

**FORMALIZING AND VERIFYING DESIGN DECISIONS
IN SINGLE SYSTEMS AND SOFTWARE PRODUCT LINES**

HENG BOON KUI

(M.Tech (SE), NUS)

A THESIS SUBMITTED

**FOR THE DEGREE OF
MASTER OF SCIENCE**

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Heng Boon Kui

30 October 2013

Acknowledgements

I would like to thank all the following people who have facilitated me in various ways for my Master of Science study at Department of Computer Science, School of Computing, National University of Singapore.

Apart from providing guidance on my work, I owe immensely to my supervisor Associate Professor Stanislaw Jarzabek on many aspects – for agreeing to supervise me as a part-time student; for allowing me to work on my area of interest; for being critical in reviewing my work; for making effort to respond promptly to my queries; and for accommodating my schedule due to my day job.

I am also very grateful to my examiners Associate Professor Khoo Siau Cheng and Associate Professor Dong Jin Song. They had assessed and provided constructive feedbacks on my thesis proposals.

Being a part-time student, I am very thankful for the support and encouragement from the management of my employer, Institute of Systems Science, National University of Singapore. Without the support, I hardly find sufficient time to conduct my work.

Not forgetting the vice-deans and administration staff of the office of graduate studies, I would like to thank the vice-deans for approving my various requests. Thanks to Ms. Loo Line Fong for her facilitation on administrative matters. Thanks to Ms. Agnes Ang for her advices on the wrapping up of my work.

Last but not least, I must thank my wife, my son, and my parents for bearing with me for depriving them of my time during the course of my study.

Table of Contents

Acknowledgements	iii
Table of Contents	iv
Summary	viii
List of Figures	xi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Overview of Solution and Contributions.....	4
1.3 Organization of Thesis.....	7
Chapter 2 Problem	8
2.1 Problem Definition	8
2.2 Running Example	13
Chapter 3 Formalization of Abstract Syntax of DDM for Single Systems 23	
3.1 Elements of DDM.....	23
3.2 Dependencies between Elements of DDM	25
3.2.1 Issue occurrence-alternative Association	26
3.2.2 Issue occurrence-decision Association.....	26
3.2.3 Decision-alternative Association.....	27
3.2.4 Comprise Association	27
3.2.5 Constrain Association	28
3.2.6 Forbid and Resolve Associations	29
3.3 Trace Links	30

3.3.1	Feature-issue occurrence Trace	30
3.3.2	Decision-code Trace.....	31
Chapter 4	Impacts of Design Decisions for Single Systems	32
4.1	Order in Applying the Implications of Decisions.....	32
4.2	Evolution of Decision and its Ripple.....	34
4.2.1	Evolution of Decision.....	34
4.2.2	Ripple	35
4.3	Addition/removal of Elements of DDM.....	37
4.3.1	Issue occurrence-alternative Association	37
4.3.2	Issue occurrence-decision Association.....	37
4.3.3	Decision-alternative Association.....	38
4.3.4	Comprise Association	38
4.3.5	Constrain Association	38
4.3.6	Forbid and Resolve Associations	39
Chapter 5	Extension for Software Product Lines	40
5.1	Extension of the Running Example	40
5.2	Extension of the Abstract Syntax	43
5.2.1	Scoping of DDM based on Feature Configuration.....	43
5.2.2	Elements of DDM	44
5.2.3	Dependencies between Elements of DDM.....	45
5.2.4	Trace Links.....	45
5.3	Extension of the Impacts of Design Decisions	46
5.3.1	Evolution of Decision and its Ripple	46
Chapter 6	Validation by Usage Examples	48

6.1	Construction of DDM.....	49
6.2	Understanding the Impacts of DDM	52
6.3	Evolution of DDM.....	54
Chapter 7	Verification by Formal Method.....	57
7.1	Use of Formal Method.....	57
7.2	Alloy as a Formal Method Tool	58
7.3	Overall Verification Approach using Alloy	59
7.4	Specification and Verification of DDM and its Instances	60
7.5	Specification and Verification of Feature Model and its Instances 61	
7.6	Comparison of Planned vs. Supported Feature Configurations	62
7.7	Derivation of Information for a Feature Configuration from DDM 63	
7.8	Verification of Instances of DDM for the Addition and Removal of Elements of DDM.....	64
Chapter 8	Implementation of Support IDEs	65
8.1	Challenges for Tool Developers	65
8.2	Solutions to Challenges	66
8.2.1	Metamodel for DDM.....	66
8.2.2	Mapping Mechanism.....	67
8.2.3	Variability Technique.....	69
8.2.4	Metamodel for Feature Model.....	70
8.2.5	Ordering Mechanism and Prioritization Scheme	70
8.2.6	Ripple Mechanism.....	72
8.3	Implementation Technologies	73

Chapter 9	Evaluation against Design Activities in Development Processes	75
9.1	Benefits for the Design Activities of Single Systems.....	75
9.2	Benefits for the Design Activities of SPLs.....	77
Chapter 10	Related Works	79
Chapter 11	Conclusion.....	81
11.1	Achievements	81
11.2	Future Works	81
	Bibliography.....	83
Appendix A	Formalization of the Running Example.....	85
A.1	Formalization for Single System	85
A.2	Formalization for SPL	87
Appendix B	Source Code of the Running Example	89

Summary

A software system is designed to fulfill both its functional requirements and quality attributes. As the system is designed, the design issues (e.g., the existence of duplicate copies of the same object) that occur have to be solved by applying the appropriate design solutions (e.g., the Singleton design pattern). In my thesis, both the design issues and design solutions are generic; meaning that – like design patterns, they can be applied in many situations in any given system and also in different systems. The same design issue may occur at different parts of the system. Each occurrence of design issue is unique and is solved by considering the context of the part of the system in which it occurs. The same design solution may also be instantiated a few times to solve design issues that occur at different parts of a system. A design decision is however not generic, it is taken for an occurrence of design issue by instantiating a design solution and customizing it to suit the context of that part of the system; the effect of the design decision is the impact on the design of the system. For a given occurrence of design issue, one or more alternative design solutions may be considered; they correspond to one or more candidate design decisions. As a result, for a given occurrence of design issue, the designers have to deliberate and select the most suitable one among the multiple candidate design decisions.

