has two OR feature values: **Electric** and **Gasoline**. The semantics of this model capture *some* combination rules (e.g., having an automatic transmission rules out having a manual one; having an electric engine and a gasoline engine is possible; the car requires a car body, a transmission, and an engine; the car *may* pull a trailer).

Typically, not all combination rules can be modeled into such a hierarchy. Instead, some composition rules take the form of textual annotations on the visual model (such as, for the sake of illustration, feature **Pulls trailer** requires feature **Transmission** to be **Automatic**).

In generative programming [Ibid.], given the ultimate goal is the construction of a generator, all feature combination rules are then either directly “translated” into code, or

first refined into an intermediate grammatical representation then further refined and implemented in the code of a generator. For example, Czarnecki and Eisenecker [Ibid.] suggest the design of a grammar before writing the generator itself. This design requires the following steps:

1. Identify the main responsibilities in the feature diagrams from the Domain Analysis.
2. Enumerate component categories and components per category.
3. Identify “uses” dependencies between component categories.
4. Sort the categories into a layered architecture.
5. Write down the [...] grammar.

As an example, given the feature model of Figure 1-2 ([Ibid.], p.572) for a simple list container, Czarnecki and Eisenecker produce the grammar given in Figure 1-3 ([Ibid.], p.577). It is essential to notice the semantic gap between these two figures.

On the one hand, the feature model proceeds from the domain, and visually captures *some* combination rules. For example, a list is monomorphic or (exclusively) polymorphic: a monomorphic list has its element type fixed to a single value, whereas a polymorphic list may hold elements of different types. We remark that the semantic relationship between features **ElementType**, **Monomorphic** and **Polymorphic** is *not* developed any further within the problem space (and no annotation complements the feature model here).

Figure 1-2: Feature diagram of a simple list container

```

List          : TracedList[OptCounterList] | OptCounterList
OptCounterList : LengthList[BasicList] | BasicList
BasicList     : PtrList[Config]
Config        :
    ElementType : [ElementType]
    Destroyer   : ElementDestroyer | EmptyDestroyer
    Copier      : PolymorphicCopier | MonomorphicCopier | EmptyCopier
    TypeChecker : DynamicTypeChecker | EmptyTypeChecker
    LengthType  : int | short | long | ...
    ReturnType  //the final list type

```

Figure 1-3: Grammar for the List Container Family

On the other hand, the proposed grammar is defined not in terms of features but, rather, in terms of reusable components (organized into categories such as a space deallocator called Destroyer, a type checker, an element copier, etc.). For example, the category TypeChecker includes both DynamicTypeChecker and EmptyTypeChecker, which are two reusable components of the generator. Neither the category, nor its actual possible reusable components, appears explicitly in the problem space. They exist at a much lower level of abstraction, one much closer to actual code of the generator. And the corresponding rule of the grammar of Figure 1-3 merely ensures that only one of these two reusable components is included in a specific member of the family of list containers. That is, this rule needs in fact to be further refined in the code of the generator (through the introduction of a boolean and of the specific names of the reusable components). More precisely, the generator uses an advanced C++ programming technique called

metaprogramming ([CE00]) to express the fact that the choice of type checker depends on a Boolean called `isMono` in the following code:

```
// simple IF: sets TypeChecker to be either Monomorphic or
// Polymorphic (which does nothing)
typedef
    IF<isMono,
        MonomorphicTypeChecker<ElementType_, tracing>,
        PolymorphicTypeChecker<ElementType_, tracing>
    >::RET TypeChecker_;
```

This code uses a meta-IF statement [Ibid.] to select (at compile time) which of the two reusable templated classes *MonomorphicTypeChecker* and *PolymorphicTypeChecker* will be put in the current family member being generated.

It must be emphasized that it is only in this code that one finally can see explicitly the nature of the semantic relationship that links features **ElementType**, **Monomorphic** and **Polymorphic**. Put simply, any list in the domain has a type checker. If a generated list is monomorphic, it does need a type checker (called *DynamicTypeChecker* in the grammar rule and implemented in class *MonomorphicTypeChecker*) that ensures any addition involves an element of the correct element type. And if a generated list is polymorphic, an *EmptyType* checker is reused (via class *PolymorphicTypeChecker*): one that does not perform any check when adding elements to the list.

The point to be grasped is that there is excessively little information in the feature model of Figure 1-2 to explicitly address the *coupling* between features **ElementType**, **Monomorphic** and **Polymorphic**. This coupling is “buried” in the solution space. This observation is the starting point of our work.

On coupling, van Vliet ([Vli02], p.301) writes:

Coupling is a measure of the strength of the intermodule connections. A high degree of coupling indicates a strong dependence between modules. A high degree of coupling between modules means that we can only fully comprehend this set of modules as a

whole and may result in ripple effects when a module has to be changed, because such a change is likely to incur changes in the dependent modules as well. Loosely-coupled modules, on the other hand, are relatively independent and are easier to comprehend and adapt. Loose coupling therefore is a desirable feature of a design (and its subsequent realization).

The problem at hand is to investigate whether or not, in the context of generative approaches to software development, coupling can be explicitly addressed in the problem space. It is commonly acknowledged that:

1. excessive coupling can drastically reduce the modifiability of a system, thus leading to the exponential maintenance costs reported throughout the software industry (e.g., [You97], [Vli02] p.411).
2. the earlier excessive coupling can be detected, the better, for the problem can then be solved before investing considerable resources into code and maintenance ([Bin00]).

Our thesis is that coupling *can* indeed be explicitly addressed at the level of features, that is, in the problem space rather than in the solution space.

1.3 The Proposal

For simplicity, we adopt Bosch's definition of a feature as a logical unit of behaviour that is specified by a set of functional and quality of service requirements ([GBS00]). In this dissertation, we will exclusively focus on functional requirements. Another dissertation of our research group tackles the relationship between features and quality of service requirements as outlined in ([Cor03]).

Feature creation (or recognition) and feature selection also lie beyond the scope of this work: we assume that a domain engineer will first come up with features (e.g., using some domain analysis method). Our contribution consists in proposing a textual format that will enable the explicit capturing of feature coupling information. How a domain engineer revisits the documented features, once coupling has been assessed, and how an

application engineer makes a specific selection of features and translates this selection (or equivalently, configuration) into inputs for a generator in order to generate a particular member of the domain are not part of this work.

We propose that feature coupling be decomposed into two facets, namely, *feature interactions* and *feature behavioral dependencies*. Much like the combination rules discussed by Czarnecki and Eisenecker ([CE00]), feature interactions view features as functional black boxes and document how features “affect” each other. Conversely, behavioral dependencies between features are to be specified by associating a list of functional “services” to each feature and documenting which services depend on others (of other features).

Our proposed format for documenting feature interactions and feature behavioral dependencies proceeds from the seminal work of Helm and Holland ([HHG90]) on the use of *contracts* for the specification of complex behaviors. We elaborate on this topic in the next chapter.

1.4 The Validation Method

The use of contracts for documenting semantic relationships between high-level abstractions is not new. For example, Jezequel ([JTM99]) used contracts to document the design patterns put forth by Gamma et al. ([GHJV95]). We believe, however, that no one has yet proposed to tackle explicitly coupling at the level of domain features, nor to use contract-like specifications to do so. This said, the question remains how can we measure the success of our proposal. Our validation method proceeds from the refinement of our initial claim. More precisely, the claim that “coupling can be explicitly addressed at the level of features” is refined into the following:

- ✎ Our proposed specification format for feature interactions captures at least the same information as the typical combination rules used in generative approaches.

- ✍ Feature behavioural dependencies are generally not addressed in current feature modeling approaches.
- ✍ Feature behavioural dependencies capture a semantically more detailed form of coupling than feature interactions.
- ✍ Feature behavioural dependencies provide a conceptual bridge between the problem space and the solution space in domain engineering.

We intend to validate these four claims through the following steps:

1. Review the state-of-the-art approach of Czarnecki and Eisenecker ([CE00]) for feature modeling, as well as the work of Helm and Holland ([HHG90]) on contracts in order to highlight the considerable semantic gap that exists in Czarnecki and Eisenecker’s work between the problem and the solution space.
2. Introduce the proposed format for capturing feature interactions and feature behavioral dependencies through small examples.
3. Contrast the resulting semantic information with what is captured in the state-of-the-art approach of Czarnecki and Eisenecker.
4. Develop at length how to capture feature coupling information for a semantically richer case study, and highlight which facets of this information would not be tackled by the reference state-of-the-art approach of [CE00].
5. Conclude with a discussion of additional benefits to be expected from the use of the proposed feature coupling format.

We add that we purposely kept our proposal to a textual format because we want to “settle” the semantic treatment of feature coupling *before* addressing the possibility of making such semantics part of a visual notation. Tool support is also excluded from this dissertation because it quite premature. Also, it must be emphasized that we are not trying here to improve the syntax or semantics of feature models: considerable work already addresses this problem (see [CE00], chapters 4 and 5). Quite on the contrary, we purposely limit our approach to the simple semantics used in the state-of-the-art approach

of Czarnecki and Eisenecker [Ibid.] in order to exclusively focus on the treatment of coupling.³

1.5 The Contributions

The Object Management Group ([OMG]) promotes the use of a model-driven approach to software development. Such an initiative reflects the current trend to deemphasize coding considerations in favour of a better understanding of the “mission” of a system with respect to the requirements of its stakeholders. While generative approaches to domain engineering do promote the use of feature models, we observe that current research in this field focuses a lot on implementation techniques (such as metaprogramming in C++ [CE00]). Feature models are most often seen merely as a starting point to get to the critical deliverable, namely the generator. And little effort is invested in discussing how one goes from feature models to a generator. Instead, detailed decisions (such as what are the reusable components and how do they interact) are etched in the code of the generator.

The goal of our research group’s work is to make quality a pervasive concern in domain-driven software engineering. We believe this standpoint has two advantages. First, it promotes the role and importance of quality engineering early in the process, a highly desirable characteristic for software development ([Bin00]). Second, we believe bringing quality into domain-driven development enables test generation across a family of systems and improves the reuse of tests across this family ([Cor03]).

While it does not address quality engineering, the present dissertation can be viewed as a stepping stone to achieve some of the goals our research group has targeted. We will elaborate on this idea in section 5.2. But already, we suggest that the contributions we now claim be understood within the context of the broader mandate of our research group:

³ [CE00] use the semantics of FODA ([KCHNP90]). In order to have a fair comparison with their approach with respect to feature coupling, we adopt the same semantics.

- ✍ Contribution 1: a textual documentation format for feature interactions that is semantically richer than the typical “feature combination rules” used in domain engineering.
- ✍ Contribution 2: a contract-like textual documentation format for feature behavior dependencies that promotes features as system-level logical clusters of *responsibilities* ([WK03]) and makes explicit these dependencies (which have generally been postponed to coding a generator).
- ✍ Contribution 3: an extensive case study that illustrates the proposed formats and emphasizes the information that they capture and that is not currently part of state-of-the-art generative approaches.

In turn, the viewpoint of contribution 2 allows us to envision the following future contributions:

- ✍ Bridging semantically between the problem space and the solution space via the notion of responsibilities [Ibid.]. More specifically, the reusable components of a generator will each be seen as a cluster of services that proceed from (and can be traced back to) the refinement of the clusters of responsibilities (i.e., features) identified in the problem space.
- ✍ Assessing coupling in the problem space and then, and only then, in the solution space. In other words, there is no point in developing a generator if, already in the problem space, we can observe an unacceptably-high level of coupling. Put another way, features *must* be loosely coupled in the problem space if we are to hope to derive a flexible architecture of a generator out of them.
- ✍ Developing a domain-engineering environment that will input the feature coupling information we propose and report to its user how much coupling is detected, thus allowing remedial action before developing a generator.

These future contributions will be briefly discussed in section 5.2.

1.6 Road Map for the Dissertation

Chapter 2 provides the necessary background information for the rest of the dissertation. In chapter 3, we will deal with steps 2 and 3 of the proposed validation method. The goal of that chapter is to provide the reader with sufficient motivation for the proposed textual format. Chapter 4 will then consist of the significant case study we have realized, and chapter 5 will conclude this dissertation with the last step of our validation method.

2. Background

2.1 The Two Spaces

Czarnecki and Eisenecker ([CE00]) suggest the following figure to summarize the passage from the problem space to the solution space.

Figure 2-1: Problem and Solution Space

Separating these two spaces allows domain engineers to focus on the conceptualizations inherent to a domain without immediate concern for the specific *platform* on which applications are to be generated. We remark that this is in perfect agreement with the recommendation from the Object Management Group of separating models at least between platform independent and platform specific ones ([OMG]). This guideline is at the basis of the Model Driven Architecture ([MA03]) and crucial to our work. Put simply, within the context of generative approaches to software development, we do want to understand variability first from a platform independent viewpoint, and then, and only then, from a platform specific one. That the latter view proceeds from the former is widely accepted. In fact, as suggested in the first chapter, it is the generator itself that bridges between the platform independent features and the platform specific reusable classes.

Our work proceeds from applying an “MDA-like” approach to feature coupling: given a domain, first we must obtain a platform independent model of feature coupling, and then and only then a platform specific one. In this dissertation, we restrict ourselves to

investigating the feasibility of such a platform independent model of feature coupling in a domain. Future work will investigate how to capture platform specific coupling information (i.e., between reusable components used by the generator) and how to go between the two levels of coupling information. Finally, it must be emphasized that, in our opinion, separating feature coupling from component coupling is essential in generative approaches because the (solution) space of components is highly variable. Thus, we believe assessing the coupling of a specific set of reusable components misses acknowledging other possible configurations of components could have been used and misses considering coupling inherent to the domain (that is independently of a specific configuration of reusable components). In other words, if we accept that the solution space is highly configurable (i.e. variable), then our contention is that it is necessary to investigate whether, regardless of any specific configuration of reusable components, there is coupling inherent to the domain. It is important to add that our position does not entail that the solution space is unconstrained, quite on the contrary: the domain limits which features belong to the problem space and, in turn, these features constrain the solution space by requiring that components of the latter space be derived from a configuration of features. Put another way, features address variability inherent to the domain, and such domain variability constrains a second level of variability, namely component variability⁴, captured in the solution space. Consequently, feature modeling is not a pure and free exercise of conceptualization; it must serve the purpose of leading to a configurable solution space.

The previous observations organize the rest of this chapter. First we briefly look at work on coupling and interactions outside of the context of domain engineering. In a nutshell, this work is only indirectly relevant because it is not concerned with variability (be it in the problem or the solution space). Second, we elaborate a bit more on the state-of-the-art approach we will use as a yardstick to compare with. We have selected the method of

⁴ Following Wirfs-Brock ([WK03]), we view components as clusters of responsibilities. It follows that modeling component variability can be viewed as trying to understand which responsibilities can be packaged together into cooperating reusable components. And, we repeat, it is crucial to grasp that feature variability indeed constrains component variability, because responsibilities are not independent of each other : they are 'owned' by features. We will use this observation by having our proposed format allow its user to associate structural and functional responsibilities with features.

Czarnecki and Eisenecker and overview it, with special attention given to the use of a GenVoca grammar. Finally, having highlighted the limitations of this approach, we motivate why the work of Helm and Holland ([HHG90]) on behavioral specifications (called contracts) appears to be an interesting starting point for the feature coupling format we propose in the next chapter.

2.2 About Coupling

Different software engineering textbooks offer similar classifications of forms of coupling. For example, van Vliet ([Vli02], p.301) identifies from tightest to loosest, content coupling, common coupling, external coupling, control coupling, stamp coupling and finally data coupling. He adds that these original forms of coupling from the 1970s have evolved in light of the development of newer programming languages. But the fact remains that the notion of coupling is still entrenched into code. Thus, most literature in software engineering about coupling (and cohesion) is conceptually quite far from our concerns. Even a discussion of coupling in object-oriented systems remains distant from the focus of this dissertation for it ignores the fundamental concern introduced in domain engineering, namely variability.

The question then is to ask whether or not research exists on feature coupling, that is, on feature interactions and behavioural dependencies. A quick search points to the work of Pamela Zave and Michael Jackson ([JZ98], [Zav93]). In her seminal paper on feature interactions and formal specifications in telecommunication systems ([Zav93]), Zave writes:

In this context a feature was either a service constituent [...] or a simple service itself, used as a synonym because the term 'service' had become overloaded [.]. Similarly, the term 'service interaction' referred to the interaction of [Intelligent Networks] and switch based services. New services introduced into the public telecommunications network showed unwanted and adverse interactions [...] where the use of one service was altered by the use of another; for example call forwarding and call waiting [.]. The

problem was variously seen as one of incomplete system specification of software use and maintenance.

For Zave, a service is user-relevant functionality (e.g., call forwarding and call waiting) and service interaction addresses the general problem of coupling between such services. In Zave's work and similar contributions (e.g., [KPR97]), the word 'feature' is used almost interchangeably with the word 'service'. In generative literature, Czarnecki and Eisenecker ([CE00], p.38) note that most researchers adopt one of two definitions:

1. An end-user-visible characteristic of a system (that is, the FODA ([KCHNP90]))
2. A distinguishable characteristic of a concept (e.g., system component, and so on) that is relevant to some stakeholder of the concept.

For Czarnecki and Eisenecker however, the FODA definition unduly puts too much emphasis on properties of a system, which directly affect end-users. They prefer the second definition, which is more general.

In our opinion, a philosophical debate on the exact semantic nature of features is a red-herring in the context of our work. Instead, the key observation to make is that in the work of Zave and similar authors, variability is not a consideration: features are indeed functional units, but they are not variation points. Zave and similar researchers want to analyze features of *a* system. Instead, we want to capture the features of a *domain* and document their interactions in a way that will allow an application engineer to *explore* feature combinations and their level of coupling. This is a key difference: their works are first and foremost semantic ones. For example, Klein et al. discuss how feature coupling can be addressed within the semantics of state machines (through reachability analysis, or checking for deadlock, absorbing states, or nondeterminism). Zave's work can also be characterized as work on feature coupling using formal semantics. Within that conceptual framework, the features of a specific system are a given, and the focus is on tools to understand their coupling. Put another way, from that perspective, the space of features is

closed and the space of semantic techniques to investigate (for their ability to capture coupling) is open. In essence we are dealing with the opposite situation: for simplicity and convenience we consider only one semantic representation (the one adopted by C&E, which we use as our yardstick), and we deal with a space of configurable features (i.e., the problem space). Put another way, variability not semantics per se is at the heart of our work. It follows that we now turn our attention to the treatment of variability and coupling in generative approaches and more specifically in the one we want to use as a comparison basis with our proposal.

2.3 About the approach of Czarnecki and Eisenecker

It is important to emphasize again the benefits of a separation of the problem space from the solution space. The point made earlier is that this carries through to the notion of variability: variability in the space of features altogether abstracts away variability in the space of reusable components. But a conceptual bridge must exist between the problem space and the solution one. Unfortunately, few authors directly address this issue. As explained in the previous chapter, we select the approach of Czarnecki and Eisenecker ([CE00]) because it is the only one which, though the use of a grammar tries to bridge between the two spaces at hand. Returning to the example of the list container, we elaborate a bit on the GenVoca grammar.

The use of GenVoca can be traced back to two independent projects: Genesis (see [Bat85-91]) and Avoca (see [Oma90a-b], [OAHP90]). Put simply, GenVoca provides semantics to specify how to organize different features into a layered dependency hierarchy. Consider for example, a simplistic queue with the following three variation points :

Feature Name	Description
Tracing	This optional feature provides the ability to have a trace function. If this feature is selected, the queue will be able to iterate over and display its elements (e.g., for debugging and other purposes)
Counter	This is an optional feature that allows the queue to record change on its number of elements. If a new element is added to the queue, the counter increases, if an element is removed, the counter decreases. (Such a counter is useful to implement the verification of pre- and post-conditions ([Mey94]) for the procedures of a list).
Storage Type	This feature refers to the underlying storage implementation used for the queue. For simplicity, the possible choices for this feature are array and linked list based storage types.