The designers typically take a few factors into account. Firstly, the design decisions selected for a system have to collectively satisfy their functional requirements and quality attributes (e.g., runtime memory usage and design-time extensibility), resolving the tensions among them. Secondly, the implications of the selected design decisions may affect each other in complicated ways; the dependencies among them must be accounted. Therefore functional requirements, quality attributes, occurrences of design issues, design solutions, and design decisions form a complicated and ever changing web of information. Understanding this web of design information is essential for making informed design decisions. Unfortunately, design

information rarely is explicitly represented. This creates problems during development, and these problems aggravate in follow up maintenance. The web of design information is even more complex in the Software Product Line (SPL) situation, where by definition, the designers deal with variable requirements that lead to even more variability in the design space.

In my thesis, I formalize the key aspects of the web of design information. My model captures the functional requirements, occurrences of design issues, design solutions, and design decisions along with their implications on design. My model also has provisions for the evolution of its elements where the potential impacts are derived. The benefits of my approach include the explicit documentation of design information, the formal verification of the integrity of design information, the derivation of the applicable code for a consistent set of design decisions, and the derivation of the potential impacts due to the evolution of an element of design information.

Furthermore, my model can be applied to the SPL situation where functional requirements can be variable. According to the feature selection for an SPL application, my model caters to the emergence or the vanishing of the corresponding elements of design information. The additional SPL-specific benefits of my approach include the formal verification of planned feature configurations against those supported by an instance of my model, and the derivation of the applicable code for a consistent set of design decisions for an SPL application.

Although my model does not currently capture the quality attributes and their influence on design decisions, I believe this aspect can be addressed in a future work that extends my work.

I validate my model by illustrating the key usage scenarios. I also devise the schemes to specify and verify my model using formal method. I also evaluate the benefits of my model against the design activities in development processes.

I envision the use of my model as a basis for IDEs that can help developers in documenting the web of design information and validating software design for single systems and SPLs. To guide the tool developers in building such IDEs, I specify the key challenges that need to be addressed as well as possible solutions to these challenges.

List of Figures

Fig. 1. Design Decision Model for a single system.	5
Fig. 2. Design Decision Model for an SPL.	6
Fig. 3. Sample design decisions with trace links from features to code.	15
Fig. 4. Sample design decisions with trace links from features to code (continued).	16
Fig. 5. Metamodel for capturing design decisions and trace links.	17
Fig. 6. Modeling of decisions with their related elements (without trace links).	19
Fig. 7. Sample DDM with trace links from features to code.	20
Fig. 8. Sample DDM with trace links from features to code (the alternative solution for Issue3).	21
Fig. 9. Overview of the elements in the DDM of the complete example (trace links omitted).	22
Fig. 10. Overview of the relationships in the DDM of the complete example (trace links omitted).	25
Fig. 11. Sample compliant chains for applying the implications of decisions.	33
Fig. 12. Sample mappings from decisions to variation points for the evolution of a decision.	35
Fig. 13. Sample ripples for the evolution of a decision.	36
Fig. 14. Sample DDM with trace links from features to code core assets (extended for SPL).	41
Fig. 15. Sample DDM with trace links from features to code core assets (the alternative solution for Issue3) (extended for SPL).	42
Fig. 16. Sample mappings from features to variation points for the evolution of decisions (extended for SPL).	47
Fig. 17. Scheme for verifying the abstract syntax of DDM and its instances.	60

Fig. 18. Scheme for verifying the abstract syntax of FM and its instances.	61
Fig. 19. Scheme for comparing planned against supported feature configurations.	62
Fig. 20. Metamodel for DDM.	66
Fig. 21. Sample mappings for some decisions of the running example.	67
Fig. 22. XVCL as a variability technique.	69
Fig. 23. Metamodel for feature model of FODA.	70
Fig. 24. Rooted directed acyclic graph for the ordering mechanism.	71
Fig. 25. Weighted rooted directed acyclic graph for the prioritization scheme.	71
Fig. 26. Weighted rooted directed acyclic graph for the ripple mechanism.	72
Fig. 27. Key artifacts to be managed by a typical SPL support IDE.	73
Fig. 28. The Analysis and Design workflow of Rational Unified Process.	75
Fig. 29. The Domain Engineering and Application Engineering workflows of the development of SPL.	77

Chapter 1 Introduction

1.1 Motivation

A software system is designed to fulfill both its functional requirements and quality attributes. As the system is designed, the design issues (e.g., existence of duplicate copies of the same object) that occur have to be solved by applying the appropriate design solutions (e.g., the Singleton design pattern). In my thesis, both the design issues and design solutions are generic; meaning that – like design patterns, they can be applied in many situations in any given system and also in different systems. The same design issue may occur at different parts of the system. Each occurrence of design issue is unique and is solved by considering the context of the part of the system in which it occurs. The same design solution may also be instantiated a few times to solve design issues that occur at different parts of a system. A design decision is however not generic, it is taken for an occurrence of design issue by instantiating a design solution and customizing it to suit the context of that part of the system; the effect of the design decision is the impact on the design of the system. For a given occurrence of design issue, one or more alternative design solutions may be considered; they correspond to one or more candidate design decisions. As a result, for a given occurrence of design issue, the designers have to deliberate and select the most suitable one among the multiple candidate design decisions.

Both the functional requirements and the quality attributes (e.g., runtime memory usage and design-time extensibility) are the primary inputs for software design, they collectively determines the selection of an appropriate design decision among the candidate design decisions that are considered for a given occurrence of design issue. Firstly, a design decision may have different impacts on different quality attributes of the system. For instance, the use of the Singleton design pattern to solve an occurrence of design issue may

positively reduce the memory footprint of the system while negatively restricting the extensibility of design (i.e., due to the difficulty in subclassing the class to be instantiated). As a result, a consistent set of design decisions is required to solve the set of occurrences of design issues that occurs during the design of a system. Secondly, the implications of the design decisions in the set are not completely independent; the implication of a design decision may ideally be isolated, however one may exist in the context of the implication of another, one may even be in conflict with the implication of another. As a result, additional occurrences of design issues may arise from these couplings and conflicts, which require even more design decisions to solve them. Last but not least, the eventual set of design decisions selected for a system should also be an optimal set where the quality attributes are concerned. As each candidate design decision contributes in a different way to the quality attributes, the combination of design decisions that satisfy the occurrences of design issues in a system must be selected in such a way that the quality attributes are fulfilled – in fact, it is an elaborate and error-prone effort to exhaustively evaluate all the combinations of these candidate design decisions.

As discussed above, the designers often have to evaluate and decide on the combinations of candidate design decisions to satisfy the above-mentioned tensions among the functional requirements and the quality attributes of a system. The implications of the candidate design decisions on the design of the system may affect each other in some complicated ways. Therefore functional requirements, quality attributes, occurrences of design issues, design solutions, and design decisions form a complicated and ever changing web of information. Understanding this web of design information is essential for making informed design decisions. Unfortunately, design information is rarely explicitly represented. This creates problems during development; and these problems aggravate in follow up maintenance.

The web of design information is even more complex in the SPL situation, where by definition the developers deal with variable requirements that lead to

even more variability in the design space. Firstly, the variability in functional requirements means that the occurrences of design issues (together with their candidate design decisions) that arise due to a variant feature will only apply when the variant feature is selected during application engineering. The emergence or the vanishing of an occurrence of design issue will also impact on the existence of its dependent occurrences of design issues. Secondly, the variability in quality attributes means that the optimal set of design decisions for each feature configuration (of functional requirements) changes as the required quality attributes vary – the derivation of each optimal set will require the elaborate effort as discussed earlier.

In this thesis, my solution deals with aspects common to single systems and SPLs as well as aspects unique to SPLs. I formalize key aspects of the web of design information. My model captures occurrences of design issues and their dependencies, design solutions, design decisions and their dependencies, trace links from features, and trace links to variation points in code. It facilitates designers in evaluating candidate design decisions by recommending valid combinations of candidate design decisions that collectively address the applicable occurrences of design issues. My solution also has provisions for the evolution of its elements. Before an element of my model is evolved, the potential impacts on other elements of the model can be derived for the change to be assessed first. Once the change is effected, the integrity of the resultant model can be checked for noncompliance.

Furthermore, my model can be applied to the SPL situation where features can be variant – either optional or alternative. My solution accounts for the impact of feature selection on the applicability (i.e., emergence or vanishing) of specific occurrences of design issues and their corresponding candidate design decisions in the model. It can recommend the feasible combinations of candidate design decisions for a given feature configuration. It can detect the feature configurations that are planned for but are not supported by a given set of candidate design decisions.

In this thesis, my model does not currently capture the aspect of quality attributes and their influence on the selection of design decisions. This aspect would include the derivation of the optimal sets of design decisions for single systems or SPLs. It can be addressed as part of possible future work that extends my model.

The benefits of my approach include the explicit documentation of design information, the formal verification of the integrity of design information, the derivation of the applicable code for a consistent set of design decisions, and the derivation of the potential impacts due to the evolution of an element of design information. The additional SPL-specific benefits of my approach include the formal verification of planned feature configurations against those supported by an instance of my model, and the derivation of the applicable code for a consistent set of design decisions for an SPL application.

I validate my model by illustrating the key usage scenarios. I also devise the schemes to specify and verify my model using formal method. I also evaluate the benefits of my model against the design activities in development processes.

I envision the use of my model as a basis for IDEs that can help developers document the web of design information and validation of software design for single systems and SPLs. To guide the tool developers in building such IDEs, I specify the key challenges that need to be addressed as well as possible solutions to these challenges.

1.2 Overview of Solution and Contributions

With the above scope in mind, I propose a *Design Decision Model* (DDM) as an intermediate structure between feature tree and code that documents the design information for a single system. A feature tree structures the features of a single system. The code is instrumented to accommodate the impacts of the candidate design decisions of the single system. I generally assume that these code is instrumented with variation points that allow them to be appropriately

configured for reuse (refer to section 8.2.3 for a specific mechanism). For a single system, the code would cater only to variability in design.