Table 2-1 : Three features for a simple queue

Czarnecki and Eisenecker suggest that through analysis, the dependencies between these features can be captured in the following very simple grammar:

Queue	:Tracing[OptCounterQueue] OptCounterQueue
OptCounterQueue	:Count[BasicQueue] BasicQueue
BasicQueue	:ArrayBased LinklistBased

Figure 2-2: Grammar for a simple Queue

It is important to note that this GenVoca grammar proceeds from the following analysis of the dependencies :

Figure 2-3: GenVoca layers

These layers are reflected in the grammar of Figure 2-2. Going top down through the grammar, a queue has Tracing or Counting. Tracing has Counting as a parameter. The Counting either involves Counting (with BasicQueue as parameter) or no counting. Finally BasicQueue is either array- or list-based. The interpretation is simpler bottom up, as suggested in Figure 2-3 :

- a queue has one of the two possible implementations. This choice is captured by BasicQueue.
- then this choice percolates up : the queue has the ability to count or not count. If it counts, this choice (in its realization) is dependent on the previous choice of implementation . If there is no count, the the only choice to percolate up is the choice of implementation (i.e., BasicQueue)
- finally, the queue has the ability to trace or not to trace. If it does, that choice depends on the choice of counting strategy (which itself depends on choice of implementation). If it does not, we still depend on choice of counting strategy.

It is important to note that a key idea of this mechanism is the abstraction of different alternatives for a feature into a single variation point. For example, the rule for OptCounterQueue refers to the variation point BasicQueue, not to the different possibilities of implementations at hand. This mechanism of layered abstractions

stemming from dependencies typically leads to the organization of the generator. In our example, the generator would have to determine the basic implementation. Then it would determine if there is counting or not. And if there is, it would have to generate the counting component specifically associated with the current choice of basic implementation. Having done so, it would then determine if tracing is required. Again, if it is, that functionality must be customized to handle the specific implementation at hand, taking into account the presence or absence of code for counting elements. That is, the actual iterator to use should be selected not only in light of the chosen implementation, but also with respect to the presence or absence of code for counting elements. It is essential to note that such considerations regarding which feature depends on which other feature would not appear in a corresponding feature model. We repeat, the GenVoca approach is the only we know to address feature dependencies so explicitly.

A fundamental disadvantage of this approach is, by the authors's own admission, its hierarchical nature: generally, it is not the case that dependencies can be organized so neatly into layers. Consider, for example, introducing the feature **ExceptionHandling** in our Queue example. The choices are with exception handling (of overflow and underflows) or without (in which case a run-time error will occur on overflow and underflow). Assume, for the sake of illustration that counting depends, not only on choice of implementation, but also on choice of exception handling strategy. This double dependency is not difficult to capture per se:

OptCounterQueue : Count[BasicQueue, ExceptionHandling] | BasicQueue

Here, we can think of Count has having two parameters (i.e., two dependencies) instead of a single one. Now assume some features, like OptCounterQueue depend on 2 features, some on one, some on none. Which features go on what layer? The point to be grasped is that dependencies between features do not necessarily organize themselves in clear-cut layers. What if a feature depends on features of different layers? Is this allowed (like in an open layered architecture) or must dependencies be with the immediate layer below (as in closed layered architectures)? The bottom line is that, as with feature models, adopting a purely hierarchical approach is problematic.

Graph-based approaches could be an alternative but their flexibility is offset by layout problems, and quick cluttering that drastically and typically reduces their understandability and usability. And thus, textual notations may be more appropriate to deal with complex dependencies.

Looking at the use of GenVoca, we remark that the short examples used by Czarnecki and Eisenecker ([CE00]) do not convince us that complex feature interactions can be handled in such a simple way. More precisely, in a GenVoca grammar, as illustrated above, a dependency is captured through a parameter. Roughly put, a feature has the feature it depends on as its parameters. Consequently a feature may only depend on a union of other features. But, in reality, we do want to address feature interactions such as:

*If choice for feature A is x AND choice for feature B is not y
THEN choice for feature C can be z1 or z2*

As mentioned in chapter 1, since GenVoca cannot handle such interactions. They may be documented separately from the feature models and the GenVoca grammar, but ultimately will be ‘buried’ in the code of the generator. Our proposed format aims at expressing such feature interactions as part of the coupling information of each feature.

Finally, we observe that the approach of Czarnecki and Eisenecker systematically views features as black-box functional units for whom possible values (in the form of other features) can be specified. Not only is there no distinction between decomposable and non-decomposable features, but, clearly, there is no attempt to address what we introduced in chapter 1 as behavioural dependencies. Recall that behavioral dependencies between features are to be specified by associating a list of functional “services” (or responsibilities) to each feature and documenting which services depend on others (of other features). This is a crucial aspect of our proposal for we suggest variability in the component space proceeds from the use of the notion of responsibilities: the task of the application engineer consists in configuring reusable components according to the constraints captured in the domain models. If components are seen as

clusters of responsibilities, and if features specifications do capture constraints amongst responsibilities, then, we repeat, we have an approach to feature modeling that bridges (i.e., minimizes but not necessarily eliminates the semantic gap) between the problem and the solution space. In comparison, from our standpoint, the black-box approach of Czarnecki and Eisenecker leaves a larger semantic gap between feature models and GenVoca grammars on the one hand, and reusable components on the other.

Having identified the flaws of the approach we will use as a yardstick, we now need to explain the conceptual roots of our proposed format for documenting feature coupling.

2.4 Capturing Complex Behavioural Specifications

The need to model complex interactions is not new. In the context of this dissertation, we must address both feature interactions and feature behavioral dependencies. The work of Helm and Holland ([HHG90], [Hol92]) on *contracts* appears particularly relevant for this purpose⁵. Whereas other authors have adopted strictly black-box approaches to feature/object dependencies, we believe the notion of contract as developed by Helm and Holland perfectly fits our need to view features as both black-box entities (with respect to feature interactions) and white-box ones (with respect to feature behavioral dependencies). Given this observation, we now summarize their work through an example. Consider the classical subject-view example used by Gamma et al. ([GHJV95]) in their illustration of their Observer pattern (see figure 2-4).

⁵ Contracts à la Helm and Holland should not be confused with contracts in Eiffel. Meyer ([Mey94]) uses the latter to capture the pre- and post-conditions of the member functions of a class, as well as the invariant of this class. In that case, contractual obligations are fairly abstract and participants need not be exhaustively identified. In other words, one writes such a contract to protect a class regardless of who it interacts with.

Figure 2-4: Subject-View Example

Helm and Holland suggest identifying the participants and obligations in such a context. Two kinds of obligations are suggested: i) type obligations, where the participant must support certain variables and external interface, and ii) causal obligations, where the participant must perform an ordered sequence of actions and make certain conditions true in response to messages received.

In our example the contract is straightforward: the views must always reflect any change in the subject. Helm and Holland suggest the following specification to detail this contract:

```

contract SubjectView
  Subject supports {
    value : Value
    SetValue(val:Value)  $\mapsto \Delta value \{value = val\}$ ; Notify()
    GetValue() : Value  $\mapsto$  return value
    Notify()  $\mapsto \langle \parallel v : v \in Views : v \mapsto Update() \rangle$ 

    AttachView(v:View)  $\mapsto \{v \in Views\}$ 
    DetachView(v:View)  $\mapsto \{v \notin Views\}$ 
  }
  Views : Set(View) where each View supports {
    Update()  $\mapsto$  Draw()
    Draw()  $\mapsto$  Subject  $\mapsto$  GetValue() {View reflects Subject.value}
    SetSubject(s:Subject)  $\mapsto \{Subject = s\}$ 
  }
  invariant
    Subject.SetValue(val)  $\mapsto (\forall v : v \in Views : v \text{ reflects } Subject.value)$ 
  instantiation
     $(\parallel v : v \in Views : \langle Subject \mapsto AttachView(v) \parallel v \mapsto SetSubject(Subject) \rangle)$ 
end contract

```

Figure 2-5: Subject-View Contract

In our opinion, this contract is remarkably precise:

- 1) the Subject supports both data and procedures (which make up its interface)
- 2) AttachView and DetachView allow a View to be exclusively included or excluded from the set of Views associated with the Subject at hand.
- 3) The Subject holds some data simply called value (an abstraction of the whole state of the Subject)
- 4) A Subject can export its value through procedure GetValue()
- 5) A Subject modifies its state through procedure SetValue whose parameter carries the new value. The delta sign indicates the update. Upon updating its value, the Subject must Notify its views. Indeed the Notify() procedure establishes that the message Update() will be sent to each element of the set of Views.
- 6) Views is a set of view, each supporting 3 procedures. Update refreshes (i.e., asks to redraw) the view. The Draw procedure gets the value of the subject, which defines the relationship called “reflects”: that the View reflects the value of the Subject. Finally, setSubject is used to associate a subject to a view.

- 7) The essence of the SubjectView contract is captured in its invariant: every view currently in Views must reflect the value of the Subject.
- 8) Finally, the authors also spell out how instances are initially hooked up: each view is attached to the subject, and in parallel, each view sets its subject to the single instance of Subject.

It must be emphasized that such a contract does capture not only the interface but the internal responsibilities of its participants. For scalability, the authors propose two additional mechanisms:

***Refinement** allows for the specialization of contractual obligations and invariants of other contracts. **Inclusion** allows contracts to be composed from simpler contracts.*

Figure 2-6 gives an example of refinement, and figure 2-7, of inclusion. Further explanations can be found in ([HHG90]). But the details of these contracts do not matter here: the key point to grasp is that this work does illustrate both type obligations (in Figure 2-5, Subject has a value and several procedures in its interface) and causal obligations (in Figure 2-7, Reshape consists in a specific sequence of three actions in the Parent-Child contract). In our opinion, both of these notions are relevant to feature coupling. While abstracting from specific reusable components, type obligations of features will capture their interface and generic state, and causal obligations will address how generic sequences of actions can be associated with some members of the interface of a feature. In other words, in our opinion the work of Helm and Holland provides both a conceptual and notational starting point for the format we propose (which has to take into account variability), especially with respect to feature behavioral dependencies. We introduce our proposal in the next chapter.

```

contract ButtonGroup
  refines
    SubjectView(Views = Buttons, Subject = State)
  State supports []
  Buttons : Setof(Button) where each Button supports [
    myvalue : Value
    chosen : Boolean

    Select()  $\mapsto$  State $\rightarrow$ SetValue(myvalue)
    Refresh()  $\mapsto$  Draw()
    Update()  $\mapsto$  if State $\rightarrow$ GetValue() = myvalue then
      Choose() else UnChoose()
    Choose()  $\mapsto$   $\Delta$ chosen {chosen = true}; Refresh()
    UnChoose()  $\mapsto$   $\Delta$ chosen {chosen = false}; Refresh()
  ]
  invariant
    Button.Select()  $\mapsto$  ( $\forall b : b \in Buttons : b.chosen \Leftrightarrow b.myvalue = State.value$ )  $\wedge$ 
      ( $\exists ! b : b \in Buttons : b.chosen$ )
  instantiation
    { $\forall b_1, b_2 : b_1, b_2 \in Buttons : b_1 \neq b_2 \Rightarrow b_1.myvalue \neq b_2.myvalue$ }  $\wedge$ 
      { $\exists ! b : b \in Buttons : b.myvalue = State.value$ }
    ([! b : b  $\in$  Buttons : State $\rightarrow$ AttachView(b) || b $\rightarrow$ SetSubject(State) )
endcontract

```

Figure 2-6: Refinement

```

contract ParentChild
  Parent supports [
    shape : Shape
    propagate : Boolean

    Reconfig()  $\mapsto$  {shape = {+c : c  $\in$  Children : c.shape}}
    Change(c:Child)  $\mapsto$  if propagate then Reconfig() else Rearrange()
    Reshape(s:Shape)  $\mapsto$   $\Delta$ shape {shape = s}; Rearrange(); Reconfig()
    Rearrange()  $\mapsto$  {|| c : c  $\in$  Children :  $\Delta$ loc; Place(c,loc) }
    Place(c:Child, loc:Location)  $\mapsto$  c  $\mapsto$  SetLocation(loc)
    ...]
  Children : Set(Child) where each Child supports [
    shape : Shape;
    location : Location;

    Reshape(s:Shape)  $\mapsto$   $\Delta$ shape {shape = s} Parent  $\mapsto$  Change(self)
    SetLocation(loc:Location)  $\mapsto$  {location = loc}
    ...]
  invariant
    Parent.Reshape()  $\vee$  Child.Reshape()  $\mapsto$  {Parent.shape = {+c : c  $\in$  Children : c.shape}}
end contract

contract AdjustView
  Viewer supports [
    Adjust(a:Adjustment)  $\mapsto$  Perspective  $\mapsto$  SetValue(fcn(Picture  $\mapsto$  getShape(),a))
  ]
  Adjuster supports [
    a : Adjustment
    Attach(v:Viewer)  $\mapsto$  {Viewer = v}
    Activate()  $\mapsto$   $\Delta$ a; Viewer  $\mapsto$  Adjust(a)
  ]
  Perspective supports []
  Picture supports [
    shape : Shape
    getShape() : Shape  $\mapsto$  return shape
  ]
  includes
    SubjectView(Views = {Viewer}, Subject = Perspective)
    ParentChild(Children = {Picture}, Parent = Viewer)
  instantiation
    Adjuster  $\mapsto$  Attach(Viewer)
end contract

```

Figure 2-7: Inclusion

3. Feature Contracts

Feature modeling plays a critical role in Generative Programming ([CE00]) (abbreviated as “GP” for the rest of the chapter). While useful as a high-level map of the problem domain, the weak feature model of GP is clearly not enough to enable sufficiently explicit control over the application generation process. To achieve that one must make visible, at the domain level, dependencies typically buried in the code of generators. The essence of our approach is to augment the weak feature model of GP with a model that expresses feature coupling. To do that we chose to build on the concept of contracts of Helm et al. ([HHG90])

This chapter introduces the notion of feature contracts and how they are used. Section 3.1, “The Feature Contract Approach”, presents 1) an analysis of the feature coupling problem, and 2) the concept of feature contracts as a way of addressing them. Section 3.2, “Applying Feature Contracts: Examples”, presents one abstract example and three concrete applications of feature contracts to Aircraft, List and Manufacturing problems.

3.1: The Feature Contract Approach

3.1.1: The Feature Coupling Problem Analysis

Feature modeling as used in GP ([CE00]) intentionally simplifies the problem of feature coupling (i.e. feature interaction and behavior dependencies). Within that weak model a limited form of interaction can be expressed: “optional and alternative variants attached to variation, may be additionally constrained by either mutual exclusion or mutual inclusion” ([Bos00]). The motive for simplification is ease of understanding ([CE00]), with the result that the hierarchical tree-like representation used for feature modeling is not able to represent various possible feature relationships ([KCHNP90], [Gri98], [CE00], [GBS00]).

We suggest to separating feature coupling into two spaces: variation and behavior. The former depicts the feature interaction while the later specifies the feature behavioral dependencies (both temporal and spatial). Let us briefly illustrate the orthogonal nature of these the two types of relationships with a simple example of a computer and a monitor. There are many models for a computer such as Pentium 2-128M-ATI 7500, Pentium 3-512M-G4 and so on. Also, a monitor can have many types such as Sony-Flat-CRT-17 inches, NEC-LCD-15 inches. In the behavior space, different models of a computer share the same function, *Compute*, while various types of monitors perform a *Display* behavior. *Display* behavior of monitor type needs to receive the result from *Compute* function, which is a temporal behavioral dependency. In variation space, some types of monitors may not be valid to some models of computer because the graphical card in a computer is incompatible with some types of monitors. However, the behavioral dependency from *Display* to *Compute* will not be changed regardless if the model of computer is incompatible with the monitor type.

3.1.1.1: Feature Interaction and Behavior Dependency Model

A picture shown in figure 3-1 outlines the core of our model. Feature properties are divided into two layers: feature variation space and feature behavior space. Horizontally features are interdependent through selection constraints between their respective variation spaces. Feature interactions can be bi-directional because of interdependencies among variants of individual features. Additionally, features are dependent on each other temporally and/or spatially as a result of a feature's behavior or structure using another feature's behavior or structure. In contrast to feature interaction, behavior dependencies are often uni-directional because 1) behaviors tend to occur sequentially, and 2) structures are hierarchically composed using other structures. In the vertical dimension, all variants of a feature specify their corresponding realizations, i.e. the behavioral outputs of the features that share the same goal of a given set of system responsibilities.

Figure 3-1: An integrated view of feature couplings

To better understand the above, we provide another figure 3-2 to illustrate feature coupling. In the following figure, each feature has a set of variants (dash box), and each variant specifies the realizations in behavior space (left hand side). In the variation space, different features interact with each other through selection constraints imposed by their variants, so the interactions are bi-directional. In behavior space, different realizations of a feature can be generalized into a behavioral output, so that different features are dependent on each other in a uni-directional way in terms of their behavioral outputs.

Figure 3-2: A close look to feature couplings.

Separating behavior and variation types of feature coupling allows model designers to use a divide-and-conquer strategy to addressing the problem of modeling feature dependencies. This separation is central to our approach.

Feature Interaction

The above discussion dealt with the coarse grain way of organizing a feature model. In this section we take the next step in refinement and begin with a critical review of feature modeling within GP as it pertains to detailed feature interactions. The immediately following subsection deals with feature behavior dependency.

Focus of feature modeling in ([CE00]) is software variability management. Feature relationships are represented with a set of elementary logical expression such as “Optional”, “Mandatory”, “OR” and “Alternative”. It is claimed that such a modeling technique has two major advantages. First, better understanding of variation due to its

conceptual decomposition, and second, addressing of the scalability problem (e.g. the combinations for $\{A\}$, $\{B\}$, $\{AB\}$, $\{BA\}$ can be simply represented in $A \mid B$). As alluded earlier, however, this model comes with several disadvantages.

Figure 3-3: A sub feature expression versus a model with condition

Ability to discriminate between a set and value is often useful, yet feature modeling within GP does not make that explicit. Consider a simple car example. In the left side of figure 3-3, the *Transmission* feature has two subfeatures, *Automatic* and *Manual*, however, the *Manual* subfeature has a 5-gears and a 6-gears mode variation, while there are no further variations for *Automatic*. It is obvious that the property of gear mode variation with *Manual* is completely different from the property of *Transmission* feature; hence, a better strategy to model this case is to treat the *Gear mode* as an individual variation available only if *Manual* transmission is chosen. The figure 3-3 shows the weakness of feature diagram in dealing with this case. The right side of the figure illustrates what exactly needs to be expressed in this case: that if *Manual* (a value) is selected for *Transmission*, then *Gear Mode* (a set) is available for further selection such as 5-gears or 6-gears; otherwise, no further options are available. Typically, our strategy

to handle a subfeature is that we separate the additional property *Gear mode* from its parent feature property (i.e. transmission type) by the means of placing a value (i.e. *Manual*) in the parent feature and creating an additional set with a triggering condition (e.g. if *Transmission* is *Manual*). With this model there is no uncertainty on whether a subfeature (*Manual*) is a value or a set.

A significant issue is the simplification, implicit in the tree-like hierarchy of feature diagrams, which forces feature selection to follow a top-down order. This is problematic when there are selection constraints across features that need to be exposed to the person selecting those features. For a small number of features, the issue may not be so obvious; however, for a feature model with many features, this could be particularly problematic because selecting another subfeature will follow another path along the feature model. Let us clarify with an example. As illustrated in figure 3-4, if the manager wants to change the feature B to C, he/she has to expect the outcome in which only the C1 and C2 can be further selected instead of B1 and B2; also, if the manager wants to change B1 to C1, the feature B has to be changed to C simultaneously because the path to B1 is different from the path to C1. Apparently, this hierarchical representation requires the manager to remember the whole hierarchy of feature model and the priorities for selecting features. Furthermore, feature model itself may not achieve one of the primary goals in domain engineering: fast feature turnover (see chapter 5 of [CE00]); meaning, a feature model should allow for indication of validity on changes in selected features based on conflicting feature interactions.

Figure 3-4: An example to illustrate the issue in tree-like representation

There is another issue in feature model representation as well as GenVoca. Neither is able to explicitly represent the feature interaction problem illustrated in the following simple example: variant a1 in feature A constrains feature B to the variant b1 because all other variants in feature B conflicts with a1. Feature model and GenVoca simply do not provide a logical representation for this case because “Alternative”, “OR”, “Mandatory” and “Optional” operators are not able to represent the “constraint” relationship in feature model, while the operators provided by GenVoca are only used to express the composition relationship. However, it is important to model this kind of feature interaction to relieve the person making feature selection decisions from having to know the details behind these selection constraints, and from having to remember to apply them manually. To address the aforementioned issue we construct a new model by adapting influence diagrams ([HM84]). To show how we adapt them we use a slightly more complex version of the car example.

In figure 3-5, left side illustrates that the *Transmission* feature has two alternatives: *Automatic* and *Manual*. Under *Automatic*, we may have *Automatic-3-gears* or *Automatic-4-gears* modes, and under *Manual*, *Manual-5-gears* or *Manual-6-gears* modes. On the right side of figure 3-5 we use a different way to express the same: each ellipse indicates a variation where all variants are inside the ellipse. A set of bidirectional arrows specifies

the relationship on how one variant in a feature constrains the selection of other features. For example, in the *Transmission Type*, if *Manual* is selected, the “null” (i.e. none of variants) in the feature *Automatic Type* must be selected, and the feature *Manual Type* must be “not null” (i.e. *Manual-5-gears* and *Manual-6-gears* are available for selection). On the other hand, if the *Automatic-3-gears* (i.e. “not null”) is selected, the feature *Manual Type* has to be disabled (i.e. “null”) and the feature *Transmission Type* must be *Automatic* (top right ellipse). The right and the left sides are equivalent in a logical sense; however, the left side forces feature selection in a top-down order, while right side leaves the order open.

Figure 3-5: Static feature model versus Dynamic feature model on **Alternative** features

In the figure 3-6, the right side illustrates how our model represents the “OR” relationship while the left side is the feature model that specifies the “OR” relationship. Again, both models are logically equivalent.

Figure 3-6: Static feature model versus Dynamic feature model on OR features

The left side of figure 3-7 illustrates “MANDATORY” and “OPTIONAL” features in terms of the feature model representation, while the right side transcribes those logical representations into conditional constraints. For example, if the *Car* concept is available in the domain (specified as *Enable*), the feature *Car body* must be available (i.e. *Enable* in *Car body* ellipse must be selected) and the feature *Pulls trailer* can be optionally available or not available (the condition “Not Enable $\not\leq$ Not Enable” indicates this). Conversely, if the *Car* concept is not available in the domain, neither feature *Car body* nor *Pulls trailer* is available in the domain. On the other hand, if the feature *Pulls trailer* is *Enable*, it implies that the concept *Car* is available in the domain (i.e. *Enable* must be selected for the *Car* feature), as a derived result, the feature *Car body* is also available in the domain. However, our model simply does not specify the case that *Disable* is selected for the feature *Pulls trailer*, this means there are not any constraints for selecting this variant, because the concept *Car* can be available in the domain although *Pulls trailer*

feature is not available and also there is no further impact on feature *Car body* for selecting the *Disable* in *Pulls trailer*.

Figure 3-7: Static feature model versus Dynamic feature model on **Mandatory** and **Optional** features

Let us summarize our analysis of feature interaction issues and our proposed model to handle them. As we can see through three examples above, our proposed model can be in compliance with various logical requirements in feature model. Additionally, our model is able to tell us whether a sub feature is set or value by the means of separating the additional property (e.g. *Electric model* in right side figure 3-6) from the sub feature (e.g. *Electric* in left side of figure 3-6) and creating a new set for that property. At the same time, our model can also avoid the variants combination issue (e.g. {A}, {B}, {AB}) because we directly place all possible variants into the corresponding variation, and fix the rule for selecting variants from a variation: only one variant is selected in a variation at a time. At this point, we agree exhaustively enumerating possible variants for a feature in our model could be tedious in some situations; however, this way we can support the special variants combinations such as {A}, {B} and {AB} but not {BA} (which is not possible with the feature model representation).

With respect to the feature selection issues in feature graphs and GenVoca grammars, our model provides an answer. In our model, a set of conditional constraints addresses how one variant in a feature constrains the selection on another feature (see right sides in figure 3-5, 3-6 and 3-7). Consider, for example a common printer constraint. We may use the statement “Regular feeder $\not\subseteq$ {Letter-sized, A4-Size}” to specify the constraint from feature *Feeder Type* to feature *Paper Size*, the statement means that if the regular feeder (i.e. not envelope feeder) is selected, the *Paper size* can be either *Letter-sized* or *A4-Size*. Additionally, our model builds bidirectional constraints between two features. For example, we may use the statement “{Envelope #10-size, Envelope C5-size, Envelope B4-size} $\not\subseteq$ Envelope feeder” to represent the constraint from the feature *Paper Size* to the feature *Feeder Type*. In this way, our model does not impose a top-down feature selection order because through triggering the corresponding constraint, the side effect of selection on a feature can be propagated to all relevant features. To show this usage, we use an abstract example for illustration. Given three features with a set of constraints in figure 3-8, we may choose to start our decision-making from feature A. We may first select A1 for feature A, and then through the constraint from feature A to B, B2 and B3 in feature B are available for further selection; also, through the constraint from feature A to C, all variants in feature C are available for further selection (Note, the statement “A2 $\not\subseteq$ {C1, C3}” does not constraint any variants in feature C while A1 is selected for feature A). Second, we may select feature C, and assuming C2 is selected for feature C, through the constraint from feature C to B, we know the B2 is the only option for feature B because the previous selection (A1) on feature A has removed the option B1 and the selection (C2) on feature C has constrained the possible choices in feature B into the subset {B1, B2}. With this example we can see that our model provides more flexibility in the selection process. We can select features through multiple paths: from feature A to B, then B to C, or from feature B to C, then to A. By establishing a set of constraints among features, our model explicitly represents these selection constraints as well as those relationships available in the feature model of GP (i.e. “Alternative”, “OR”, “Mandatory”, “Optional”).

Figure 3-8: An example to illustrate the random walk-through among features

Feature Behavioral Dependency

The behavior output of a feature suggests a system responsibility of that feature. By reviewing the history of design methodology, we perceive that for any of design methodology, it is necessary to clarify the responsibility for the atomic entity proposed by the design methodology. For example, in object-oriented paradigm, we may need to specify the responsibility for a class, while in database design; we may need to define the storage responsibility for each table. A feature-oriented design methodology followed in generative programming also needs to address responsibilities of a feature in terms of a model. However, both feature diagrams and GenVoca grammars are focused on capturing commonalities and variations in a domain and not on representing responsibilities associated with features. On the other hand, GenVoca, and other approaches extending its semantics, are used to represent the composition relationship among features. They do not provide enough semantics to illuminate how features are integrated to achieve overall system functionality. Furthermore, the behavior dependencies among features are omitted in those models because they lack behavior descriptions.

Associating responsibilities and features is fundamentally important in establishing a chain of responsibility across products generated within a given product line. Indeed, the chain here is what we call behavioral dependencies among features of two types: Spatial and Temporal. For example, in a list we might have ordering type feature (FIFO/LIFO/Priority), which is assumed to export a function: Insert that adds an element into a list. Also, another feature StructureForm offers a variable infrastructure for a list such as array-based and link-list-based. In this case, the StructureForm feature may have a structure called [Container] and a set of elementary functions such as Push_front, Push_back and Push_priority. Regardless of which variant is selected for either feature, there is a reference between two features. That is the Insert function must refer to one of Push_front, Push_back and Push_priority.

Behavior dependencies in a behavior-intensive system can be complicated. This in turn implies the need to provide descriptive notations to depict the behavior dependencies at the model level because it may be intractable to do at the implementation level. This is a common practice in object-oriented design ([JGJ97], [UCM], [Amy99]).

To summarize, there is a strong requirement to elicit the behavior aspect of features before implementing a generative model. Therefore, we realize there should be some kind of model existing within the proposed GP model (feature model & Genvoca). In order to fill the gap between design phrase and implementation phrase in GP, we suggest to link features with system responsibilities by modeling explicitly behavior dependencies between features. Additionally, our model not only covers the key feature concept but also includes, for the sake of reuse, aspect and aspectual feature concepts (aspects and features are discussed in the next subsection). We propose the following model that partitions feature behavioral description into three core parts: Input, Output and Composition:

Input: This is about to address how many parameters are created for the current aspect/aspectual feature. We use the term *singleton feature* for a feature occurring in one location, and *aspectual feature* for one that occurs multiple locations. The

singleton feature does not have a parameter. (Aspects and singleton features are discussed in the next subsection) This section is helpful to build up a contract between clients and the current aspect/aspectual feature because clients have to know at least how many parameters should be instantiated in order to use this aspect/aspectual feature. Therefore, many clients through a simple instantiation process can reuse this aspect/aspectual feature.

Output: This section is used to illustrate the behavior output of the current feature/aspect. This section exports a set of abstraction notations to address the system responsibilities (e.g. function/structure) of the current feature/aspect. Those notations in turn serve the later specification of behavior dependencies among features. Exporting notation to indicate the feature's responsibilities is fundamentally important for expressing behavior dependencies.

Composition: Based on the defined behavior outputs of each feature, this section will be mainly used to specify the behavior dependencies between current feature and other features / among different behavior outputs of current feature. Two types of behavior dependencies are being depicted in this section: temporal and spatial.

The above organization allows our model to express for a given feature 1) what form (aspect/singleton feature/aspectual feature, defined in the next subsection) a concept is, 2) what parameters of current aspect/aspectual feature are exported, 3) what the responsibility of the current feature is, and 4) dependencies between this feature's responsibilities and any other features' responsibilities. This level of detail may not be an issue for a small number of features, but for a large number, explicitly answering those questions is useful for later reusing the corresponding domain model (for example, an explicit model for STL lib can help application programmers to use or extend such a generic library). More beneficially, such a model may also help domain designers to integrate all found features with corresponding system responsibilities so that complex behavior dependencies can be elicited before implementation. Indeed, our model provides

a rough draft that allows domain designers to decompose overall system responsibilities into different features and also specify the cooperation among the behaviors.

Features and Aspects

Aspects play an important role in feature modeling and reuse. Czarnecki defines them in chapter 8 of ([CE00]) as follows: “*an aspect is a localized function unit which crosscuts a system concern (e.g. synchronization, component interaction, persistency, security control and so on)*”. Aspect-oriented programming (AOP) recognizes that aspects can be seen as an aggregation of concerns, parts of software that are relevant to a particular concept, goal or purpose ([OT00]). A few basic terms need to be understood in connection with aspects and their clients. An aspect is typically conceived to have minimal coupling to its clients. Binding time of an aspect can be run-time or compile-time, and its binding site (how it is woven into a client) is either static (e.g. inline) or dynamic (e.g. virtual function). In terms of reusability, “noninvasive adaptability” means that aspects should not require further modification on clients while they are “injected” into them. More details regarding these definitions can be found in chapter 8 of ([CE00]).

An aspect only interacts with clients through a parameterization mechanism but with minimal invasive involvement with them; the aim often being domain independency. All information about clients required by an aspect is obtained via a set of exported parameters. In addition, an aspect has no variation in general, since it does not provide any different realizations for any functional units, but rather generalizes a set of procedural routines to adapt different importing parameters. Examples of such a property of aspects can be found in the well-known STL library vector component, which does not provide any variation on implementation within a vector itself. The element type involved with internal operations in the vector is parameterized and provided by users (figure 3-9). Different data types can be imported into the aspect (vector) by instantiating the *Data Type* parameter of the aspect so that the aspect (vector) can obtain the type information about the imported data.

Figure 3-9: An example for an aspect

Consider the definition of feature in chapter 4 of ([CE00]) “a variation point is a feature that has at least one direct variable sub feature (or feature)” and also Jacobson’s definition of variation point: “a variation point identifies one or more locations at which the variation will occur” ([JGJ97]). We realize that 1) features vary their implementations at a specific decision point, and 2) a variation is different from a parameter. A variation is used to provide different options at some point, while a parameter is an abstraction mechanism to avoid the “ad hoc” behavioral coupling typically found in non-parameterized designs (for example `VectorInt`, `VectorDouble` instead of `Vector<DataType>` instantiated to `Vector<int>`, `Vector<double>`). More precisely, a variation provides sets of variable implementations on functions or structures, whereas a parameter offers a generic entry for later instantiation but with one implementation. Distinguishing these concepts can help us to understand the relation between features, concerned with variability, and aspects, concerned with genericity.

Features can occur in one location or multiple locations in a domain. In the case that a feature occurs in only one location in a domain, behavioral dependencies among features (particularly for temporal dependencies) are quite often specified directly using feature names. For example, while behavior output `Func1` of Feature A refers to another behavior

output Func2 from feature B, then this may be represented as A.Func1 refers to B.Func2. However, if a feature appears in many locations, for the sake of reusability, such a feature should not be explicitly referenced by its actual name. Instead, the dependencies between that feature and its clients should be parameterized like an aspect. For example, behavior output Func1 of feature A (to be used by multiple clients) needs to reference the behavior output Func2 of multiple features: A.Func1 refers to <<Some features>>.Func2. In such sense, the supplier of Func2 is abstracted through parameterization mechanism so that feature A can be used in multiple contexts.

The distinction between one location and multiple locations of feature is important. It is not necessary to parameterize cooperators for a feature that only occurs in one location, because such a feature may not be useful for other locations or other domains. An example can be found in our case study shown in chapter 4, where the feature Allocation Scoring Algorithm is responsible to provide a scoring function for elevator system to evaluate which elevator will be the best one to serve a floor request. However, such a feature may not be useful in other domains, so we do not need to parameterize the behavior dependencies between this feature and other features. On the other hand, feature Communication (parameterized on the sender and receiver) can be used in many locations in an elevator system, or other domains that need communication between objects. Another important reason to identify the need to parameterize behavioral coupling is that parameterization typically requires additional operations such as type checking which un-parameterized behavioral coupling does not.

Let us further illustrate the concept of aspectual feature with the list example from the STL library. A list has a variation for constructing a list in different element orderings (e.g. push front versus push back, and pop front versus pop back) while it also exports Data Type as parameter. The following figure3-10 illustrates the difference between variation and parameter in a linked list. In figure 3-10, different element types such as integer, floating point and so forth can instantiate the Data Type parameter of the generic list component. On the other hand, the generic component also allows us to switch our

decision on ordering type (this is variation), for each possible variant such as FIFO, LIFO and so on, while providing a corresponding implementation.

Figure 3-10: A list example

The following table 3-1 summarizing the above distinction between variation and parameters.

Type	Purpose	Number of implementations
Parameterization	Minimize the behavioral dependencies by abstracting interactions with clients into parameters.	Only offer one implementation.
Variation	Provide a point to control variable realizations on functions or structure representations	Multiple implementations

Table 3-1: A summary of variation and parameterization

In the table 3-2, we clarify definitions for various forms of features and aspects. Aspects generally have parameters to decouple their dependencies on clients, but do not have

multiple implementations (i.e. variation). They can be used in many domains because they parameterize behavioral coupling; as a consequence, aspects have a high adaptability to different usage contexts. On the other hand, singleton features do not have any parameters because they have been designed to be context-dependent (i.e. only used in one place). Finally, aspectual features inherit characteristics of both features and aspects. They can have variation while also decoupling behavioral dependencies through use of parameterization mechanism. They can be adapted to different contexts as well as providing options to clients.

Type	Variation	Parameterization	Domain- Independent	Adaptability
Aspect	No	Yes	Yes	High
Singleton feature	Yes	No	No	Low
Aspectual feature	Yes	Yes	Yes	High

Table 3-2: A summary of distinctions between features and aspects

Determining if a concept should be modeled as an aspect, a singleton feature or an aspectual feature is a design strategy. Further discussion on such strategies is beyond the scope of this thesis, but it is clear that various forms of representation for a concept often coexist in product line family. As a result, a feature model is required to provide semantics to satisfy these needs. We, therefore, formulate the representation of variation and parameter so that our proposed model is able to support the expression of different forms of features and aspects.

3.1.2: Feature Contracts

The reader may find it helpful to refer to any one of the examples in Section 3.2 before completing all of this section to get a sense of how feature contracts are used.

3.1.2.1: Descriptive Model

In this section, we propose a descriptive model to represent features and aspects. In our model, we formalize the expression of feature variation, and their behavioral output. Moreover, the description for bi-directional feature interactions and uni-directional behavioral dependencies among features are integrated into the model. In order to obtain a clear representation for features relationship, we intentionally encapsulate all interactions and dependencies of all relevant features and current feature (i.e. the feature being described) into the statement of each feature. In other words, instead of centralizing all feature interactions and behavioral dependencies occurring among features into one place, we distribute those contents into each single feature so that each feature statement is able to demonstrate its related feature relationship (including feature interaction and behavioral dependencies). For example, we may possibly centralize the feature interactions in one statement section. However, such a representation may not be able to immediately point out which feature interaction belongs to which feature. The figure 3-11 shows the skeleton of proposed feature model. There are two main sections, which establish the core part of this model: Interaction and Behavior. Interaction section is about to demonstrate the “constraint” relationship among features, it includes two parts: “*Variation*” to describe how many variants are available for this feature, and “*Contract*” to specify the interactions between all relevant features and current feature. On the other hand, *Behavior* section, is namely to expose what behavior a feature should output and the behavioral dependencies between current feature and relevant features. Additionally, the *Behavior* section may possibly contain the “Input” subsection if the current feature is *aspect* or *aspectual feature* (see Features and Aspects sub section of 3.1.1.1).

Figure 3-11: Hierarchy of contents in model

Name

-- *Description:* Name of feature or aspect. It is noticed that feature name should be distinguished from feature variation for the reason that a feature also has behavioral properties (i.e. the behavioral output of the feature). For example, the statement “ $\times$ FeatureA.<V>” represents that the variation V in feature A is being referred. On the other

hand, the statement “~~✗~~ FeatureA.Func()” means that the behavioral output *Func()* is being referred. Through the example just mention, we can recognize that feature name and feature variation are not equivalent because a feature may also contain other properties such as a function/structure output, and the variation of a feature only indicates one of feature properties: that is a variable control point.

Interaction

-- *Description:* Selecting a particular option of a feature might influence and further constraint the options space of other features. It is, therefore, essential to provide a clear description of such a relationship among features. For that purpose, this subsection first introduces what *Variation* is exported by this feature and how many variants are included in the *Variation*. Later, how the decisions on other relevant features constrain the options of current feature are also exposed in this subsection.

Variation

-- *Description:* This section is used to demonstrate how many possible variants belong to the variation exported by current feature. However, not all kinds of features have a variation, only *Singleton* and *Aspectual* feature (see Features and Aspects sub section of 3.1.1.1) have a variation. An aspect typically does not have a variation because it only provides one implementation (see vector example in Features and Aspects sub section of 3.1.1.1). Therefore, this section could be NULL for an aspect.

-- *Formal expression:*

$\langle V \rangle ? \{v1, v2, v3, \dots\}$

This means that current feature has a variation V that contains a set of variants such as v1, v2, v3, ...

-- *Example:*

Given a feature called *StructureForm* for a list which allows the infrastructure of a list varies on array-based or link-list-based, we may describe it as follow:

Variation: <type> ? {array-based, linklist-based}

It means the feature *StructureForm* has a variation *type*, which contains two variants: *array-based* and *linklist-based*.

Contract

-- *Description:* By using a set of logical representations, this section specifies how the selection on other features constrains current feature. In other words, this establishes uni-directional constraint from other to current feature in the way that, if a certain variant of other feature is selected, how the current feature responds to previous selection effect. Apparently, the possible variants set of current feature may be further constrained into a subset, it means some options of current feature may be ruled out as a consequence of previous selection.

As we mentioned, the feature interaction is mutual between two features in section 3.1.1. One may question where another direction is stated in. Straightforwardly, the answer is the statement of the feature that is interacting with current feature. For example, if the current feature A interacts with feature B, the statement of feature A is to specify how the selection on feature B affects the feature A. In turn, in the statement of feature B, we may define another direction, that is how the selection on feature A affects the feature B.

Different operation such as “?” , “||” , “&” , “?” , “” and “ ” will be used to symbolize feature interactions. Additionally, in order to clarify this part, we order all relative contents into two sections: the first section gives a global view of the relationships between other features and current feature, by a glance, this can give rough information about how other features interact with current one; The second

section, called *Mapping*, provides more precise details on how selection on other features affect the selection on current feature.

--Formal expression:

In global view:

Feature A $\bowtie$ Feature B

It means that feature A interacts with feature B.

In detail view:

Mapping:

Feature A.<V> ? {a1, a2} $\bowtie$ {b1, b2}

This means if variant a1 or a2 is selected for variation V in feature A, then current variation has to be variant b1 or b2.

--Example:

Selected from a list domain (see list example in section 3.2.2), feature *Length* provides an option on whether the length of a list is fixed or not. Simply, there is a feature interaction between the *Length* and the *StructureForm*: if the length of a list is fixed, the infrastructure of the list cannot be *link-list-based* because the length of a link list is variable. The following statement demonstrates how the selection on feature *Length* affects the current feature *StructureForm*:

? Contract: Length $\bowtie$ StructureForm

Mapping

Length.<Variability> ? {fixed} $\bowtie$ {array-based}

The statement above indicates that feature *Length* interacts with the current feature (*StructureForm*) so that if the variant *fixed* in feature *Length* is selected, the current feature has to be *array-based*.