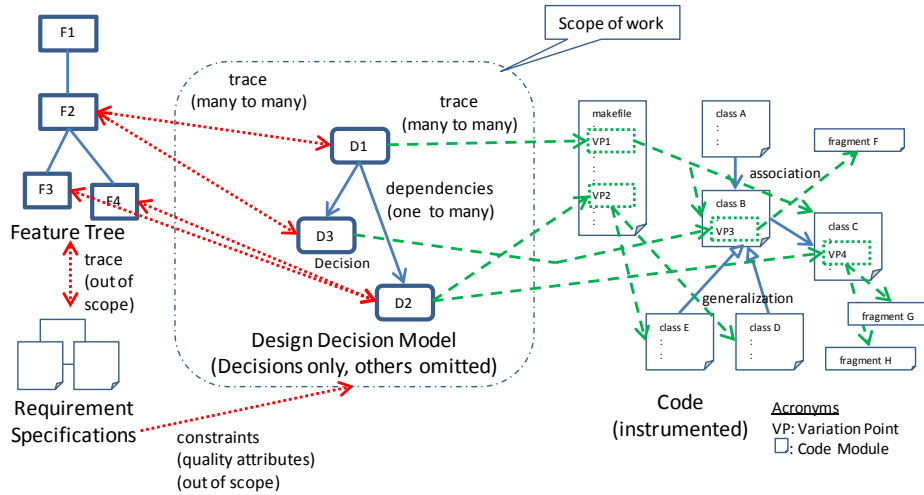


Fig. 1. Design Decision Model for a single system.

Fig. 1 shows DDM in the context of single system design. The model comprises elements (only design decisions are shown, others are omitted for now) of DDM and dependencies among them. The trace links between features and the model associate features with the related design decisions in DDM. The trace links from the model to variation points in code associate the design decisions in DDM with their impacted code. As the requirements of the features evolve, the elements of DDM, trace links, and code must also evolve in tandem. I hence propose a set of traceability rules for enforcing the integrity of DDM.

To apply the above solution to the SPL situation, a feature model is used instead of a feature tree. A feature model describes the variability of features in an SPL. Each SPL application is characterized by a specific selection of features. For an SPL, the code (core assets) would cater to variability in features and design.

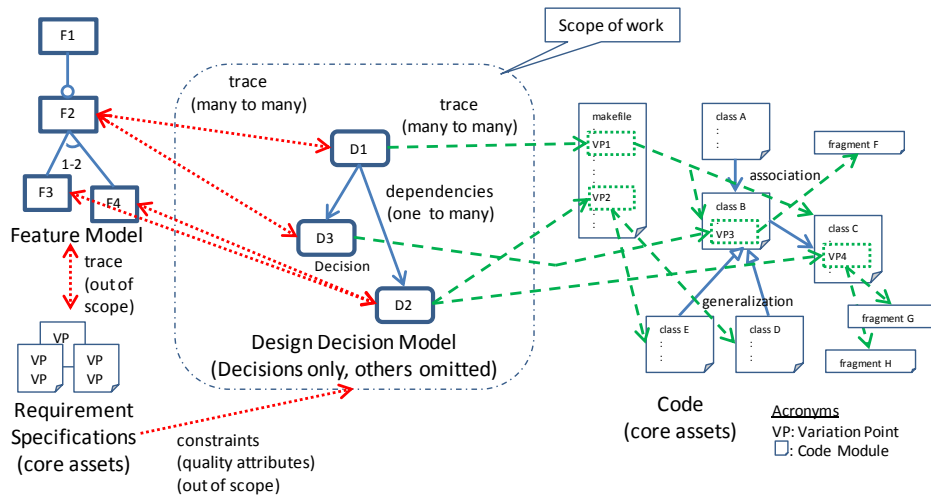


Fig. 2. Design Decision Model for an SPL.

Fig. 2 shows DDM in the context of SPL domain engineering. A feature model is used in place of the feature tree in **Fig. 1**. The features in the feature model can be mandatory or variant (i.e., optional or alternative). Since a variant feature may not be selected for an SPL application, DDM also needs to provide for the emergence or the vanishing of the elements in DDM that correspond to the variant feature.

Because of its impact on productivity, support for traceability between features and code has received much attention in single system and SPL engineering research. However, no comprehensive and practical enough solutions have been proposed, and current solutions provide only limited support for traceability. One reason why traceability solutions have not been more successful is that the problem has not been defined and formalized at sufficient level of details. DDM is proposed as an effective means to support such traceability.

In this thesis, I propose a semi-formal notation for specifying the abstract syntax of DDM and the trace links from features to code via DDM. I propose how formal method can be used to formalize and verify the consistency of the abstract syntax, the consistency of the instances of DDM, and the comparison of *planned* feature configurations against those *supported* by an instance of

DDM (for the SPL situation). I also propose how the formalization can be used in systematically deriving the applicable code for a given feature configuration (for the SPL situation) as well as highlighting the impacts due to the evolution of the elements of DDM. I envision the use of this abstract syntax and its formalization as the basis for IDEs that can help developers in the design of single systems as well as in the domain engineering and the application engineering of SPLs.

A critical advantage of my solution is in allowing the use of the automatic reasoning capability of formal method in the verification of properties of interest and the derivation of information from DDM. As compared to manual inspection, this approach conducts systematic analyses that are much more exhaustive, reliable, and quick. This minimizes the required human effort and potential oversights.