Behavior

-- *Description*: This section specifies three parts: what kind of parameters current feature exports for further instantiation (see aspect discussion in Features and Aspects sub section of 3.1.1.1); what behavioral output current feature implements; how the behaviors of current feature depend on other features.

Input

--*Description*: a set of parameters to be exported for further instantiation so that the behavioral dependencies between external clients and current feature/aspect can be decoupled (see Features and Aspects sub section of 3.1.1.1).

However, this can be NULL since it is not necessary to parameterize all references if current feature is not designed to be domain independent or for the case of multiple occurrences (see Features and Aspects sub section of 3.1.1.1). In other words, this section is used to demonstrate what parameters current aspect or aspectual feature should have. For the singleton feature, this can be null because its behavior dependency may not decoupled in the targeted domain.

--*Formal expression*:

Input: <<Parm>>

This means a feature exports a parameter, which is called “Parm”.

--*Example*:

In the list domain, a feature *MemoryAllocation* provides different strategies to allocate memory for the data (a copy or a reference to inserted element) to be stored in a list. However, this feature can also further support different data structures to store data such as binary tree, stack and so forth. The *MemoryAllocation*, thus, should be transformed into a more generic form for the sake of reuse. Obviously, the main concern in such a transformation is to decouple the behavior dependency between external clients (who need to allocate memory) and *MemoryAllocation*. In the request of memory allocation, various clients may want to allocate different size of memory in order to store different kinds of data. Therefore, the major task to build up a reusable *MemoryAllocation* feature is to abstract the data type to be allocated memory. For this purpose, we may parameterize the data type for the *MemoryAllocation* feature so that the behavior dependency (that is the data type itself) between memory requester (external clients) and memory provider (*MemoryAllocation* feature) can be decoupled. The following statement specifies this parameter in the *MemoryAllocation* feature:

Input: <<Data Type>>

The whole statement for *MemoryAllocation* can be found in list example in section 3.2.2.

Output

--*Description:* by using a set of abstract symbols, behavioral commonalities of current feature such as a function or structure are specified in this section. Notably, this section can be NULL if current feature only generates some high-level constraints on other features; it is for the case that some feature model may be designed in form of hierarchy in order to facilitate decision-making process (e.g. decision tree). In this case, there is nothing to be produced but the set of conditions that further constrain possible options of other features. For example, in a war game, commander will not directly get involved with the soldiers but will generate a series of commands to constrain

their actions; in this case, soldiers perform various tasks issued to them as commands.

It is necessary to discriminate if a function or structure of feature is externally visible to other features. We use bold font to represent externally visible output.

--*Formal expression:*

Output: **Function()**

This indicates current feature has a function as output that is externally-visible.

Output: **[structure]**

This means current feature has structure output that is externally-visible.

--*Example:*

Selected from a list domain again, the *StructureForm* feature must have a container to store elements in a list, regardless of its infrastructure (e.g. array-based/link-list-based). Meanwhile, different infrastructures should export a set of elementary functions so that external clients can manipulate a list. Those functions can be *Push_back*, *Push_front*, *Pop_up* and *Pop_front*. Based on our design, those functions are internally-visible. The statement below equivalently specifies the above:

? Output:

Function & Structure:

✍ *Push_back()*
✍ *Push_front()*
✍ *Push_priority*
✍ *Pop_back()*
✍ **[Container]**

Composition

--Description:

This section specifies the inter-dependencies among feature behavioral outputs. This content is categorized into two types: temporal and spatial. For the spatial behavior dependencies, this section specifies two types of relationship: “*A structure consists of an internal structure*” and “*a structure exports a function*”. For the temporal behavior dependencies, this section defines the temporal sequence of behaviors occurrences. Two types of temporal dependency relationships are specified: “*A function refers to a function*” and “*A function refers to a structure*”. Indeed, the former is to address who invokes whom with respect to the interactions among functions, while the later is to explicitly represent that a function needs to access some data.

--Formal expression:

-- Spatial type dependency:

Feature A.**[structure]**![internal structure]

This specifies that the externally-visible **[structure]** consists of an internally-visible [internal structure] where the **[structure]** is the behavior output of feature A, and the [internal structure] is the behavior output of current feature.

Feature A.**[structure]**!function()

This specifies that the externally-visible **[structure]** exports an internally-visible *function()* where the **[structure]** is the behavior output of feature A, and the *function()* is the behavior output of current feature.

-- Temporal type dependency:

Function1(){

```

    ↗ Feature A.Function2();
}

```

This specifies that the externally-visible **Function1()** needs to refer the *Function2()* where the **Function1()** is the behavior output of current feature, and the *Fuction2()* is the behavior output of feature A.

```

Function1(){
    ↗ Feature A.[Structure];
}

```

This specifies that the externally-visible **Function1()** needs to access the externally-visible [**Structure**] where the **Function1()** is the behavior output of current feature, and the [**Structure**] is the behavior output of feature A.

--Example:

Reusing the list example to illustrate this section, we choose behavior dependencies among feature *StructureForm* and *ListType*. As demonstrated before, the *StructureForm* defines different infrastructures for a list; it outputs an externally-visible structure [**Container**] and four internally-visible functions: *Push_back*, *Push_front*, *Pop_back*, and *Pop_front*. On the other hand, the feature *ListType* offers two options on how to order the elements in a list. Those options are FIFO and LIFO ordering. In terms of the behavior output, the feature *ListType* outputs two externally-visible functions: **Insert** for adding an element into a list, and **Delete** for removing an element from a list. The following statement specifies the behavioral dependencies for feature *ListType*:

? Composition

```
✎ StructureForm. [Container]!Insert(){  
    ✎ StructureForm.{Push_back()? Push_front};  
}  
  
✎ StructureForm. [Container]!Delete(){  
    ✎ StructureForm.{Pop_back()? Pop_front()};  
}
```

This means that the current feature (*ListType*) outputs two externally-visible functions: *Insert* and *Delete*. Both functions should be composed by the behavior output **[Container]** of feature *StructureForm*. Additionally, the **Insert** requires a reference to either the *Push_back* or *Push_front* function (both are the output of *StructureForm* feature), and the **Delete** refers to either *Pop_back* or *Pop_front* function. The symbol “?” corresponds to the “Alternative OR” relationship. For example, the statement “Push_back()? Push_front()” indicates that only one of *Push_back* and *Push_front* will be referred by the **Insert**. In other words, the **Insert** will select one of those functions based on the selected variant (e.g. FIFO or LIFO).

Responsibilities --- a textual description that specifies the responsibility of each behavioral output such as function or structure.

3.1.2.2: Operators

“◇” --- This symbol scopes a variation name in *Variation* section. For example: “<strategy>” means the current feature exports a variation called “strategy”.

“<<>>” --- This symbol scopes an input parameter in *Input* and *Composition* section. For example: <<tasking type>> is a parameter for further instantiation.

“&” --- “AND” logical operator which can be applied in both *Contract* and *Composition* sections. For instance, in *Contract* section, the statement “B $\bowtie$ A & C $\bowtie$ A” means that both interactions between A and B and those between C and A exist concurrently. Consequently, both interactions have to be represented. Also, this operator can be applied in *Composition* section, for example, $\bowtie$ F.{f1() & f2()} means both f1() and f2() need to be referenced by the current feature. The $\bowtie$ operator here indicates the reference.

“||” --- “OR” logical operator used in *Composition* section. For example, giving a variation called *Plan* for feature *LifeStrategy*, we may have three possible options: full-time study, full-time work and part-time study (i.e. work and study at the same time). In terms of the behavior outputs for this feature, we may design one externally-visible function **Behave** and two internally-visible functions: *Work* and *Study*. In this case, the following statement says that the **Behave** function may refer to one of the combinations: *Work*, *Study*, *Work* and *Study*.

```
Behave(){
     $\bowtie$  Work() || Study();
}
```

“?” --- “Exclusive OR” logical operator with similar usage as “&” in *Contract* and *Composition* section.

“~” --- “Negation” logical operator, which is only applied in the *Mapping* subsection. For example, the statement “F1.<V> ? {v1} $\bowtie$ ~{b2}” means if variant v1 is selected for variation V of feature F1, then the current feature can not be selected as b2.

“ $\bowtie$ $\bowtie$ ” --- This symbol represents interaction occurring between two features. It is only used in the *Contract* section to specify “*which feature interacts with which feature*”. For

example, the statement “Feature A $\bowtie$ Feature B” means feature A interacts with feature B by a set of selection constraints where the details for those selection constraints will be shown in the *Mapping* subsection.

“ $\Rightarrow$ ” --- This symbol represents “implies” obligation between two predicates in the *Mapping* subsection (see *Contract* in subsection 3.1.2.1).

“[]” --- This symbol scopes the structure output of a feature or aspect in *Output* section. For example, a statement “[Container]” in the *Output* section implies that the current feature exports a structure abstraction called **Container**.

“()” --- This symbol represents a function output in *Composition* and *Output* section. For example, the statement “Func()” in the *Output* section means current feature has a function output called “Func”.

“{}” --- if this symbol is used in the *Variation*, it scopes a set or subset of variants (see *Contract* in subsection 3.1.2.1). However, if this symbol appears in the *Composition* section, it scopes the temporal dependencies for the function output of current feature (see *Composition* in subsection 3.1.2.1).

“<=>” --- This symbol represents an instantiation relationship in section *Composition*. The left side is the instantiator (the value to be set) and the right side is the instantiatee (e.g. the parameter for instantiation). For example, [component] <=>B.<<structure type>>, means the parameter “structure type” of feature B will be instantiated as the value [Component]. Normally, this is used in reference scope of function output.

“?” --- This symbol represents a “is one of” relationship. It is used in the *Variation* and *Contract* section (see *Variation* and *Contract* in subsection 3.1.2.1). For example: in Variation section, the statement “<V> ? {v1, v2}” means variation V can be one of variants v1 and v2 but not both; in Contract section, the statement “Feature A.<V> ? {a1,

$a_2 \in \{b_1\}$ ” means if one of variants a_1 and a_2 is selected for variation V in feature A , then b_1 is the unique option for the current feature.

“ $\in$ ” --- This symbol signifies that the following behavioral output will be referred, or instantiated by this feature behavioral output. This symbol is used in the section **Composition** to represent that a function needs to refer other features outputs (see **Composition** in subsection 3.1.2.1).

“@” --- This symbol represents an instance of the output structure in the **Output** section. It is necessary to have this notation to distinguish one instance from all instances of an output structure. This is similar to the distinction between a class and an object of the class in object-oriented paradigm. For the precise example of the usage of this symbol, refer the elevator system case study in Chapter 4.

“?” --- This symbol represents all instances of a structure output in the **Output** section.

“...” --- This symbol is to indicate the permutation of all variants that can not be individually listed. Generally, this symbol is used in **Variation** section.

“NULL” --- This symbol represents that there is no input in **Input** section, no output in **Output** section or no variation in **Variation** section. Additionally, when the *NULL* keyword appears within a function output, it means “do nothing”. For example, in Composition section, the following statement defines that **CreateElement** function of current feature may refer to the function output **Allocate** of feature *MemoryAllocation* or do nothing:

```
CreateElement(){  
     $\in$  MemoryAllocation.Allocate()? NULL;  
}
```

“!” --- This symbol defines the spatial dependencies among behavior outputs. More precisely, this symbol indicates the “Compose” relationship for behaviors output. Two possible combinations for this kind of dependency: structure output vs. function output; and structure output vs. function output (see *Composition* in subsection 3.1.2.1).

“;” – This symbol is used in Composition section to represent the end of one reference. For example, the following statement specifies that the function Insert of current feature will first refer the Allocate function of feature MemoryAllocation, then Push_back function of StructureForm. In this case, the symbol “;” is used to terminate each reference.

```
Insert(){  
    ↗ MemoryAllocation.Allocate();  
    ↗ StructureForm.Push_back();  
}
```

3.2: Applying Feature Contracts: Examples

Four examples are provided to illustrate the use of feature contracts: Abstract, List, Aircraft, and Manufacturing.

3.2.1: Abstract Example

Consider three abstract features (figure 3-12). Each of the nodes (A, B, and C) represents an individual feature based on some design view, and each edge indicates the possible interaction between two features. In this example, the selection on one feature may affect selections on other features. For example: Selecting a variant in feature A may constrain the possible options in feature B.

Figure 3-12: Three feature node interactions

The following statements show how our model represents interactions of three feature nodes as well as their behavior dependencies.

Name: A

Interaction:

? Variation: <Variation A> ? {a1, a2, a3}

? Contract:

B $\not\Leftarrow$ A & C $\not\Leftarrow$ A

Mapping

$\not\Leftarrow$ B.<Variation B> ? {b1} $\not\Leftarrow$ {a1, a2}

⌞ C.<Variation C> ? {c1} ⌞ {a3}

Behavior:

? Input: NULL

? Output:

Structure:

⌞ [Astruct]

? Composition: NULL

? Responsibility: (To be omitted)

Name: B

Interaction:

? Variation: <Variation B> ? {b1, b2, b3}

? Contract:

A ⌞ ⌞ B & C ⌞ ⌞ B

Mapping

⌞ A.<Variation A> ? {a1} ⌞ {b1}

⌞ C.<Variation C> ? {c1} ⌞ {b3}

Behavior:

? Input: NULL

? Output:

Function:

⌞ Bfunc()

? Composition:

⌞ A.[Astruct]!Bfunc()

? Responsibility: (To be omitted)

Name: C

Interaction:

? Variation: <Variation C> ? {c1, c2, c3}

? Contract:

A ⌞ ⌞ C & B ⌞ ⌞ C

Mapping

⌞ A.<Variation A> ? {a3} ⌞ {c1}

⌞ B.<Variation B> ? {b1} ⌞ {c1, c2}

Behavior:

- ? Input: NULL
- ? Output:
- Function:
- $\Leftarrow$ **Cfunc()**
- ? Composition:
- $\Leftarrow$ A.[Astruct]!Cfunc(){
- $\Leftarrow$ B.Bfunc();
- }
- ? Responsibility: (To be omitted)

Let us briefly explain what the above specification says. Selections made on features A, B, and C drives the generation of the single structural component **[Astruct]** and the associated behavioral components (functions *Bfunc* and **Cfunc**). Feature A decides what form of system structure is generated, that is **[Astruct]**, feature B provides variation on *Bfunc*, and feature C drives different realizations of **Cfunc**. The behavioral dependencies among features are that **[Astruct]** (varied by feature A) has one externally-visible function **Cfunc** (varied by feature C) and one internally-visible function *Bfunc* (varied by feature B).

Notice how feature contracts allow 1) a clear specification of a many-to-many mapping between features and the components that need to be generated, and 2) a direct representation of selection constraints. For example, if *b1* is selected in feature B, then feature A has to be one of variants *a1* and *a2*, while the feature C is constrained to the option subset *c1* and *c2*.

3.2.2: List Example

Consider the following feature model for a list in figure 3-13: Structure form feature allows for selection of either an array-based or a link-list-based implementation. Memory allocation feature has two variants (Bulk and Butterfly) where the Bulk option is

appropriate to the situation when all elements are inserted/deleted in blocks of insert/delete operations, and the Butterfly option is appropriate when insert/delete operations occur individually. Storage base feature has two options, Copy and Pointer, to store an element in the list. The ordering of elements in a list can be FIFO, LIFO and Priority-based. Finally, length is an optional feature, which may set up a fixed length constraint on a list.

Figure 3-13: List feature model

Let us now look at feature contracts. Each feature is first explained and then its contract is presented.

✍ StructureForm Feature

--Explanation:

This feature has two variants (array-based and link-list-based). There is a feature interaction for this feature: if the feature Length is *fixed*, this feature must be array-based. With respect to the behavioral outputs, this feature has one structural output [**Container**] which contains either an array structure if the array-based is applied, or a link-list

structure if link-list-based is selected. Meanwhile, this feature has a list of internal functions (e.g. *push_back*, *push_front*, *push_priority*, *pop_back*, *pop_front* and *pop_priority*) and one external function (*Initialize*). Those internal functions can be seen as elementary functions that will be referred by other features (e.g. *ListType*). Therefore, the temporal dependencies for those functions output will be specified in other features instead of this feature. However, there is a simple spatial dependency for this feature: all functions of this feature will be included in the structural output **[Container]**.

Name: StructureForm

Interaction:

? Variation: <type> ? {array-based, linklist-based}

? Contract: Length $\bowtie$ $\bowtie$ StructureForm

Mapping

$\bowtie$ Length.<Variability> ? {fixed } $\bowtie$ {array-based}

Behavior:

? Input: NULL

? Output:

Function & Structure:

$\bowtie$ Push_back()

$\bowtie$ Push_front()

$\bowtie$ Push_priority

$\bowtie$ Pop_back()

$\bowtie$ Pop_front()

$\bowtie$ Pop_priority

$\bowtie$ Initialize()

$\bowtie$ **[Container]**

? Composition

$\bowtie$ **[Container]!**Initialize()

$\bowtie$ **[Container]!**Push_back()

$\bowtie$ **[Container]!**Push_front()

$\bowtie$ **[Container]!**Push_priority()

$\bowtie$ **[Container]!**Pop_back()

$\bowtie$ **[Container]!**Pop_front()

$\bowtie$ **[Container]!**Pop_priority()

? Responsibility:

[Container] – a list container whose infrastructure could be either an array or a link list.

Push_back – a function that pushes an element into the end of a list.

Push_front – a function that pushes an element into the head of a list.

Push_priority – a function that pushes an element into a list based on a certain priority

Pop_back --- a function that pops an element from the end of a list.

Pop_front – a function that pops an element from the head of a list.

Pop_priority – a function that pops an element from a list based on a certain priority.

Initialize – a function that initializes all states of a list.

✂ MemoryAllocation Aspectual Feature

--Explanation:

A memory allocator may be used in many places (e.g. a tree structure, a vector structure), and so for the sake of reuse, it is designed to be an aspectual feature. Therefore, the type of data to be allocated by it is parameterized. As we can see in the *Input* subsection of the contract below, <<Data Type>> parameter represents the type of data to be allocated. This feature has two functional outputs: **Allocate** that allocates an element in memory based on the memory requirements of <<Data Type>> (see *Composition* subsection in the feature contract below) and **Deallocate** that remove a <<Data Type>> size from memory. This feature requests a structure ([Allocator]) to store the pre-allocated memory so that individual <<Data Type>> memory allocations can be done in constant time. The temporal dependencies for this feature are that the [Allocator] structure exports two functions (**Allocate** and **Deallocate**), and the [Allocator] is located inside of the [Container] in feature *StructureForm*. The variation of this feature between *Bulk* and *Butterfly* options means that **Allocate**, **Deallocate**, and [Allocator] may have widely different underlying implementations.

Name: MemoryAllocation

Interaction:

? Variation: <strategy> ? {Bulk, Butterfly}

? Contract: NULL

Behavior:

? Input: <<Data Type>>

? Output: NULL

Function & Structure:

✂ **Allocate()**

```

    ✎ Deallocate ()
    ✎ [Allocator]
? Composition
    ✎ StructureForm.[Container]![Allocator]!Allocate (){
        ✎ <<Data Type>>;
    }

    ✎ StructureForm.[Container]![Allocator]!Deallocate (){
        ✎ <<Data Type>>;
    }

? Responsibility:
    Allocate – a function to allocate an element-size in memory
    Deallocate – a function to remove an element-size memory.
    [Allocator] – a structure that stores a pre-allocated memory.

```

✎ **StorageBase Aspectual Feature**

--Explanation:

This feature provides two alternatives for storing an element: by copying the element value, or by referring to it through a pointer. Two functional outputs (*CreateElement* and *DestroyElement*) are coupled with the feature *MemoryAllocation* because *CreateElement* needs to allocate memory for the new element. In the interest of reuse this feature parameterizes the type of element. The Composition section of the feature contract below specifies that *CreateElement* instantiates the <<Data Type>> in feature *MemoryAllocation* by passing either the data type of elements to be created or the pointer type of those elements (Please note, the <<Data Type *>> implies the pointer type of elements like in C++). Similar to *CreateElement*, the *DestroyElement* function has the same coupling relationship with feature *MemoryAllocation*. Both functional outputs (*CreateElement* and *DestroyElement*) of this feature are located in the [Container] of feature *StructureForm* and they are internally-visible.

Name: StorageBase

Interaction:

? Variation: <type> ? {Copy, Pointer}

? Contract: NULL

Behavior:

? Input: <<Data Type>>

? Output:

Function:

✍ DestroyElement()

✍ CreateElement()

? Composition

```
✍ StructureForm.[Container]!CreateElement(){
  ✍ MemoryAllocation.Allocate(){
    {<<Data Type>> ? <<Data Type *>>} <=> MemoryAllocation.<<Data Type>>;
  }
}
```

```
✍ StructureForm.[Container]!DestroyElement(){
  ✍ MemoryAllocation.Deallocate(){
    {<<Data Type>> ? <<Data Type *>>} <=> MemoryAllocation.<<Data Type>>;
  }
}
```

? Responsibility:

CreateElement -- Create an element copy or a pointer to the element based on selected variant.

DestroyElement – Remove an element copy or pointer based on selected variant.

✍ Length Feature

--Explanation:

This feature provides an option to specify a fixed length for the List. If the option is selected, the list has to check the upper bound for insertion, lower bound for deletion, and expand the size of a list when the upper bound is reached. As specified in the *Output* subsection of the contract below, a list exports three functional outputs (*UpperBoundCheck*, *LowerBoundCheck* and *Expand*) for the elementary operations mentioned above. Meanwhile, another three functional outputs (*CheckUpBound*, *CheckLowBound* and *Resize*) will either refer to those functions or do nothing represented as **NULL** (see NULL in the *Composition*). In other words, three elementary functions (*UpperBoundCheck*, *LowerBoundCheck* and *Expand*) will be effective only if the variant *fixed* is selected. Finally, the spatial dependencies are that all function outputs of

this feature are coupled to the structure **[Container]** in feature *StructureForm*. Note that in the Interaction section below that Mapping specifies that a choice of the link-list based option in the StructureForm implies variable option for the length feature.

Name: Length

Interaction:

? Variation: <Variability> ? {fixed, variable}

? Contract: StructureForm. $\not\Leftarrow$ $\not\Leftarrow$ Length

Mapping

$\not\Leftarrow$ StructureForm.<type> ? {linklist-based} $\not\Leftarrow$ {variable}

Behavior:

? Input: NULL

? Output:

Function:

$\not\Leftarrow$ UpperBoundCheck()
 $\not\Leftarrow$ LowerBoundCheck()
 $\not\Leftarrow$ Expand()
 $\not\Leftarrow$ CheckUpBound()
 $\not\Leftarrow$ CheckLowBound()
 $\not\Leftarrow$ Resize()

? Composition

$\not\Leftarrow$ StructureForm.**[Container]!**UpperBoundCheck()
 $\not\Leftarrow$ StructureForm.**[Container]!**LowerBoundCheck()
 $\not\Leftarrow$ StructureForm.**[Container]!**Expand()

$\not\Leftarrow$ StructureForm.**[Container]!**CheckUpBound () {
 $\not\Leftarrow$ UpperBoundCheck() ? NULL;
}

$\not\Leftarrow$ StructureForm.**[Container]!**CheckLowBound () {
 $\not\Leftarrow$ LowerBoundCheck() ? NULL;
}

$\not\Leftarrow$ StructureForm.**[Container]!**Resize () {
 $\not\Leftarrow$ Expand() ? NULL;
}

? Responsibility:

UpperBoundCheck – a function that check if the upper bound of a list is met while the length of a list is fixed.

LowerBoundCheck -- a function that check if the lower bound of a list is met while the length of a list is fixed.

Expand – a function to expand a list if the list length is fixed.

CheckUpBound – a wrapper function that either refers the UpperBoundCheck or does nothing based on the selected variant.

CheckLowBound -- a wrapper function that either refers the LowBoundCheck or does nothing based on the selected variant.

Resize – a wrapper function that either refers the Expand or does nothing based on the selected variant.

✍ **ListType Aspectual Feature**

-- *Explanation:*

This feature allows for variability in list organization: *FIFO*, *LIFO* and *Priority* order. All the variants export two functions (*Insert* and *Delete*). *Insert*, has the following temporal dependencies: 1) perform *CheckUpperBound* function of feature *Length* (whether this is a null operation or an actual check depends on fixed/variable selection within the *Length* feature), 2) request to resize the list if necessary (again, determined by selection within feature *Length*), 3) obtain storage for the element based on <<Data Type>> and options in *StorageBase*, and 4) based on choice of organization perform one of three functions (*Push_front*, *Push_back* and *Push_priority*) from the feature *StructureForm*. Similar explanation holds for the *Delete* function.

The *Composition* section below defines the spatial dependency between this feature and the *StructureForm* feature: both externally-visible functions of this feature are bound to the **[Container]** structure of feature *StructureForm*.

We can see a chain of parameter instantiation through the three features: from *ListType* to *StorageBase*, from *StorageBase* to *MemoryAllocation*. Note that list users only see *Insert* and *Delete* operations, and so they provide the <<Data Type>> parameter to *ListType* feature which exports it to *StorageBase*, which in turn exports it to *MemoryAllocation*.

Name: ListType

Interaction:

? Variation: <order> ? {FIFO, LIFO, Priority}

? Contract: NULL

Behavior:

? Input: <<Data Type>>

? Output:

Function:

✍ **Insert()**

✍ **Delete()**

? Composition

```
✍ StructureForm. [Container]!Insert() {  
  ✍ Length. CheckUpperBound();  
  ✍ Length.Resize();  
  ✍ StorageBase.CreateElement(){  
    <<Data Type>> <=> StorageBase.<<Data Type>>;  
  }  
  ✍ StructureForm.{Push_back()? Push_front ? Push_priority()};  
}  
  
✍ StructureForm. [Container]!Delete(){  
  ✍ Length.CheckLowerBound();  
  ✍ StructureForm.{Pop_back()? Pop_front()? Pop_priority()};  
  ✍ StorageBase.DestroyElement(){  
    <<Data Type>> <=> StorageBase.<<Data Type>>;  
  }  
}
```

? Responsibility:

Insert – a function to insert an element into a list

Delete – a function to delete an element from a list.

3.2.3: Product Distributor Example

This example shows why a multiple-occurrence feature needs to decouple the behavior dependency using parameterization.

In daily operations of product distribution, a list of customer orders needs to be processed: *Prioritize the customer requests* and *Release inventory* for each of customer requests. Each of these operations has a set of alternatives.

Figure 3-14: Customer Order Process Scenario

On a typical day a distributor may receive a list of customer orders as follows:

Customer Account #	Customer Name	Ordered Item	Amount	Order Date	Order Due Date
00002	Wal Mart	Tide Detergent	100	02/02/2003	02/30/2003
00002	Wal Mart	Forger Coffee	100	02/3/2003	02/14/2003
00001	Loeb	Sunny D Juice	100	02/04/2003	02/12/2003
00003	Pharma Plus	Head & shoulder shampoo	200	02/02/2003	02/8/2003
00004	Blockbuster	Pringles Potato Chips	100	02/03/2003	02/10/2003

Table 3-3: A sample of customer orders

The distributor may have different strategies to prioritize the customer orders. For example, the “key account strategy” is based on the importance of a customer: assuming WalMart is the most important customer, then, regardless of who made the order first, the distributor will execute the WalMart order first. However, some other strategies may prioritize on request date or by request due date. There is clearly a need for variability in the strategy for prioritizing orders.

The next step is to carry out the order. Consider the following table.

Product	Location	Status	Expire Date	Amount
Tide Detergent	Toronto	On Hand	05/10/2003	200
Forger Coffee	Ottawa	On Hand	06/10/2003	200
Tide Detergent	Ottawa	Rework	05/14/2003	200
Tide Detergent	Montreal	Return Product	12/10/2003	300
Forger Coffee	Toronto	Rework	12/01/2003	100
Sunny D Juice	Ottawa	On Hand	12/20/2003	50
Pringles Potato Chip	Ottawa	Return Product	12/02/2003	100
Head & shoulder shampoo	Toronto	On Hand	12/14/2003	400

Table 3-4: A sample of inventory

Let us explain some terms that may not be obvious. A product may have the following status in the distribution center: *On Hand* indicates a product that is new and ready for distribution, *Re Work* indicates that there were quality problems that were identified but have been fixed, and *Return Product* indicates a customer return.

The distributor may also have different strategies for fulfilling the customer's order and releasing the inventory. For example, given a request for 100 units of Tide Detergent by WalMart, the *By Product Expiry Date* strategy may select the inventory

Tide Detergent Toronto On Hand 05/10/2003 200

On the other hand, the *By Inventory Status* may select product inventory based on the order *Return Product*, *Re Work*, *On Hand* because distributor always delivers the oldest product first. Under this strategy, the following inventory will be selected

Tide Detergent Montreal Return Product 12/10/2003 300

Alternatively, the *By Product Location* will deliver the inventory whose location is nearest the customer. In this case, given Ottawa is the location of customer WalMart, the following inventory will be selected:

Tide Detergent Ottawa Rework 05/14/2003 200

The feature model below reflects the variations in this simple example. Both operations *Prioritize Request Strategy* and *Release Inventory Strategy* need to sort a list based on a certain priority. However, there are two significant differences: the data to be sorted and the way to prioritize it. Further, the way to prioritize data will vary across different strategies for each operation. For example, *By Requested Date* and *By Key Account* strategies will construct a new customer request list based different priorities.

Figure 3-15: A simplified manufacturing feature

Given that sort is required both for prioritizing requests and for releasing inventory, let us consider if a single aspectual sort feature could be used. For *Prioritize Request Strategy*, while different kinds of strategy are being applied, the sort function (regardless of sort algorithm used) requests a *Comparator* that compares any two customer orders in a customer request list. The *Comparator* can be decoupled from the sort function

allowing the Prioritize Request strategy to specify the internal operation of the *Comparator*.

It is reasonable to parameterize both *Data* and *Comparator* in order to reduce coupling between a sort function and its clients, and allow for reuse of it as a generic component. Designed as such, a sort function becomes a good example of an aspectual feature. In this form, the sort function contains one variation (different sort algorithms), and two parameters (*Data* and *Comparator*). With this design, the clients such as *Prioritize Request Strategy* and *Release Inventory Strategy* can choose the sort algorithm and provide the parameters for that algorithm to function.

Let us now look at feature contracts. Each feature is first explained and then its contract is presented.

✂ **Sort Algorithm Aspectual Feature**

--*Explanation:*

The sort algorithm has two parameters (<<*Data*>> and <<*Comparator*>>) as its inputs, and one behavioral output: the Sort function that sorts the parameter <<*Data*>> by using the <<*Comparator*>> predicate. The sort algorithm also has a variation on the sort method. Finally, the **Composition** subsection specifies the temporal dependency of the *Sort* function: **Sort** function has to refer to the raw data (<<*Data*>>) to be sorted and the way to compare two elements (<<*Comparator*>>).

✂ Sort Algorithm Aspectual Feature

Name: Sort-Algorithm

Interaction:

? Variation: <Algorithm> ? {bubble sort, shell sort, quick sort}

? Contract: NULL

Behavior:

? Input: <<*Data*>>, <<*Comparator*>>

? Output:

Function:

```

    ✎ Sort()
? Composition
    ✎ Sort(){
        ✎ <<Data>>
        ✎ <<Comparator>>;
    }

? Responsibility:
    Sort --- sort the <<Data>> based on a selected algorithm and given comparator.

```

✎ Prioritize Request Feature

-- Explanation:

This feature contains a variation for how to order the customer requests. For this purpose, several strategies (by requested date/by request due date/by key account) can be employed. However, unlike the sort feature it does not require parameterization because it is used in only one place in the distribution domain. Therefore, it does not have parameters in the **Input** subsection. This feature has three internally-visible function outputs (*CompareByRequestDate*, *CompareByDueDate*, *CompareBykeyAccount*). In fact, those three functions share the same goal that is to compare two elements and return the judgment on the order of two elements. Meanwhile, each of three functions corresponds to the three variants specified in the **Variation** subsection (by requested date, by request due date, by key account). Therefore, this feature has an externally-visible function **Arrange** which will use the three internal functions. In the **Composition** subsection of the feature contract below, we define the temporal dependency for **Arrange** function as follows: 1) obtain a raw customer request list (the **getCustomerRequestList** function returns a customer request list, where the function itself is provided by some other, unspecified, feature) 2) the list is used to infer the <<Data>> parameter of *Sort-Algorithm* aspectual feature, 3) one of three internal functions will instantiate the <<Comparator>> parameter of *Sort* aspectual feature based on the selected variant (e.g. *by requested date/ by request due date/ by key account*), and 4) the **Arrange** function will use the **Sort** function of *Sort-Algorithm* aspectual feature to obtain a prioritized customer request list.

✍ Prioritize-Request Feature

Name: Prioritize-Request

Interaction:

- ? Variation: <Strategy> ? {by requested date, by request due date, by key account}
- ? Contract: NULL

Behavior:

- ? Input: NULL
- ? Output:

Function & Structure:

- ✍ CompareByRequestDate()
- ✍ CompareByDueDate()
- ✍ CompareByKeyAccount()
- ✍ **Arrange()**

- ? Composition

Arrange() {

```
✍ Sort-Algorithm.Sort(){
    OtherFeature.getCustomerRequestList() <=> Sort-Algorithm.<<Data>>;
    {CompareByRequestDate() ? CompareByDueDate() ? CompareByKeyAccount()}
    <=> Sort-Algorithm.<<Comparator>>;
}
```

- ? Responsibility:

- ✍ CompareByRequestDate – a function to compare two customer requests in order to justify the order of both requests based on request date.
- ✍ CompareByDueDate -- a function to compare two customer requests in order to justify the order of both requests based on request due date.
- ✍ CompareByKeyAccount -- a function to compare two customer requests in order to justify the order of both requests based on the importance of the customers.
- ✍ **Arrange** – a function to prioritize all receiving customer requests.

✍ **Release Inventory Feature**

-- *Explanation:*

This feature provides three internal function outputs (*CompareByExpireDate*, *CompareByInventoryDate*, *CompareByProductLocation*) and each of three outputs corresponds to each of three strategy variants (by expiry date, by inventory status, by

product location) of this feature. At the same time, one externally visible function output is exported, the **Release** function. Its behavior dependency is specified below in the composition section as follows: 1) use the function **Arrange** of *Prioritize-Request* feature, 2) infer the <<Data>> parameter of *Sort-Algorithm* aspectual feature from the inventory list (the **getInventoryList** returns a list of inventory, where the providing feature is left unspecified), 3) one of three internal functions will instantiate the <<Comparator>> parameter of *Sort* aspectual feature based on the selected variant (by expiry date, by inventory status, by product location), 4) the **Release** function will use the **Sort** function of *Sort-Algorithm* aspectual feature, and 5) based on the outcome (a prioritized customer requests list) of the **Arrange** function, each of customer requests will be satisfied by selecting the earliest inventory in the sorted inventory list (the final output of *Sort* function being called previously).

✍ Release-Inventory Feature

Name: Release-Inventory

Interaction:

? Variation: <Strategy> ? {by expire date, by inventory status, by product location}

? Contract: NULL

Behavior:

? Input: NULL

? Output:

Function & Structure:

```
✍ CompareByExpireDate()
✍ CompareByInventoryStatus()
✍ CompareByProductLocation()
✍ Release()
```

? Composition

```
✍ Release(){
    ✍ Prioritize-Request.Arrange();
    ✍ Sort-Algorithm.Sort(){
        OtherFeature.getInventoryList() <=> Sort-Algorithm.<<Data>>;
        {CompareByExpireDate() ? CompareByInventoryStatus() ?
          CompareByProductLocation()} <=> Sort-Algorithm.<<Comparator>>;
    }
}
```

? Responsibility:

```
✍ CompareByExpireDate --- a function to compare two inventories based on the expire date
```

- ✍ **CompareByInventoryStatus** --- a function to compare two inventories based on the inventory status
- ✍ **CompareByProductLocation** --- a function to compare two inventories based on the product location
- ✍ **Release** --- a function to select an appropriate inventory out of warehouse for a specific customer request.

Conclusion:

As we can see in this example, our model may not be able to explore all details for implementation of features. However, our model is able to roughly demonstrate the important behavioral dependencies among features such as the temporal dependencies in three features. At the same time, the important elements (e.g. variants, parameters) are involved in the dependencies description so that the model itself can represent integrated view for features such as parameter vs. behavioral output, variants vs. behavioral output, temporal dependencies, spatial dependencies (though may not found in this example but we present this dependency in some other examples) and interactions (see other example).

4. A Case Study

4.1 Game Plan

In this chapter, we illustrate the proposal presented in Chapter 3 through a case study. We have chosen a typical one: the elevator system. Contrary to the extensive one offered by Gomaa ([Gom00]) (summarized in Figure 4-1), we aim to model not a specific system (such as the centralized one that author adopts) but a family of designs. We will first discuss these variants then present how to capture in our proposed format and conclude this lengthy chapter with a brief comparison with what could have been achieved using GenVoca, our reference approach.

Figure 4-1: Gomaa's Elevator System

Our initial analysis of the domain will rely considerably on the use of the Use Case Map (UCM) notation (see [UCM]), a scenario modeling technique that emphasizes abstract causal paths. Other scenario modeling techniques, such as Message Sequence Charts ([ITU93]) could have been used but we found UCMs to be the simplest one to handle variability. We will use it in our initial analysis in order to derive sets of high-level functional requirements for the elevator system. The UCMs we obtain will serve two primary objectives: providing a global view on high-level system behavior and linking features and system behavior. More precisely, following the ideas of de Bruin ([BV02]), we will refine system behaviors in order to discover variation points and associate features with the latter. Once and only once features have been identified, we will then proceed to documenting each one using the proposed format.

As will become apparent below, the elevator system is of remarkable complexity and we do not claim an exhaustive list of features but the functional ones we suggest will suffice, we believe, to illustrate how to deal with complex feature interactions and behavioral dependencies.

4.2 About Use Case Maps

It is not our intent to provide here an introduction to Use Case Maps and to their role in an object-oriented approach to software development: such considerations can be found in ([BC96], [URN], [Amy03]). Let us simply briefly highlight the notions most relevant to our case study.

UCMs capture paths (i.e., temporal sequences) of system responsibilities. In a bound UCM, as opposed to an unbound one, responsibilities can be associated to components. For our purposes, the notion of a stub is of crucial importance. Roughly put, a stub (captured as a diamond) is a variation point: it acts as a place holder for a set of alternative behaviours (which may be detailed subsequently). Most interestingly, each of these alternative behaviours can itself be expressed in a UCM that may contain stubs. In

other words, from our standpoint, UCMs provide a hierarchical approach to variability, which explains our use of them. This discussion is summarized in Figure 4-2 below:

Figure 4-2 : Stubs in UCMs

4.3 Getting Started with the Elevator Control System

In a nutshell, elevator control systems are responsible for taking passengers from one floor to another. So, an elevator control system should control the activities of all elevators and schedule different elevators to respond to passenger requests. Two kinds of requests are possible: internal requests and floor requests. The former occurs while passengers are inside an elevator. In this case, passengers will select a specific floor number so that they can travel to that floor. Floor requests occur when a passenger “calls” an elevator by pressing the Up or the Down button available on that floor.

Following the work of Gomaa, we observe that:

- an elevator is physically composed of the following elements:

- A set of floor buttons** – a user presses a button to select a destination.

- An elevator motor** – controlled by commands to move up, move down and stop.

- An elevator door** – controlled by commands to open and close the door.

- A set of elevator lamps** – to indicate the floor currently visited by the elevator.

- for each floor, there are :

A pair of direction lamps – to indicate the direction(s) that have been requested.

Up and down floor buttons – a user presses a button to request an elevator.

A sensor – to detect the arrival of an elevator at the current floor.

We will focus our study mainly on how the system schedules elevators to service passengers' requests. For this reason we will not pay a lot of attention and simplify the issue of lamps and buttons: the actual user interface is not our concern here.

4.4 Scenario Analysis

Consider the highly abstract initial UCM of Figure 4.3.

Figure 4-3: An Initial UCM

In this UCM, we have three paths (each with a different starting point captured as a black circle). One path deals with floor requests, one with internal requests. The third one, on the right, deals with the possibility that upon having an elevator approach a floor, a sensor notifies the system of this event and that further processing proceeds from this

event. This third path is necessary to deal with so-called opportunistic systems, generally ignored in published elevator case studies. Intuitively, in these systems, floor requests are not immediately assigned to elevators. Instead, the system ‘waits’ until an elevator approaches a floor waiting to be serviced and then tells this elevator to stop at that floor provided it is moving in the right direction. (More details on this later).

Each path can now be described at a very high level of abstraction (akin to a use case):

Floor Request:

- ? Passenger (while on floor) makes a floor request for service.
- ? The floor request is sent to the *conceptual* control unit.
- ? Conceptual control unit receives and processes the floor request.
- ? Eventually, one elevator stops at the requesting floor.