1.3 Organization of Thesis

In this thesis, Chapter 2 describes the problem. Chapter 3 and Chapter 4 formalize DDM and impacts of design decisions respectively. Chapter 5 extends the formalization for the SPL situation. Chapter 6 validates the usage of DDM and its impacts by means of usage examples. Chapter 7 describes how formal method can be used to specify and verify the abstract syntax of DDM, instances of DDM, and feature configurations of instances, and to derive information from instances of DDM. Chapter 8 suggests how the key salient features of IDEs adopting DDM can be implemented. Chapter 9 evaluates the benefits of DDM against the design activities of single systems and SPLs. Chapter 10 discusses related works. Chapter 11 concludes by summarizing the achievements and recommending future works.

Chapter 2 Problem

This chapter describes the problem, explains its relevance in the design of single systems and SPLs, and also motivates it with a running example.

2.1 Problem Definition

In the software design of single systems and SPLs, the designers may consider some alternative design solutions for each design issue that occurs at a part of the system without explicitly documenting the corresponding candidate design decisions. The core of my problem focuses on the explicit documentation of these candidate design decisions and their implications on the design of the system, and the benefits that can be derived to help developers in the design of single systems and SPLs.

Assuming object-oriented design, the structure of code is specified by the design elements (i.e., classes and interfaces) and their relationships (i.e., association, dependency, generalization, and realization); while the behaviour is specified by the design objects and their interactions. A design issue may occur in the structural and/or behavioural design of one or more features (i.e., a part of the system). The design issue may be solved by one or more alternative design solutions. A design solution is generic – not specific to the context of any part of the system, it may be instantiated a few times to solve multiple design issues that occur at different parts of the system. When a candidate design decision is taken for an occurrence of design issue, an alternative design solution is instantiated to the context of that part of the system. The implication of a candidate design decision is on the structure and/or the behaviour of the code. For each occurrence of design issue, the designers evaluate the candidate design decisions and select the most appropriate one. As the implication of a design decision may give rise to a new design issue or may even be in conflict with the implication of another design decision; this results in dependencies among the design decisions.

These dependencies must also be documented so that they can be taken into account when the candidate design decisions are evaluated by the designers. I refer to a model that captures these occurrences of design issues, the design solutions, and the corresponding design decisions as *Design Decision Model* (DDM).

In the domain engineering of a SPL, the domain engineers design code core assets to realize the variability in features, aiming for optimized reuse during application engineering. To support the variability in features, DDM needs to be flexible in terms of the emergence or the vanishing of the elements of DDM that are associated with each variant feature. A variant feature can be associated with zero or more occurrences of design issues, each of which is in turn associated with one or more candidate design decisions.

Fig. 1 of section 1.2 is a simplified illustration of selected design decisions without showing occurrences of design issues, design solutions, and other candidate design decisions (these will be detailed in Chapter 3). There are three design decisions as in D1, D2, and D3. D1 handles a design issue that occurs in the design of feature F2. D2 handles a design issue that occurs in the design common to features F3 and F4. D3 handles a design issue that occurs in the design of feature F2 that arises due to D1. In general, the relationship between features and design decisions, via occurrences of design issues, is many-to-many. One or more occurrences of design issues that arise from one or more features may be addressed by one or more design decisions; while a design decision may address an occurrence of design issue that arises from one or more features. I generally assume the variability technique in code to comprise variation points that control the reuse of code. A design decision affects its implication on the design by configuring one or more applicable variation points; while a variation point may be impacted by multiple design decisions. In **Fig. 1**, D1 impacts on variation point VP1 that reuses classes B and C. D2 impacts on VP2 and VP4 where VP2 reuses classes D and E while VP4 reuses fragments G and H. D3 impacts on VP3 which reuses fragment F.

In addition, there are also dependencies among the design decisions as one may be taken on the premise of the others and one may be in conflict with another – I analyze them further in section 3.2. **First subproblem:** The abstract syntax of DDM and trace links from features through to variation points should be specified and verified for *consistency*. (SPL-specific) The abstract syntax also has to provide for the emergence and the vanishing of the elements of DDM for each variant feature. **Second subproblem:** Instances of DDM should also be verified to be *consistent* with the abstract syntax.

Fig. 2 of section 1.2 extends **Fig. 1** for the SPL situation. A feature model is used to describe the variability in features. It specifies the composition and dependencies among the features of an SPL. It implies a set of feature configurations which are *planned* by the domain engineers. On the other hand, an instance of DDM represents the actual design for the features. It implies a set of feature configurations which are *supported* within the constraints of the instance of DDM. Since the design is often compromised due to the realities in implementation technologies or human oversights, it is highly likely for some planned feature configurations to be unsupported for a given instance of DDM. **Third subproblem (SPL-specific):** In order to establish the *correctness* of the design for an SPL, the set of planned feature configurations must be exhaustively derived and verified against those supported by the instance of DDM – this is a laborious and error-prone task. A mismatch can be addressed by the domain engineers by either constraining the set of planned feature configurations in the feature model or expanding the set of supported feature configurations in the instance of DDM.

Having verified the feature configurations of a feature model, each feature configuration represents a supported application of the SPL. For a given feature configuration, the applicable code for the application are derived from the core assets. **Fourth subproblem (SPL-specific):** For a given feature configuration, the possible combinations of design decisions, the impacted

variation points and their configurations, and the preferred order of applying these design decisions are systematically derived from an instance of DDM.