Internal Request:

- ? Passenger (while inside of elevator) makes an internal request for service.
- ? The internal request is sent to the conceptual control unit.
- ? Conceptual control unit processes the internal request.
- ? Relevant elevator eventually stops at requested floor.

Approaching floor Event:

- ? Elevator approaches a floor.
- ? Corresponding sensor event (or message) is sent to conceptual control unit.
- ? Conceptual control unit processes the event.
- ? Possibly, the elevator is made to stop at this floor.

4.5 Selecting Features

Given these scenarios, we identify seven features and some of their variants. That is, we do not claim our features capture all possible variations in an elevator system. But they will be sufficient to illustrate complex interactions and behavioural dependencies. We regroup our features into three categories: System infrastructure, Communication, and Control logic. The first category pertains to what software components are to exist, the

second, to how components will exchange information, and the third, to strategies for servicing requests.

System infrastructure Category:

<i>Component Organization</i>

In general, there are many strategies to organize an elevator system into different components. Ideally, we envision the designer would have an application-engineering environment that would allow her to explore the ‘forces’ of each architecture in order to discriminate between them (e.g., with respect to non-functional requirements).

Conceptually, an elevator system has internal buttons (inside the elevators) that generate internal requests, floor buttons that generate floor requests, floor sensors that notice an elevator is approaching a floor, and elevators that carry passengers. We need not introduce more conceptual components in our case study: these will suffice to illustrate interactions. The feature **Component Organization** addresses which of these conceptual entities are realized in software. The table below lists three broad architectural alternatives.

Variant Id	Organization Type	Description of structure variants	Concrete components structure
1	Centralized	All system data and control is encapsulated into one single software component. This has the advantage of minimizing inter object communication...	singleton component
2	Decentralized	Each of our four conceptual components is realized in a software component which owns its state and offers a simple interface. For example, an elevator should have the information about its moving status; which floor it is on; and its assigned requests (both floor and internal ones). In terms of functionality, an elevator should export move up, move down and stop functions.	Elevator component Internal Button component Floor Button component Floor Sensor component
3	Hybrid (partially decentralized)	Here each conceptual component is realized into a concrete one and we introduce a central component, to centralize data and processing.	Elevator component Internal Button component Floor Button component Floor Sensor component Central component

Table 4-1 : Three types of architectures for elevator system

Both the decentralized and the partially decentralized architectures greatly contrast with the centralized one. In the latter, there is one and only one software component.

Therefore, there are no interactions between software components under this strategy: elevators, buttons and sensors are merely hardware that create/receive interrupts to which the unique software entity responds. Clearly, it follows that all control is also located in this sole component.

Figure 4-4 illustrates the decentralized strategy. Each cube here implies a software component that encapsulates the corresponding knowledge and some elementary procedures (e.g., internal button has a function to generate an internal request). Under the decentralized strategy, there are simply four interactions occurring in the system: Internal button components send internal requests to their corresponding elevator component; floor sensor components send the approaching floor message to elevator component(s); floor buttons send the floor request to elevator components; and elevators interact with one another.

This architecture implies that it is the elevators that have to decide whether or not to service a floor request and what is the next action for an elevator when it receives an approaching floor message. This consideration pertains to control logic, not to component organization per se. But, as a simplifying assumption, we will decree that if a decentralized component organization is adopted, then *necessarily* a distributed control strategy is also chosen. As explained later in this section, such a strategy has each elevator decide on its own whether or not to service the requested floors (possibly leading to duplication of service since we do not consider here cancellation strategies, another simplification on our part.). In other words, in contrast to the other two component organizations, there is no decision maker in this case !

Figure 4-4: Decentralized architecture

Figure 4-5 illustrates the partially decentralized architecture. Recall that here there is a specific decision to centralize information coming from floor and internal buttons as well as from floor sensors. This repository of data (i.e., decision maker) is used to control the elevators. More precisely, the central component has to decide i) which elevator is the best one to serve the floor request and ii) the next operation for an elevator while the elevator is approaching floor. As with the centralized architecture, here we have a single decision maker (which clearly *excludes* the use of distributed control logic!)

Figure 4-5: Partially decentralized architecture.

The following table summarizes how the different conceptual entities we introduced are realized (or not!) in the different architectures.

Conceptual component	partially		
	centralized	decentralized	decentralized
<i>[elevator]</i>	Singleton component	Elevator component	Elevator component
<i>[[floor sensor]]</i>	Singleton component	Floor sensor component	Floor sensor component
<i>[[floor button]]</i>	Singleton component	Floor button component	Floor button component
<i>[internal button]</i>	Singleton component	Internal button component	Internal button component
<i>[central object]</i>	Singleton component	Elevator component	Central component

Table 4-2 : Component realization in three architectures of elevator system

Assigned-Request-List

This feature has two possible variants: enabled or disabled. If present, it indicates that each elevator will be associated to a list of requests assigned to it.

Pending-Request-List

Similarly, a particular architecture may require that pending requests be centralized in a single object. We shall call this set of request the pending-request-list. For example,

opportunistic approaches (which use late decisions) will require such a list. Also, ownership of this list belongs to the software realization of the conceptual entity we called “central object”. Should this feature be disabled, then necessarily we must have a system using early decision (since requests cannot be stored for later processing).

Communication Category:

Communication

Communication strategies can be abstracted according to the following notions:

Sender: we assume any communication strategy has a single initiator.

Receiver: We have one or several receivers.

Operation: Any communication triggers some processing in the receiver. We generically refer to this as the operation of the communication.

Communication Strategy: How a sender reaches its receivers may vary. For simplicity, we will restrict ourselves to two simple cases illustrated below: synchronous broadcast simultaneously sends a message to one/more receivers from one sender, and asynchronous broadcast that sends a message to one/more receivers from one sender sequentially.

Figure 4-6a: asynchronous broadcast

Figure 4-6b: synchronous broadcast

Control logic Category:

For simplicity we assume that the domain dictates⁶ that all requests will be added into the *assigned request list* of an elevator in a circular order: any newly assigned request will be compared to the last stop along with the current direction of travel. If the newly requested floor is higher or lower than the last stop (i.e., the end of this direction), it will be added at the tail of the list to start the next direction cycle, otherwise, the floor request will be directly added into the current cycle, that is, in some position along the current direction. Figures 4-7a and 4-7b illustrate those two cases; both assuming that the current floor is 5 and the moving direction is down. In the figure 4-7a, the new request (floor 2) will be added into the current cycle because it is somewhere along the current direction. Conversely, in figure 4-7b, the new request (floor 6) will be added into the next cycle since the elevator cannot go to floor 6 along the current direction.

Figure 4-7a: a request that is along with the current direction can be added into the current cycle.

Figure 4-7b: a request that is not ahead in the current direction has to be added into the next cycle. (i.e. next direction)

⁶ Here our simplifying assumption is to assume that the way to store assigned requests is the same for all elevator systems.

Given this approach to maintain assigned requests, we now consider how to select the next destination among these once an elevator approaches a floor and is committed to stop at that floor.

Elevator-Moving-Strategy

Variants:

Nearest Floor First: Under this strategy, the elevator always selects, from the *assigned request list*, the request that has the shortest distance with the current floor. This (unlikely!!) strategy implies that the elevator may not maintain a consistent direction and the passenger may feel an elevator is oscillating. We consider it strictly for illustrative purposes.

One End: an elevator always moves in one direction to the end of the requests in that direction, before switching to the other direction.

Select-Pending-Request

As previously mentioned, some systems may delay decisions on floor requests and use a (typically FIFO) pending-requests-list. For such systems, we must consider what happens when an elevator approaches a floor and when an elevator is idle. We consider several variants, not to be exhaustive, but to be able later to discuss complex interactions.

Variants:

All with the same direction: Two cases. First, if an elevator has no other destinations, that is, when it becomes idle, it picks up the first request in the list of pending requests. The direction will be determined by comparing this new destination with the current floor of that elevator. Second, while approaching a floor,

if an elevator has destinations in its current direction, then this strategy will assign all pending requests in that direction to this elevator!

One with the same direction: This is a simplification of the previous strategy. Whether the elevator becomes idle or is moving in a direction, it picks up the first request of the pending request list in that direction.

Idle with one : This is a simplification of the previous strategy. Here, only idle elevators are allowed to deal with the pending requests. More precisely, in this case, the next pending request is assigned to an elevator when it becomes idle.

Idle with all: Same as previous one with an added twist: here, when an elevator becomes idle, it picks up the next pending request, figures out its direction, and then picks up all other pending requests in that direction!

One-up-or-down: Here, when an elevator becomes idle, it picks up the next pending request. If it is approaching a floor, and there is a pending request in the same direction that is one floor away, then this pending request is picked up by that elevator.

Disable: For systems with no pending request lists...

Clearly some of these variants are affected by choice of elevator-moving strategy. More precisely, the strategies **All with the same direction**, **One with the same direction**, **Idle with all** require that the system adopt the **One End** moving strategy.

<i>Allocation-Scoring-Algorithm</i>

Assuming early decision based on “best elevator”, some scoring mechanism must be used to rank the different elevators. Here, we only provide some simple schemes; more complex schemes are possible but add little to our study with respect to coupling.

Variants:

Shortest distance: once a floor request is issued, each elevator will calculate a score based on how many floors it is currently away from the floor requested. The one with the shortest distance will be granted with the highest score and immediately be assigned this request.

Shortest distance with idle: Shortest distance is computed only for idle elevators. So it is the best of the idle elevators that is selected. If there are no idle elevators, then the request is added to the pending request list.

Less stop with same direction: for each elevator, if it is going in the requested direction of the floor, compute a score based on how many stops separate the elevator from the requested floor. The elevator with the fewest stops is assigned the floor request. If no elevator is moving in the right direction, then the floor request is added to the pending request list.

Less stop with both direction: same as previous except here all elevators, regardless of their current direction, are considered. For each elevator, the score takes into account the total number of stops (including those in the current direction even if a change of direction is required to service the floor request at hand). In this case, necessarily, the floor request will be immediately assigned to the winning elevator (rather than stored in the pending request list).

Disable: In some situations, we may not require a scoring function in an elevator system. For example, as wasteful as it may be, we may want to force all elevators to service each of floor requests. Having several elevators attempt to service a floor

request lies beyond the scope of this study but could be justified when we consider that elevators may experience failures.

Allocation-Decision-Time

Clearly, from the previous features, we see different possibilities for when a floor request is in fact assigned to an elevator:

Variants:

Early: the system will immediately commit one elevator to serve the floor request.

Late: when a floor request arrives, the system tries to select a ‘best’ elevator to service this floor request according some scoring strategy. Given some scoring strategies do not select a best elevator, the system may queue the floor request in the pending request list. Later, each time an elevator approaches a floor, this list is checked to see if a floor request could be serviced by stopping at that floor.

When approaching floor: wait until an elevator becomes idle then process the next pending floor request....

Distributed: when floor requests occur, the system distributes them to all elevators, so that each elevator may commit to servicing the floor request, even though there will be duplication of service (unless a cancellation policy is implemented, a facet of the system we decided to *not* model).

We repeat that only a decentralized architecture can use the latter, and only the latter!

4.6 Scenario Refinement

In this section, we link features to system behavior through scenarios. We first provide in Figure 4-8 a feature diagram. The overall system behavior is first divided into the elementary behaviors located in the second level (e.g, “Generate internal request”, “Receive Internal Request” , etc.). This level’s behaviors are obtained through the scenario analysis presented earlier. In the third level, we further refine the behaviors of the second level using the features we have just introduced. For instance, under the behavior “Process Floor Request”, one of behaviors (“Assign Floor Request” and “Store Floor Request”) will be alternatively selected by feature **Allocation-Decision-Time**. Furthermore, the **Allocation-Decision-Time** feature also decides if “Assign-Pending-Request” behavior will be enabled in the system while an elevator is approaching a floor.

Further correlation between features, responsibilities and system behavior can be found in the Appendices. In the Appendix A, we consider which functions and structures each feature should export. In Appendix B, we propose a refinement of the third level of Figure 4-8. For example, the “Assign Floor Request” can be decomposed into two small behaviors such as “Score” for elevator self-evaluation and “Commit Floor Request” to assign request to one/more elevators.

Figure 4-8: Elevator System Behavior Decomposition using Features.

Our goal now is to refine the original scenarios presented in Figure 4-3.

Figure 4-9 depicts a refinement for the stub *process floor request* shown in figure 4-3. Scoring is optional, and so is using the pending request list. The intent is to have the stubs in 4-9 be variation points corresponding to the features introduced in the previous section.

Figure 4-10 refines the stub *process approaching floor message* in figure 4-3.

Integrating figures 4-9 and 4-10 with figure 4-3 leads to the detailed use case map of Figure 4-11, which is extremely useful because it shows features in context, that is, it

shows when the different features we have identified are used in system behavior. In turn, this will help understanding feature coupling.

Figure 4-9: A plug-in for process floor request

The numbers in the figure indicate features that have impact on the corresponding behaviors.

1. Allocation-Scoring-Algorithm	2. Allocation-Decision-Time
3. Select-Pending-Request	4. Elevator-Moving-Strategy
5. Communication	

Figure 4-10: A plug-in for processing approaching floor message

1. Elevator-Moving-Strategy	2. Select-Pending-Request
3. Allocation-Decision-Time	4. Communication

Figure 4-11: A refined scenario mapping behaviors with features.

1. Component-Organization	2. Communication	3. Allocation-Decision-Time
4. Allocation-Scoring-Algorithm	5. Pending-Request-List	6. Select-Pending-Request
7. Elevator-Moving-Strategy		

4.7 Feature Interactions

We now provide a table that summarizes the interactions we identify between the different features we have proposed.

	Component Organization			Elevator Moving Strategy		Select Pending Request					Allocation Scoring Algorithm					Pending Request List			Allocation Decision Time			
	Centralized	Decentralized	Partially Decentralized	Nearest floor first	One end	All with the same direction	One with the same direction	Idle with one	Idle with all	One-up-or-down	Disable	Shortest distance	Shortest distance with idle	Less stop with same direction	Less stop with both direction	Disable	Enable	Disable	Early	Late	Distributed	When approaching floor
Centralized				DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	T	T	F	T
Decentralized				DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	F	F	T	F
Partially Decentralized				DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	DC	T	T	F	T
Nearest floor first						F	F	T	F	T	T	T	F	F	T	T	DC	DC	DC	DC	DC	DC
One end						T	T	T	T	T	T	F	F	T	T	T	DC	DC	DC	DC	DC	DC
All with the same direction												F	T	T	F	T	T	F	F	T	F	T
One with the same direction												F	T	T	F	T	T	F	F	T	F	T
Idle with one												F	T	T	F	T	T	F	F	T	F	T
Idle with all												F	T	T	F	T	T	F	F	T	F	T
One-up-or-down												F	T	T	F	T	T	F	F	T	F	T
Disable												T	F	F	T	T	F	T	T	F	T	F
Shortest distance																	F	T	T	F	F	F
Shortest distance with idle																	T	F	F	T	F	F
Less stop with same direction																	F	T	T	F	F	F
Less stop with both direction																	T	F	F	T	F	F
Disable																	F	T	F	F	T	T
Enable																			F	T	F	T
Disable																			T	F	T	F
Early																						
Late																						
Distributed																						
When approaching floor																						

Table 4-3: A summary of features interactions in elevator system

T -- No violation F -- Violation DC -- Do not care

Please recall that our claim is not that the proposed set of features or the feature interactions identified here are complete or correct. We merely want to put in place a sufficiently-complex example to show how the format we propose to capture coupling information between features can be used. This is our next task.

4.8 Documenting Feature Coupling

This section applies descriptive feature model presented in Chapter 3 to document all features we propose for an elevator system. The interactions and dependencies among features are also explicitly expressed.

System Infrastructure Category:

Component-Organization Feature

Name: Component-Organization

Interaction:

? Variation: <Organization Type> ? {Centralized, Decentralized, Partially decentralized}.

? Contract:

Allocation-Decision-Time $\bowtie$ $\bowtie$ Component-Organization

Mapping

Allocation-Decision-Time.<decision time> ? {Early, Late, When approaching floor} $\bowtie$ {Centralized, Partially decentralized}

Allocation-Decision-Time.<decision time> ? {Distributed} $\bowtie$ {Decentralized}

Behavior:

? Input: NULL

? Output:

Structure & Function:

Structure:

[elevator] [floor button] [floor sensor] [internal button] [central object] [local list]

Function:

$\bowtie$ Stop()

$\bowtie$ MoveUp()

$\bowtie$ MoveDown()

- ✍ **addAssignedRequest()**
- ✍ **getFirstAssignedRequest()**
- ✍ **removeAssignedRequest()**

? Composition:

- ✍ **[elevator] ! Moveup()**
- ✍ **[elevator] ! Movedown()**
- ✍ **[elevator] ! Stop()**
- ✍ **[elevator] ! [local list]**
- ✍ **[elevator] ! addAssignedRequest()**
- ✍ **[elevator] ! getFirstAssignedRequest()**
- ✍ **[elevator] ! removeAssignedRequest()**

? Responsibility:

✍ **[elevator]:**

- + **Moveup()** --- a function to move elevator along the up direction.
- + **Movedown()** --- a function to move elevator along the down direction.
- + **Stop()** --- set moving status as idle.
- + **[Local list]** --- a list to store assigned requests.
- + **addAssignedRequest()** --- add an assigned request into the local list in circular order.
- + **getFirstAssignedRequest()** --- return the first element of the local list based on circular order.
- + **removeAssignedRequest()** --- remove a request from the local list.

Pending-Request-List Feature

Name: Pending-Request-List

Interaction:

? Variation: <Available> ? {Enable, Disable}

? Contract:

Select-Pending-Request ✍ ✍ Pending-Request-List &
Allocation-Scoring-Algorithm ✍ ✍ Pending-Request-List &
Allocation-Decision-Time ✍ ✍ Pending-Request-List

Mapping

Select-Pending-Request.<checkout strategy> ? {All with same direction, One with same direction, Idle with one, Idle with all, One-up-or-down} ✍ {Enable}

Select-Pending-Request.<checkout strategy> ? {Disable} ⚡ {Disable}

Allocation-Scoring-Algorithm.<score algorithm> ? {Shortest distance with idle, Less stop with both direction} ⚡ {Enable}

Allocation-Scoring-Algorithm.<score algorithm> ? {Shortest distance, Less stop with same direction, Disable} ⚡ {Disable}

Allocation-Decision-Time.<decision time> ? {Late, When approaching floor} ⚡ {Enable}

Allocation-Decision-Time.<decision time> ? {Early, Distributed} ⚡ {Disable}

Behavior:

? Input: NULL

? Output:

Structure:

[global list]

push()

? Composition:

⚡ **Component-Organization.[central object]![global list] ! push()**

? Responsibility:

⚡ [global list] --- a structure to store the pending requests which have not been assigned to any elevator.

⚡ **push()** --- append a request into the list with time priority.

Select-Pending-Request Feature

Name: Select-Pending-Request

Interaction:

? Variation: <checkout strategy> ? {All with the same direction, One with the same direction, Idle with one, Idle with all, On-up-or-down, Disable}

? Contract:

Elevator-Moving-Strategy ⚡ ⚡ Select-Pending-Request &

Allocation-Scoring-Algorithm ⚡ ⚡ Select-Pending-Request &

Pending-Request-List ⚡ ⚡ Select-Pending-Request

Mapping

Elevator-Moving-Strategy.<strategy type> ? {Nearest floor first} ⌘ {Idle with one, One-up-or-down, Disable}

Allocation-Scoring-Algorithm.<score algorithm> ? {Shortest distance, Less stop with both direction} ⌘ {Disable}

Allocation-Scoring-Algorithm.<score algorithm> ? {Shortest distance with idle, Less stop with the same direction} ⌘ ~{Disable}

Pending-Request-List. <Available> ? {Enable} ⌘ ~ {Disable}

Pending-Request-List. <Available> ? {Disable} ⌘ {Disable}

Behavior:

? Input: NULL

? Output:

Function:

⌘ **CheckOutPendingRequest()**

⌘ **CheckInPendingRequest()**

? Composition:

⌘ Component-Organization.[**central object**] ! **CheckOutPendingRequest()** {
 ⌘ Pending-Request-List.[global list] ? NULL;
}

⌘ Component-Organization.[**central object**] ! **CheckInPendingRequest()** {
 ⌘ Pending-Request-List.[global list].**push()** ? NULL;
}

? Responsibility:

CheckOutPendingRequest() --- when elevator arrives at a floor, this function will, based on the selected strategy, pick up some suitable pending request(s) to assign to the elevator, then, remove those requests from the pending request list. If this feature is *Disable*, this will do nothing;

Otherwise, this function requires to navigate the [global list] structure in order to pick up the suitable requests.

CheckInPendingRequest() --- when a new floor request arrives, this function is responsible for adding the incoming floor request into the *pending request list* (i.e. [global list]) if this feature is not *Disable*, the request will be appended into the list in FIFO order by referring the **push** function that is offered by feature **Pending-Request-List**. However, if this feature is *Disable*, this function may do nothing.

Communication-related Category:

Communication Aspect

Name: Communication

Interaction:

Variation: <strategy> ? {direct call, routing, broadcast}

Contract: NULL

Behavior:

? Input: <<sender>>, <<receiver>>, <<operation>>

? Output:

Function:

✎ **Send()**

? Composition:

✎ <<receiver>>!<<operation>>

✎ <<sender>>!Send(){

✎ <<receiver>>.<<operation>>;

}

? Responsibility:

Send() --- provide a generic communication interface between two sides.

Control-related Category:

Elevator-Moving-Strategy Feature

Name: Elevator-Moving-Strategy

Interaction:

Variation: < strategy type> ? {Nearest floor first, One end}

Contract:

Select-Pending-Request $\bowtie$ Elevator-Moving-Strategy &
Allocation-Scoring-Algorithm $\bowtie$ Elevator-Moving-Strategy

Mapping

Select-Pending-Request.<checkout strategy> ? {All with the same direction, One with the same direction, Idle with all} $\bowtie$ {One end}.

Allocation-Scoring-Algorithm.<score algorithm> ? {shortest distance, shortest distance with idle} $\bowtie$ {Nearest floor first}

Allocation-Scoring-Algorithm.<score algorithm> ? {Less stop with same direction, Less stop with both direction} $\bowtie$ {One end}

Behavior:

? Input: NULL

? Output:

Function:

CheckOutNextMovement()

CommitRequest()

GetDirectionOfNextRequest()

RemoveCommittedRequest()

? Composition:

$\bowtie$ Component-Organization.[elevator] !GetDirectionOfNextRequest(){

$\bowtie$ Component-Organization.[local list];

}

$\bowtie$ Component-Organization. [elevator] !CheckOutNextMovement(){

$\bowtie$ Component-Organization.[local list];

$\bowtie$ Component-Organization.{Movedown() ? Moveup() ? Stop()};

}

$\bowtie$ Component-Organization. [local list] !CommitRequest(){

$\bowtie$ Component-Organization.addAssignedRequest();

```

    }
    ↗ Component-Organization. [local list] !RemoveCommittedRequest() {
        ↗ Component-Organization.[local list].removeAssignedRequest();
    }

```

? Responsibility:

CheckOutNextMovement() --- this function is responsible for selecting the next assigned request from the assigned request list (i.e. [local list]) based on the selected strategy, then removing the request from assigned request list. If no more request, maintain the elevator moving status as idle.

CommitRequest() --- add a committed floor request into conceptual elevators local list in circular order, however, this condition is able to be removed in future since we might allow this behavior to be varied.

GetDirectionOfNextMovement() --- return the direction of first element in the *assigned request list*(i.e. [local list]).

RemoveCommittedRequest() --- remove an committed request from the *assigned request list*(i.e. [local list]) while an elevator is approaching.

Allocation-Scoring-Algorithm Feature

Name: Allocation-Scoring-Algorithm

Interaction:

? Variation: <score algorithm> ? {shortest distance, shortest distance with idle, less stop with same direction, less stop with both direction, disable}

? Contract:

Elevator-Moving-Strategy ↗ ↗ Allocation-Scoring-Algorithm &

Select-Pending-Request ↗ ↗ Allocation-Scoring-Algorithm &

Allocation-Decision-Time ↗ ↗ Allocation-Scoring-Algorithm

Mapping

Elevator-Moving-Strategy.<strategy type> ? {Nearest floor first} ↗ ~{Less stop with the same direction, Less stop with both direction}

Elevator-Moving-Strategy.<strategy type> ? {One end} ⌘ ~{Shortest distance, Shortest distance with idle}

Select-Pending-Request.<checkout strategy> ? {All with the same direction, One with the same direction, Idle with one, Idle with all, One up-or-down} ⌘ ~{shortest distance, Less stop with both direction}

Select-Pending-Request.<checkout strategy> ? {Disable} ⌘ ~{Shortest distance with idle, Less stop with same direction}

Allocation-Decision-Time.<decision time> ? {Distributed, When approaching floor} ⌘ {Disable}

Allocation-Decision-Time.<decision time> ? {Early} ⌘ {Shortest distance, Less stop with same direction}

Allocation-Decision-Time.<decision time> ? {Late} ⌘ {Shortest distance with idle, Less stop with both direction}

Behavior:

? Input: NULL

? Output:

Function:

Score()

? Composition:

⌘ Component-Organization.[elevator] ! Score(){
 ⌘ Component-Organization.[local list];
}

? Responsibility:

⌘ **Score()** --- provides a self evaluation for an in-coming floor request based on the selected strategy.

Allocation-Decision-Time Feature

Name: Allocation-Decision-Time

Interaction:

? Variation: <decision time> ? {Early, Late, When approaching floor, Distributed}

? Contract:

Select-Pending-Request $\not\sim$ Allocation-Decision-Time &

Allocation-Scoring-Algorithm $\not\sim$ Allocation-Decision-Time &

Pending-Request-List $\not\sim$ Allocation-Decision-Time

Mapping

Select-Pending-Request.<checkout strategy> ? {Disable} $\not\sim$ {Early, Distributed}

Select-Pending-Request.<checkout strategy> ? {All with the same direction, One with the same direction, Idle with one, Idle with all, One-up-or-down} $\not\sim$ {Late, When approaching floor}

Allocation-Scoring-Algorithm.<score algorithm> ? {Shortest distance, Less stop with same direction} $\not\sim$ {Early}

Allocation-Scoring-Algorithm.<score algorithm> ? {Shortest distance with idle, Less stop with both direction} $\not\sim$ {Late}

Allocation-Scoring-Algorithm.<score algorithm> ? {Disable} $\not\sim$ {Distributed, When approaching floor}

Pending-Request-List. <Available> ? {Enable} $\not\sim$ {Late, When approaching floor}

Pending-Request-List.<Available> ? {Disable} $\not\sim$ {Early, Distributed}

Component-Organization.<Organization Type> ? {Decentralized} $\not\sim$ {Distributed}

Component-Organization.<Organization Type>? {Centralized, Partially Decentralized} $\not\sim$ {Early, Late, When approaching floor}

Behavior

Input: NULL

Output:

Function:

AssignRequest()

StoreRequest()

```

RemoveRequest()
PendingRequestCheck()
genInternalRequest()
genApproachingFloor()
genFloorRequest()
receiveFloorRequest()
receiveInternalRequest()
receiveApproachingFloor()
ProcessApproachFloor()
ProcessFloorRequest()
ProcessInternalRequest()

```

Composition

// Internal request

```

⌘ Component-Organization. [internal button] ! genInternalRequest(){

```

//For distributed (decentralized), the central object is the elevator approaching floor. For others, the central object is either the Central Component or the Singleton Component

```

⌘ Communication.send() {
    @ [internal button] <=> Communication.<<sender>>;
    @ [central object] <=> Communication.<<receiver>>;
    receiveInternalRequest() <=> Communication.<<operation>>;
}
}

```

```

⌘ Component-Organization. [central object]!receiveInternalRequest(){
    ⌘ processInternalRequest();
}

```

```

⌘ Component-Organization. [central object]!ProcessInternalFloor(){
    //Add the request into the elevator local list
    ⌘ Communication.send() {
        @ [central object] <=> Communication.<<sender>>;
        @ [elevator] <=> Communication.<<receiver>>;
        Elevator-Moving-Strategy.CommitRequest() <=> Communication.<<operation>>;
    };
}

```

// Approaching floor message

```

⌘ Component-Organization. [floor sensor] ! genApproachingFloor(){

```

//The corresponding central object receives the approaching floor message. For the distributed (decentralized), the central object is the elevator approaching the floor. For other strategies, the central object is either the Central Component or the Singleton Component.

```

    Communication.send(){
        @[floor sensor] <=> Communication.<<sender>>;
        @ [central object] <=> Communication.<<receiver>>;
        receiveApproachingFloor() <=> Communication.<<operation>>;
    }
}

    Component-Organization.[central object]!receiveApproachingFloor {
    processApproachingFloor ();
}

    // For late or when approaching floor, then add the appropriate pending requests.

    Component-Organization.[central object]!PendingRequestCheck(){

        //Check out the next direction of the current elevator (in local list)
    Communication.send(){
        @[central object] <=> Communication.<<sender>>;
        @[elevator] <=> Communication.<<receiver>>;
        Elevator-Moving-Strategy.GetDirectionOfNextMovement() <=>
        Communication.<<operation>>;
    };

        //See if any pending requests have the same direction as the next direction
    Select-Pending-Request.CheckOutPendingRequest();

        //Then add those pending requests into the local list if any
    Communication.send(){
        @ [central object] <=> Communication.<<sender>>;
        @ [elevator] <=> Communication.<<receiver>>;
        Elevator-Moving-Strategy.CommitRequest() <=> Communication.<<operation>>;
    }
}

    // For distributed, ask all elevators to remove the requests; otherwise, just remove the
    request
    from the current elevator.

    Component-Organization.[central object]!RemoveRequest() {
    Communication.send(){
        @ [central object] <=> Communication.<<sender>>;
        @ ? ? [elevator] <=> Communication.<<receiver>>;

```

```

        Elevator-Moving-Strategy.RemoveCommittedRequest ()<=>
        Communication.<<operation>>;
    }
}

✎ Component-Organization.[central object]!ProcessApproachFloor() {

    //PendingRequestCheck is only for late or when approaching floor. It is not required for
    early or distributed.
    ✎ PendingRequestCheck() ? NULL;

    //Then remove the request from the local list (for distributed, this requires to remove it
    from all elevators)
    ✎ RemoveRequest();

    //Move to the next floor
    ✎ Communication.send(){
        @ [central object] <=> Communication.<<sender>>;
        @ [elevator] <=> Communication.<<receiver>>;
        Elevator-Moving-Strategy.CheckOutNextMovement() <=>
        Communication.<<operation>>;
    }
}

// Floor request

✎ Component-Organization. [floor button] ! genFloorRequest(){

    //For distributed (decentralized), all central objects (elevators) receive the request. For
    others, because only one central object is available in the system, all requests will send to
    this unique central object.
    ✎ Communication.send() {
        @[floor button] <=> Communication.<<sender>>;
        @? ? [central object] <=> Communication.<<receiver>>;
        receiveFloorRequest() <=> Communication.<<operation>>;
    }
}

✎ Component-Organization. [central object]!receiveFloorRequest() {
    ✎ processFloorRequest();
}

//Only for distributed or early.
✎ Component-Organization.[central object]!AssignRequest(){

```

//For distributed (decentralized), the central object (an elevator) commits the request to the elevator itself; For early, the central object (Central Component or Singleton Component) commits the request to the elected elevator.

```

✎ Communication.send(){
    @[central object] <=> Communication.<<sender>>;
    @[elevator] <=> Communication.<<receiver>>;
    Elevator-Moving-Strategy.CommitRequest()<=> Communication.<<operation>>;
}
}

```

//Only for late or when approaching floor.

```

✎ Component-Organization.[central object]!StoreRequest(){
    ✎ Select-Pending-Request.CheckInPendingRequest();
}

```

```

✎ Component-Organization.[central object]!ProcessFloorRequest(){
    ✎ Communication.send(){
        @[central object] <=> Communication.<<sender>>;
        @? ? [elevator] <=> Communication.<<receiver>>;
        Allocation-Scoring-Algorithm.score() <=> Communication.<<operation>>;
    }? NULL; // NULL for distributed
}

```

//AssignRequest for early and distributed while StoreRequest is for late and when approaching floor

```

✎ AssignRequest() ? StoreRequest();
}

```

Responsibility:

✎ **[internal button]:**

+ **genInternalRequest()** --- generate a internal request.

✎ **[floor sensor]:**

+ **genapproachingFloor()** --- generate approaching floor message.

✎ **[floor button]:**

+ **genFloorRequest()** --- generate floor request.

✎ **[central object]:**

+ **receiveFloorRequest()** ---- receive a floor request.

+ **receiveInternalRequest()** --- receive an internal request.

- + **ReceiveApproachingFloor()** --- receive an approaching floor message.
- + **ProcessApproachFloor()** --- process the approaching floor message.
- + **ProcessFloorRequest()** --- process the floor request message.
- + **ProcessInternalRequest()** --- process internal request message.
- + **AssignRequest()** --- while any score strategy is available, system will assign an elevator to service the floor request based on the returning score, if no score strategy is available, then, system may commit all elevator to service that request depending on the selected strategy of this decision. This function will be used by the *ProcessFloorRequest*.
- + **StoreRequest()** --- store the incoming floor request into pending request list. This function will be used by the *ProcessFloorRequest*.
- + **RemoveRequest()** --- remove the committed request from the local list of elevator(s). This function will be used by the *ProcessApproachingFloor*.
- + **PendingRequestCheck()** --- Check out the possible pending requests for the approaching floor elevator. This function is required by late/when approaching floor strategy and used by *ProcessApproachingFloor*.

4.9 A Comparison to a State-of-the-Art Approach

Let us start by contrasting the above specification with the sort of feature model proposed in system family engineering. Figure 4-12 gives such a feature model for the elevator system using the features we previously introduced. Some coupling information is captured through the semantics of such a diagram (more specifically through the semantics of the arcs as explained in chapter 1).

But this diagram offers a very limited viewpoint: most feature interactions are missing and the semantics of such feature models do not deal with feature behavior dependencies. When comparing to the previous feature contracts, we cannot help notice how static such a feature model appears. For example, it's one thing to capture that "Pending Request list" is an optional feature (white dot). But to document when this feature is relevant is much more useful! And, minimally, we should be able to capture that if "Pending

Request list” is Disabled, then “Select from Pending Request List ” must also be Disabled (as we do in the contract section of this feature).

Figure 4-12: A feature diagram for elevator system

Clearly, the contract/mapping section spells out feature interactions in a very explicit and straightforward way. Let us briefly elaborate. In chapter 3, we introduce the notion of a “Feature Variation Space”. We observe that this space is readily captured by the Contract /Mappings section of each feature, whereas it is at best written separately from feature models, if not simply ignored. As an example, consider interactions between features “Allocation-Decision-Time” and “Allocation-Scoring-Algorithm”.

In “Allocation-Decision-Time”

?	Allocation-Scoring-Algorithm .<score algorithm>	?	{Shortest distance, Less stop with same direction}	⊗	{Early}
?	Allocation-Scoring-Algorithm .<score algorithm>	?	{Shortest distance with idle, Less stop with both direction}	⊗	{Late}
?	Allocation-Scoring-Algorithm .<score algorithm>	?	{Disable}	⊗	{Distributed, When approaching floor}

Table 4-4 : The Contract in “Allocation-Decision-Time” feature

In “Allocation-Scoring-Algorithm”

?	Allocation-Decision-Time .<decision time>	?	{Distributed, When approaching floor}	⊗	{Disable}
?	Allocation-Decision-Time .<decision time>	?	{Early}	⊗	{Shortest distance, Less stop with same direction}
?	Allocation-Decision-Time .<decision time>	?	{Late}	⊗	{Shortest distance with idle, Less stop with both direction}

Table 4-5 : The Contract in “Allocation-Scoring-Algorithm” feature

It is important to repeat that the specification of each feature is meant to be self-contained, so that a designer does not have to consult several feature specifications in

order to figure out their interactions. So the semantic redundancy between the last rule in each of the two boxes above is highly desirable in our opinion.

Futhermore, the proposed format associates responsibilities with features. For example, the responsibility “Process Floor Request” (ProcessFloorRequest()) is assigned to feature “Allocation-Decision-Time”. And we notice that ProcessFloorRequest() possibly requires the Score() responsibility exported by feature “Allocation-Scoring-Algorithm”. Thus, here the behavioral specification of a responsibility implicitly builds a dependency relationship to another responsibility. Consequently, our proposed format implicitly captures a behavioral hierarchy of responsibilities (which could be easily extracted and displayed by a tool).

Now let us consider what improvements the use of GenVoca brings up. Following the method of Czarnecki and Eisenecker ([CE00]), we come up with the following grammar (in which each feature ultimately ends up being a layer initially):

Component-Organization:	Centralized Decentralized Partially Decentralized
Communication:	Asynchronous Synchronous
Pending-Request-List:	Enable Disable
Select-Pending-Request:	All with the same direction One with the same direction Idle with one Idle with all One up or down Disable
Elevator-Moving-Strategy:	Nearest floor first One end
Allocation-Scoring-Algorithm:	Shortest distance Shortest distance with idle Less stop with the same direction Less stop with both directions Disable
Allocation-Decision-Time:	Early Late When approaching floor Distributed

Table 4-6 : Initial Genvoca Model for elevator system

Next in that method, high-level dependencies are considered. We obtain:

<u>Feature</u>	<u>Responsibility</u>
Component-Organization	Provide the component architecture for elevator system
Communication	Support two ends communication
Pending-Request-List	Offer a list to store the pending requests.
Select-Pending-Request	Supply an algorithm to select suitable pending requests from pending request list if the list is available
Elevator-Moving-Strategy	A strategy that guides an elevator to decide the next moving destination.
Allocation-Scoring-Algorithm	A self evaluation scheme for elevators whose ultimate goal is to elect one/many elevator(s) to service a floor request.
Allocation-Decision-Time	A strategy that states when and how the system elects one/many elevator(s) to service a floor request.

Third, we must identify the level of each layer. This means we have to clarify which feature depends on which other feature so that a lower layer can export itself as a parameter to the upper layer. This is the essential step of GenVoca, namely the hierarchizing of the layers. We remark that this task is extremely subjective if based solely on the above high-level dependencies. To remedy this problem we suggest going beyond what Czarnecki and Eisenecker call for and developing a dependency graph such as the one given in Figure 4-13 based on the analysis of this section.

This figure simply states that:

Component-Organization feature

Exports

Structures: [elevator] [central object] [local list]

Interface: MoveUp, MoveDown, Stop, Assign Request to [elevator], Remove request from [elevator]

Communication feature

Exports

Interface: Send

Pending-Request-List feature

Imports

Structure: [central object]

Exports

Structure: [global list]

Figure 4-13: A layer dependency flow for elevator system

Select-Pending-Request feature

Imports

Structure: [global list]

Exports

Interface: Check out Pending Request, Check in Pending Request

Elevator-Moving-Strategy feature

Imports

Structure: [elevator]

Interface: MoveUp, MoveDown, Stop, Assign request to [elevator], Remove request from [elevator]

Exports

Interface: Check out next movement of [elevator], Commit a request, Remove a request

Allocation-Scoring-Algorithm feature

Imports

Structure: [elevator], [local list]

Exports

Interface: Score

Allocation-Decision-Time feature

Imports

Interface: Check out Pending Request, Check in Pending Request, Check out next movement of [elevator], Commit a request, Remove a request, Send, Score

Be it visual or textual, this analysis allows us to indeed establish a hierarchy of behavioral composition. For example, the feature “Elevator-Moving-Strategy” depends on the feature “Component-Organization” because the export interface of the former depends on the elevator component exported by the latter. Ultimately we obtain the following GenVoca model:

Allocation-Decision-Time :

Early[Elevator-Moving-Strategy, Allocation-Scoring Algorithm, Communication]

| Late[Select-Pending-Request, Elevator-Moving-Strategy, Allocation-Scoring Algorithm, Communication]

| When approaching floor[Select-Pending-Request, Elevator-Moving-Strategy, Allocation-Scoring Algorithm, Communication]

| Distributed[Elevator-Moving-Strategy, Allocation-Scoring Algorithm, Communication]

Communication:

Asynchronous | Synchronous

Elevator-Moving-Strategy:

Nearest floor first[Component-Organization]

| One end[Component-Organization]

Allocation-Scoring-Algorithm:

Shortest distance[Component-Organization]

| Shortest distance with idle[Component-Organization]

| Less stop with the same direction[Component-Organization]

| Less stop with both directions[Component-Organization]

| [Component-Organization]

Select-Pending-Request:

All with the same direction[Pending-Request-List]

| One with the same direction[[Pending-Request-List]

| Idle with one[Pending-Request-List]

| Idle with all[Pending-Request-List]

| One up or down[Pending-Request-List]

| [Pending-Request-List]

Pending-Request-List:

[Component-Organization]

Component-Organization:

Centralized | Decentralized | Partially Decentralized

Table 4-7 : Final Genvoca Model for elevator system

This GenVoca model explicitly reflects *compositional* relationships among features in the elevator system, which is necessary in order to build a corresponding generator. But it is crucial to emphasize that here features are viewed essentially as black boxes. Because feature behavior is not refined, inter-feature dependencies remain coarse-grained. For example, we cannot know which aspect of a feature depends directly on a specific aspect of another feature. Thus, as mentioned at the onset of this dissertation, such detail design decisions will be ‘buried’ in the code of the generator.