The design for a single system or an SPL is evolved in response to changes in its required features, the adopted implementation technologies, etc. An instance of DDM guides the developers by deriving the potential impacts of a change. After the change is effected, the developers update the instance of DDM to reflect the evolved design. **Fifth subproblem:** For a change in the design, the potential impact of the change is systematically derived from the instance of DDM. (SPL-specific) The derivation also has to provide for the removal of a variant feature. As for the resultant instance of DDM, it has to be verified to be consistent with the abstract syntax – this is subsumed as part of the second subproblem.

In summary, the problem can be broken down to the following five sub-problems:

1. Specification and verification of the abstract syntax of DDM and trace links from features to variation points. (SPL-specific) The abstract syntax also has to provide for the emergence and the vanishing of the elements of DDM for each variant feature.
2. Specification and verification of the instances of DDM against the abstract syntax.
3. (SPL-specific) Derivation and verification of planned feature configurations against those supported by an instance of DDM.
4. (SPL-specific) Derivation of the possible combinations of design decisions, the impacted variation points and their configurations, and the preferred order of applying these design decisions for a given feature configuration of an instance of DDM.
5. Derivation of the potential impact of a change in the design of an instance of DDM. (SPL-specific) The derivation also has to provide for the removal of a variant feature.

As a guide to locate the solution to the above subproblems, the following indices to the key sections are provided against each subproblem:

1. Chapter 3, section 5.2, section 7.4, section 8.2.1, and section 8.2.2.
2. Chapter 3, section 5.2, section 7.4, section 8.2.1, and section 8.2.2.
3. Section 7.6.
4. Section 7.7 and section 8.2.5.
5. Chapter 4, section 5.3, and section 8.2.6.

2.2 Running Example

This section introduces an example which is a part of a Car Rental System. It is referred by the later sections. It contains feature tree, DDM, and code. As in **Fig. 1** of section 1.2, only some selected design decisions of DDM are illustrated in the earlier part of this section. Other elements of DDM are detailed in the later part of this section and Chapter 3.

Fig. 3 and **Fig. 4** illustrate four design decisions D1 through D4, the associated features F5 through F10, and the impacted variation points VP1 through VP4 in code. In each of the two figures, on the left is a fragment of feature tree; on the right is the code that realizes the design of the features; in the middle are the design decisions and the trace links from features to code.

Using trace links, D1 and D2 are associated with F5 while D4 is associated with F6 through F10. D3 is not directly associated with any features as it resolves a conflict that arises between D1 and D2. Trace links are also used to associate D1 through D4 with their impacted variation points that include/exclude code. D1 impacts VP1; D2 impacts VP3; D3 impacts VP4; and D4 impacts VP2. I assume that these variation points are instrumented using a variability technique that can include/exclude and configure code. With the above, it is possible to trace end-to-end from a feature to its associated design decisions and further to the impacted variation points.

The design decisions are not isolated; there are inherent dependencies among them which are explained as they are specified in section 3.2. In the two figures, I illustrate that D1 “constrains” D2 and “comprises” D4; D2 “forbids” (i.e., conflicts with) D2; and D3 “resolves” the conflict between D2 and D1.

Apart from the design decisions, there are also other elements that are essential in decision making. In order to specify these additional details, I refer to the related works by Kruchten et al. [11] and Capilla et al [4]. [11] analyzes architectural design decisions and focuses on managing design knowledge in terms of such decisions. It suggests the possible attributes of a decision as

description, rationale, scope (system, time, and organization), author (time-stamp and history), state, categories (usability, security, etc.), etc. It also suggests the possible relationships between these decisions as constrains, forbids, enables, subsumes, conflicts with, overrides, comprises, is an alternative to, is bound to, is related to, dependencies, etc. [4] proposes a reference metamodel to model architectural design decisions. I adapt and extend both the existing works below.