Also, we must add that GenVoca simply cannot model what we called an aspect. For example, feature “Communication” should be construed as an aspect (i.e., as an input parameter, not as a layer per se) for other features. Such an approach, explicitly supported by our proposal, allows for further abstractions we denote:

<<Sender>>, <<Receiver>>, <<Operation>>

In other words, a feature that inputs “Communication” as a parameter can refer in its detailed specification to the sender, the receiver, the operation, without having to know what they are.

Finally, we remark that the GenVoca model still is weak in capturing feature interactions when compared to our Contract/Mappings sections.

In summary, we do believe the format we propose to capture detailed coupling semantics of features represents a significant improvement over the state-of-the-art approach we just compared it to.

5. Conclusions

5.1 Recapitulation

The problem we set out to investigate at the beginning of this dissertation was whether or not, in the context of generative approaches to software development, coupling can be explicitly addressed in the problem space. Our thesis is that coupling *can* indeed be explicitly addressed at the level of features, that is, in the problem space rather than in the solution space. We have proposed that feature coupling be decomposed into two facets, namely, *feature interactions* and *feature behavioral dependencies*. Much like the combination rules discussed by Czarnecki and Eisenecker ([CE00]), feature interactions view features as functional black boxes and document how features “affect” each other. Conversely, behavioral dependencies between features are to be specified by associating a list of functional “services” to each feature and documenting which services depend on others (of other features). Our proposed format for documenting feature interactions and feature behavioral dependencies proceeds from the seminal work of Helm and Holland ([HHG90]) on the use of *contracts* for the specification of complex behaviors. In chapter 3, we have developed at length the semantics we put forth for capturing feature coupling and we have illustrated with small examples every facet of our proposal. We have then consolidated our discussion into a single medium-sized case study and compared our approach to the state-of-the-art approach of Czarnecki and Eisenecker ([CE00]).

We believe our approach represents an improvement over the state-of-the-art in feature coupling. Our contention mainly rests on the fact that our proposal includes feature behavioural dependencies, which are generally not addressed in current feature modeling approaches. In turn, these dependencies being fine-grained due to their consideration of feature responsibilities, they appear to provide a conceptual bridge between the problem space and the solution space in domain engineering. It is from this perspective that we now briefly discuss future work we envision.

5.2 Future work

The goal of our research group's work, we repeat, is to make quality a pervasive concern in domain-driven software engineering. We believe this standpoint has two advantages. First, it promotes the role and importance of quality engineering early in the process, a highly desirable characteristic for software development ([Bin00]). Second, we believe bringing quality into domain-driven development enables test generation across a family of systems and improves the reuse of tests across this family ([Cor03]).

While it does not address quality engineering, the present dissertation can be viewed as a stepping stone to achieve some of the goals our research group has targeted. Let us elaborate.

First, clearly, our proposal must be applied to other domains. We envision our fellow researchers developing the feature contracts we put forth in domains already in the literature (e.g., matrix libraries) or investigated in depth by our group (in particular container and graph libraries, distributed election algorithms, etc.).

Second, our proposal must be tied by ongoing research in our group tying features to non-functional requirements. Recall that here we have restricted ourselves to functional considerations. But we envision our feature contracts could eventually be augmented with information pertaining to non-functional requirements. In particular, feature interactions could be related to trade-offs. A colleague has already started building an application engineering environment supporting this idea. A key idea of this environment is that it must be able to analyze our feature contracts in order to report possible excessive coupling to the user before the latter starts exploring different configurations of features. The environment should also use the feature contracts to scrutinize the configurations of features submitted by the use and report, for example, not only of their 'legality' but also on their consequences (if our contracts are tied to non-functional requirements).

Third, our feature contracts need to be also tied to another line of ongoing work in our group, namely the generation of domain tests. From this perspective, feature interactions and mappings are likely to be one of several possible sources to derive tests. Other sources include, for example, scenarios using stubs as explained elsewhere [Ibid.].

Fourth, because feature behavioral dependencies are fine-grained and bridge between the problem space and the solution space via the notion of responsibilities, we forecast they could be used to derive some test components.

From this perspective, we conclude our proposal is likely to act as the stepping stone for a lot more research!

6. References

- [Acc98] Rafael Accorsi. Exploring the Behavior of Feature Interaction via Stable Model. In workshop on Computation Aspects of Nonmonotonic Reasoning, Trento, Italy, June, 1998.
- [Amy03] D. Amyot. Introduction to the user requirements notation: learning by example. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, Volume 42, Issue 3, June 2003.
- [Atk01] C. Atkinson et al.. *Component-Based Product-Line Engineering with UML*. Addison-Wesley, 2001.
- [Bat85] D.S. Batory. Modeling the Storage Architectures of Commercial Database System. *ACM TODS*, Volume 10, Issue 4, pg 463 – 528, December 1985.
- [Bat87] D.S. Batory. Extensible Cost Models and Query Optimization in GENESIS. *IEEE Database Engineering*, 1987.
- [Bat88] D.S. Batory. Concepts for a Database System Compiler. *ACM POSD*, 1988.
- [Bat89] D.S. Batory. On the Reusability of Query Optimization Algorithms. *Information System*, 1989.
- [Bat90] D.S. Batory. The Genesis Database System Compiler: User Manual; U. of Texas Tech, Rep. TR-90-27, 1990.
- [Bat91] D.S. Batory and J.R. Barnett. Date: The Genesis DBMS software layout editor. Technical report, Department of Computer Sciences, University of Texas at Austin, 1991.
- [BBGSTTW90] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K. Tsukuda, B.C. Twichell, and T.E. Wise. GENESIS: An Extensible Database Management System; *IEEE Trans. Software Engineer* (pp. 1711-1730), 1990.
- [BBRTG89] D.S. Batory, J.R. Barnett, J.Roy, B.C. Twichell, and J. Garza. Construction of File Management Systems from Software Components; *COMPSAC*, 1989.
- [BC96] R. J. A. Buhr, R. S. Casselman. *Use Case Maps for Object-Oriented System*. Prentice Hall, 1996.
- [BCS99] B. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product-Lines. *The first Product Line Conference*, 1999
- [BFKLMSD99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J-M. Debaud. PuLSE: A Methodology to Develop Software Product Lines. *Symposium on Software Reuse (SSR'99)*, 1999.

- [Bin00] R. Binder. Testing Object-Oriented Systems. Addison-Wesley, 2000.
- [Bos00] J. Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-line approach. Addison-Wesley, 2000.
- [BV02] H. de Bruin, H. van Vliet. Top-Down Composition of Software Architectures. Pg 147-Electronic Edition (IEEE Computer Society DL), 2002.
- [CE00] K. Czarnecki and U.W. Eisenecker. Generative Programming Methods, Tools, and Application; Addison-Wesley, 2000.
- [CN01] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns: Addison-Wesley, 2001
- [Cor03] J.-P. Corriveau, Envisioning Domain-Driven Validation. TOOLS-USA, 2003.
- [Dij76] E. W. Dijkstra. A Discipline of Programming; Prentice Hall Englewood Cliffs, Nj, 1976.
- [ECOOP01] Feature interaction in Composed System in ECOOP Workshop 2001. Retrieved from:

<http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/program.html>
- [FFB02] D. Fey, R. Fajta, and A. Boros. Feature Modeling: A Meta-Model to Enhance Usability and Usefulness. Software Product Lines, The Second Software Product Conference, August, 2002.
- [FHS02] S. Ferber, J. Haag, and J.Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. Software Product Lines, The Second International Conference, SPLC 2, San Diego, CA, USA, August 2002.
- [GBS00] J. Gurp, J. Bosch, M. Svahnberg. Managing Variability in Software Product Lines. Landelijk Architecture Congress, 2000.
- [Gey00] L. Geyer. Feature Modeling Using Design Spaces. The first German Workshop on Product Line Software Engineering, Kaiserslautern, Germany, 2000.
- [GFD98] M. L. Griss, Favaro J and D'Alessandro M. Integrating Feature Modeling with the RSEB. Proceedings of ICSR98, Victoria, BC, IEEE, June 1998 (pp. 36-44).
- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

- [Gom00] H. Gomma. Designing Concurrent Distributed, and Real-Time Application with UML. Addison-Wesley, 2000.
- [Gri98] M. Griss. et al. Integrating Feature Modeling with the RSEB. Proceedings of Fifth International Conference on Software Reuse (ICSR'5), Victoria, Canada, June 1998 (pp. 76-85).
- [Gri00] M. L. Griss. Implementing Product-Line Features with Component Reuse. 6th International Conference, proceedings/ICSR-6, Vienna, Austria, June 27-29, 2000.
- [HA00] J. D. Hay and J. M. Atlee. Composing Features and Resolving Interactions. ACM International Symposium on the Foundations of Software Engineering (FSE), pp. 110-119, November 2000.
- [HHG90] R. Healm, I.M. Holland and D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. ACM: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, 1990.
- [HM84] R. A. Howard and J. E. Matheson. Influence Diagrams. Menlo Park, California, April 1984.
- [Hol92] I. M. Holland. Specifying reusable components using Contracts. Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP), 1992.
- [ITU93] International Telecommunication Union (ITU-T), Recommendation Z. 120. Message Sequence Chart (MSC), ITU-T. Geneva, 1993.
- [JGJ97] I. Jacobson, M. Griss and P. Jonsson. Software Reuse – Architecture, Process, and Organization for Business Success. Addison-Wesley, 1997.
- [JTM99] J.-M. Jezequel, M. Train, C. Mingins. Design Patterns and Contracts. Wiley 1999.
- [JZ98] M. Jackson, P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. IEEE Transactions on Software Engineering XXIV(10):831-847, October 1998)
- [KCHNP90] K Kang, S. Cohen, J. Hess, W. Nowak and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report No. CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 1990.
- [KLKSH88] S.J. Kim, J. Lee, K.J. Kim, S.H. Shin, and M.H. Huh. FORM: A Feature-Oriented reuse Method with Domain specific reference architectures; Annals of Software Engineering, Vol. 5, 1988 (pp.143-168).

- [KPR97] C. Klein, C. Prehofer and B. Rumpe. Feature specification and refinement with state transition diagrams. In P. Dini, ed., Fourth IEEE Workshop on feature interactions in telecommunications networks and distributed systems, IOS press, 1997.
- [LY01] L. Liu, and E. Yu. From Requirements to Architectural Design – Using Goals and Scenarios, ICSE-2001 Workshop: From Software Requirements to Architectures (STRAW 2001) May 2001, Toronto, Canada. pp. 22-30.
- [MA03] D. Muthig and C. Atkinson. Model-Driven Product Line Architectures. ECOOP 2003.
- [Mey94] B. Meyer. Reusable Software: The BASE Object-Oriented Component Libraries. Addison-Wesley, 1994.
- [Mey97] B. Meyer. Object-Oriented Software Construction, Second Edition. Prentice Hall Professional Technical Reference, 1997
- [Oma90a] S.W.O'Malley. Ph.D. Dissertation. University of Arizona, 1990.
- [Oma90b] S.W.O'Malley. Avoca: An Environment For Programming With Protocols, Ph.D. Dissertation. University of Arizona TR90-31, August 1990.
- [OAHP90] S .W.O'Malley, M.B. Abbott, N.C. Hutchinson, and L.L. Peterson. A Transparent Blast Facility. Journal of Internetworking, 1 #2, December 1990.
- [OMG] Object Management Group. <http://www.omg.com>
- [OT00] H. Ossher and P. Tarr. Multidimensional separation of concerns and the hyperspace approach. Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, 2000.
- [Par76] D. Parnas. On the Design and Development of Program Families. IEEE Transactions on Software Engineering, vol. SE-2, no. 1, pp. 1--9, 1976.
- [Rau03] A. Rauschmayer. Probe: Visualizing Algebraic Transformations in the GenBorg Generative Software Environment. International Conference on Software Engineering, 2003.
- [SB98] Y. Smaragdakis and D. Batory, Implementing Layered Designs with Mixin Layers, ECOOP '98.
- [SEI90] Proceedings of workshop on Domain-Specific Software Architectures. Software engineering Institute, July 1990.
- [Str97] B. Stroustrup. The C++ Programming Language (Special Edition). Addison-Wesley, 1997.

- [Sva00] M. Svahnberg et al. Research report: On the Notion of Variability in Software Product Lines. Blekinge Institute of Technology Research Report, 2000.
- [UCM] Use Case Map Organization. <http://www.usecasemaps.org>
- [URN] User Requirements Notation group. <http://www.usecasemaps.org/urn>
- [Vli02] H. van vliet. Software engineering: principles and practice. Wiley, 2n edition, 2002.
- [Wit96] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report, CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1996
- [WL99] D. M. Weiss and C. T. R. Lai. Software Product-line Engineering: A Family-Based Software Development Process. Reading, MA: Addison-Wesley, 1999.
- [WK03] R. Wirfs-Brock, A. McKean. Object Design: Roles, Responsibilities, and Collaborations. Addison-Wesley, 2003.
- [You97] E. Yourdon. Death March. Prentice Hall 1997.
- [Zav93] P. Zave. Feature interactions and formal specifications in telecommunications. IEEE Computer, vol. 26, no.8, pp. 20-28, Aug. 1993.

7. Appendices

Appendix A. Features Description For Elevator system

This section provides a table to demonstrate the responsibilities for the captured features of elevator system. The table itself also clarifies the tie between a feature and its corresponding decision to be extracted. Finally, a set of behavioural outputs is listed for each feature.

Feature	Decision to be captured	Responsibility	Exported Structure	Exported Function
Component-Organization	How to encapsulate elevator knowledge	Provide a system infrastructure in which different system knowledge are encapsulated into components and a set of elementary functions that must be available for the corresponding component.	[elevator]; [internal button]; [floor button]; [central object]; [floor sensor]	MoveUp; MoveDown; Stop (for elevator); AddAssignedRequest (for elevator); SelfAssignedRequest (for elevator)
Pending-Request-List	If allows to store the incoming floor request temporarily	Provide a list to store incoming floor request temporarily	[global list]	Push
Select-Pending-Request	How to check out the pending requests from the pending request list	Offer a strategy to select suitable pending requests from pending-request-list while the elevator approaching floor		CheckOutPendingRequest; CheckInPendingRequest
Communication	How to transmit the message between two ends of a communication	Build up a generic object-communication architecture		Send
Elevator-Moving-Strategy	How to check out the committed requests in order to move the elevator to next destination	Provide a strategy to decide the next movement of elevator while an elevator has a list of committed jobs		CommitRequest; CheckOutNextMovement; GetDirectionOfNextRequest; RemoveCommittedRequest
Allocation-Scoring-Algorithm	How to select the best of elevator to service an incoming floor request	Offer a strategy for each elevator to self-evaluate if the elevator itself is the best one to serve an incoming request		Score
Allocation-Decision-Time	When to make a decision on electing one more elevators to service an incoming floor request	This feature simply manipulates the availability of different system behaviors that are exported by other features. For example, if earlier decision is selected, then the scoring function for the elevator must be available. However, if this scoring function is not useful to the distributed decision since system will broadcast the floor request to all elevators once it is issued. Because this feature affects many system behaviors, it exports three major system behaviors "process floor request", "process internal request" and "process approaching floor message" that contains other small functions (e.g. score, check out pending request and etc.). By this way, this feature can decide which small function will be chosen.		GenerateInternalRequest (for internal button); GenerateFloorRequest (for floor button); GenerateApproachingFloorMessage (for floor sensor); ReceiveInternalRequest (for internal button); ReceiveFloorRequest (for floor button); ReceiveApproachingFloorMessage (for central object); StoreRequest (for central object); AssignRequest (for central object); RemoveRequest (for central object); PendingRequestCheck (for central object); ProcessFloorRequest (for central object); ProcessInternalRequest (for central object); ProcessApproachingFloorRequest (for central object)

Appendix B. Features Versus High-level Responsibilities For Elevator System

This section provides a table to explain how each captured feature in the elevator system is related to the high-level responsibilities, and the refined behaviours corresponding to those high-level responsibilities.

Elevator-Moving-Strategy	Rationale:	<p>This feature provides a function that inserts the assigned request into [local list]</p> <p>This feature provides a function that inserts the assigned request into [local list]</p> <p>Since the operation is involved with the [local list] this feature is responsible to return the first request in the [local list]</p> <p>This feature provides a function that inserts the assigned request into [local list]</p> <p>This feature provides a function that removes the current floor from the [local list]</p> <p>This feature provides a function that selects a request from [local list] as the next operation based on the selected strategy</p>											
	Imported:	no	yes	no	no	no	no	no	no	no	no	no	no
Allocation-Scoring-Algorithm	Rationale:	<p>Based on the select strategy, this feature provides a "score" operation for the [elevator]</p>											
	Imported:	no	no	yes	yes	yes	no	no	no	no	yes	yes	no
Allocation-Decision-Time	Rationale:	<p>This feature decides if the floor request will be assigned immediately (e.g. in early and dedicated) if so, it implies that the "score" function may be available for early decision, and the "commit" floor request operation has to be available in the system</p> <p>This feature decides if the floor request will be temporarily stored in [global list] (e.g. late and when</p> <p>For distributed, the [central object] [elevator] that is approaching floor, asks all [elevators] to remove the current floor from its committed request list (i.e. [local list]). For other strategies, the [central object] [longest] Component or Central Component asks only the [elevator] that is approaching floor, all the current floor from its committed request list (i.e. [local list])</p>											
	Imported:	no	no	yes	yes	yes	no	no	no	no	yes	yes	no

Appendix C. Grammar For Proposed Model

This section describes the grammar for our proposed model. The grammar description uses the EBNF syntax, where “:=” means *is defined as*, “|” means a choice, “?” optionality and “*” means zero or more times. In the description of the *feature name*, *variant name*, *variation name*, *parameter name* the syntax for lexical tokens from the JavaCC parser generator is used. See <http://www.suntest.com/JavaCC>.

variant-expression	:= (“~”)? “{”<variant name>“,”<variant name>* “}”
variation-expression	:= “<”<variation name> “>” “∈” variant-expression
interaction-expression	:= <feature name> “↔” <feature name> (“&” <feature name> “↔” <feature name>)*
mapping-expression	:= (<feature name>.”<variation name>“∈” variant-expression “→” variant-expression)*
parameter-expression	:= “<<”<parameter name>“>>”
input-expression	:= parameter-expression(“,”parameter-expression)*
structure-expression	:= (“@” “Δ”)? “[”<structure name>“]”
function-expression	:= <function name>“()”
output-expression	:= (structure-expression)* (function-expression)*
logical-operator	:= “⊕” “ ” “&”
logical-relation	:= “{”(structure-expression) (logical-operator structure-expression)* (function-expression) (logical-operator function-expression)* “}”
instantiation-relation	:= (function-expression structure-expression logical-relation)“<=>” (<feature name>.”)?parameter-expression
instantiation-expression	:= function-expression“{”(instantiation-relation “,”)* “}”
refer-expression	:= “{” (“→”(<feature name>.”)?(structure-expression function-expression instantiation-expression) “,”)* “}”
composition-relation	:= (<feature name>.”)?(structure-expression parameter-expression) (“!” (parameter-expression structure-expression

	function-expression))* (refer-expression)?
parameter name	:= "A"-"Z" ("a"-"z" "0"-"9" "A"-"Z" "_")*
feature name	:= "A"-"Z" ("a"-"z" "0"-"9" "A"-"Z" "_")*
variation name	:= "A"-"Z" ("a"-"z" "0"-"9" "A"-"Z" "_")*
variant name	:= "A"-"Z" ("a"-"z" "0"-"9" "A"-"Z" "_")*
structure name	:= "A"-"Z" ("a"-"z" "0"-"9" "A"-"Z" "_")*
function name	:= "A"-"Z" ("a"-"z" "0"-"9" "A"-"Z" "_")*