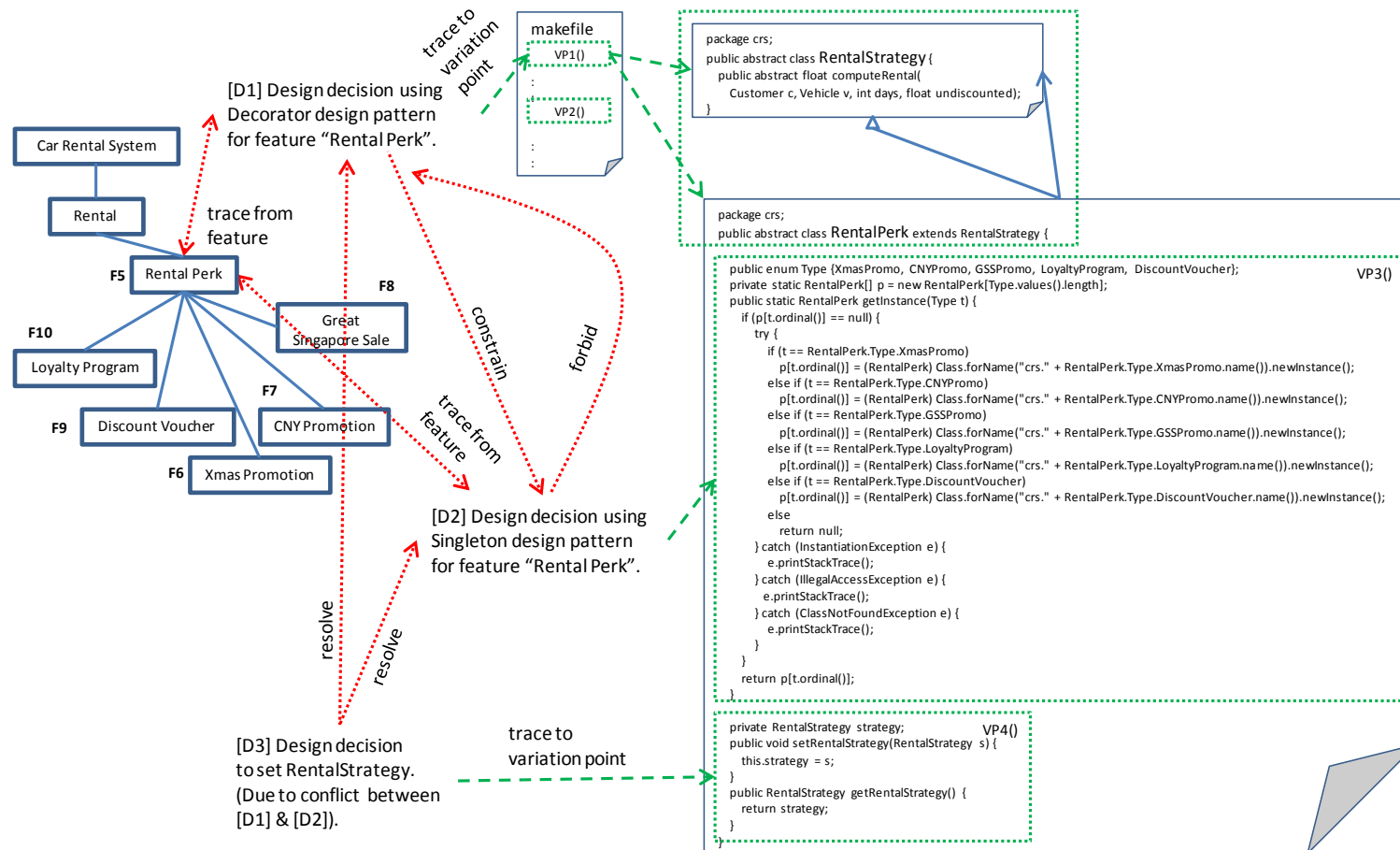


Fig. 3. Sample design decisions with trace links from features to code.

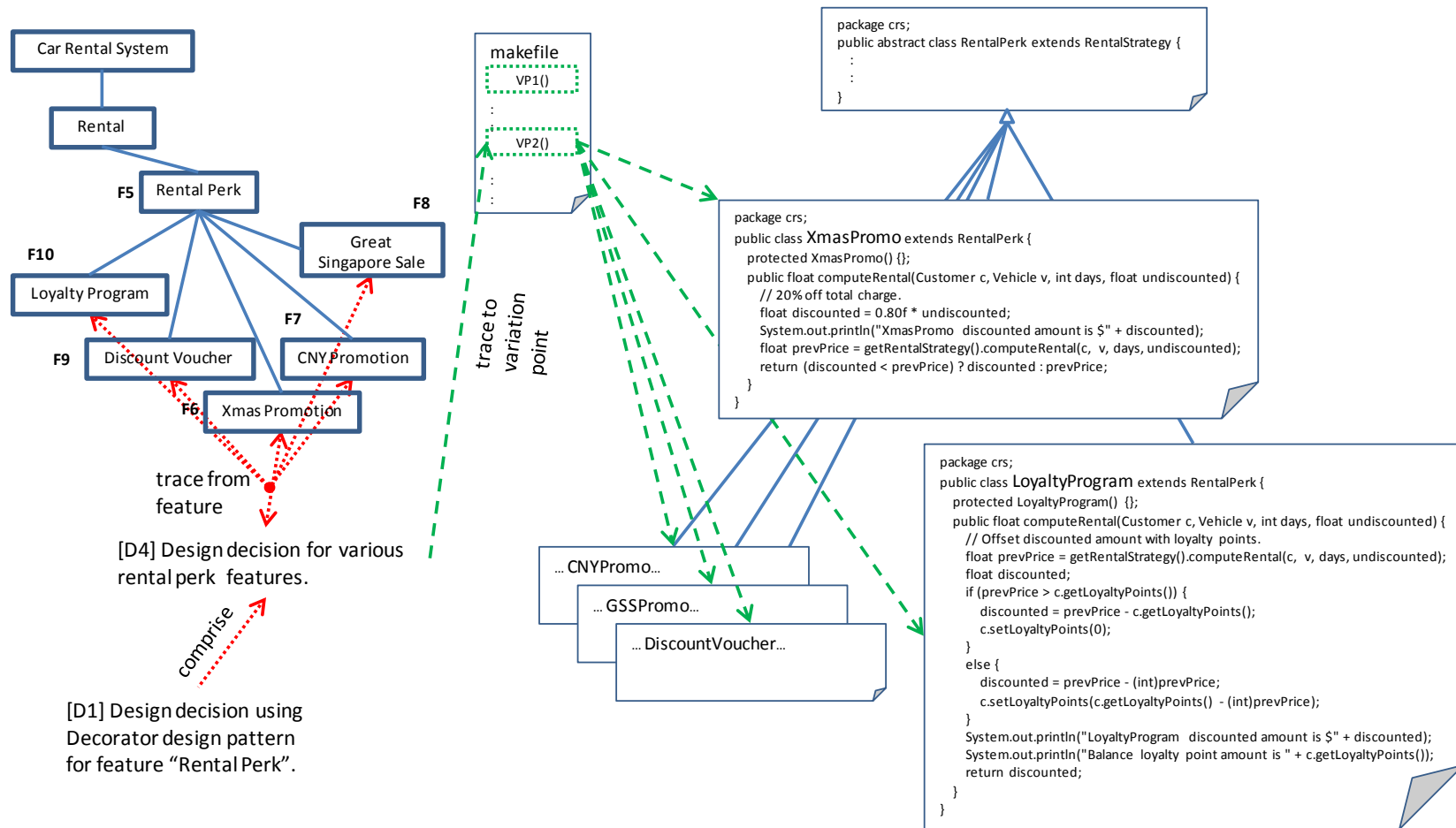


Fig. 4. Sample design decisions with trace links from features to code (continued).

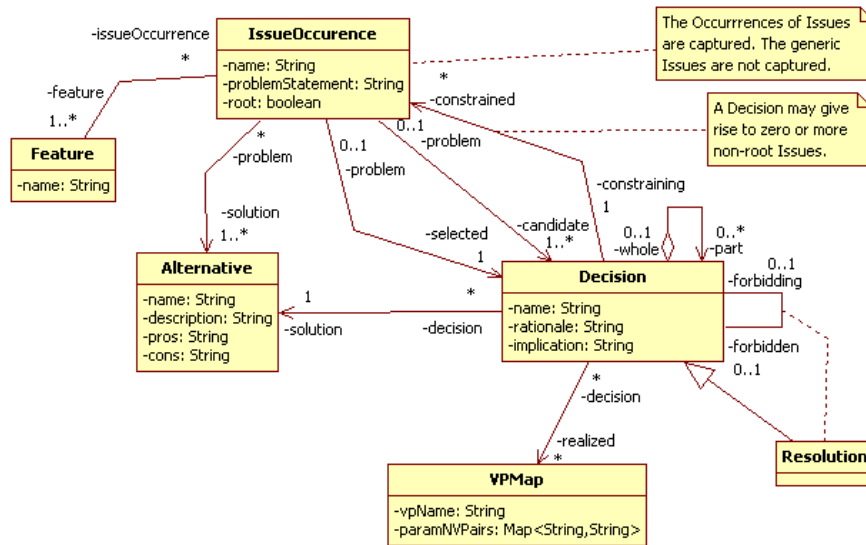


Fig. 5. Metamodel for capturing design decisions and trace links.

Fig. 5 is a UML class diagram that illustrates my metamodel, as adapted and enhanced from [11] and [4], for capturing design decisions in single system design. The key elements of the metamodel are *issue occurrence*, *alternative*, and *decision* (termed as *outcome* in [4]). An *issue occurrence* is an instance of design issue that arises in the context of the design for one or more features. (Note that the generic design issues are omitted from the metamodel as they add little information while the essential information is already captured by the issue occurrences.) The issue occurrence may possibly be addressed by one or more *alternative* solutions. A *decision* is taken to instantiate an alternative solution to the context of the issue occurrence. For a given issue occurrence, there are as many candidate decisions as the number of alternative solutions considered – Each candidate decision impacts the code differently. Each issue occurrence is solved by selecting one decision among the candidate decisions of the issue occurrence.

Fig. 6 illustrates the modeling of D1 and D2 with their related elements. D1 is modeled as Decision6, Issue3, Alternative4, and Alternative6. (Refer to Fig. 8 for the candidate decision for Alternative6.) D2 is modeled as Decision8,

Issue4, and Alternative5. Decision6 constrains Decision8 via Issue4. Decision8 forbids Decision6. (Note that Issue3 and Issue4 are actually issue occurrences.)

Fig. 7 shows a sample DDM with features and variation points that covers decisions D1 through D4. D3 is modeled as Decision9 (*Name: Resolve conflict between Outcome8 and Outcome6. Rationale: Rental perk child classes have public constructors while Singleton constructors should be protected or private. Cannot initialize a RentalPerk instance with a RentalComp instance via constructor. Implication: Make constructors of rental perk child classes protected. Add setRentalStrategy() to initialize a RentalPerk instance with a RentalStrategy instance.*) Decision9 resolves the conflict between Decision8 and Decision6. D4 is modeled as Decision7 (*Name: Extensibility of rental perks. Rationale: Decouple other classes from rental perk child classes. Implication: Add, modify or remove rental perk child classes to/from rental perk hierarchy.*) Decision6 comprises Decision7 (i.e., Decision7 is a part of Decision6).

Fig. 8 extends **Fig. 7** to show the candidate decision for alternative solution for Issue3. Decision10 (*Name: Represent combinations of rental perks using subclasses. Rationale: Create a hierarchy of subclasses to represent required combinations. Acceptable for small number of combinations. Implication: Use one subclass for each combination of rental perks.*) solves Issue3 using Alternative6. Decision10 gives rise to and constrains Issue5 (*Name: Too many instances of rental perk combinations. Problem statement: Each rental scheme is configured with its own instances of rental perk combination.*) Decision12 (*Name: Share instances of rental perk combinations. Rationale: Rental perk combinations are not specific to any rental scheme. Implication: Apply Singleton pattern to RentalPerkComb. Add getInstance() that instantiates and shares instances of child classes.*) solves Issue5 by using Alternative5 (*Name: Singleton Design Pattern. Description: Ensure a class only has one instance, and provide a global point of access to it. Pros:*

Controlled access to sole instance. Can vary number of instances. **Cons:** Direct instantiation is not allowed.) Decision10 also comprises Decision11 (**Name:** Extensibility of rental perk combinations. **Rationale:** Decouple other classes from rental perk combination child classes. **Implication:** Add, modify or remove rental perk combination child classes to/from rental perk combination hierarchy.)

Lastly, **Fig. 9** shows all the elements of the sample DDM for the running example.

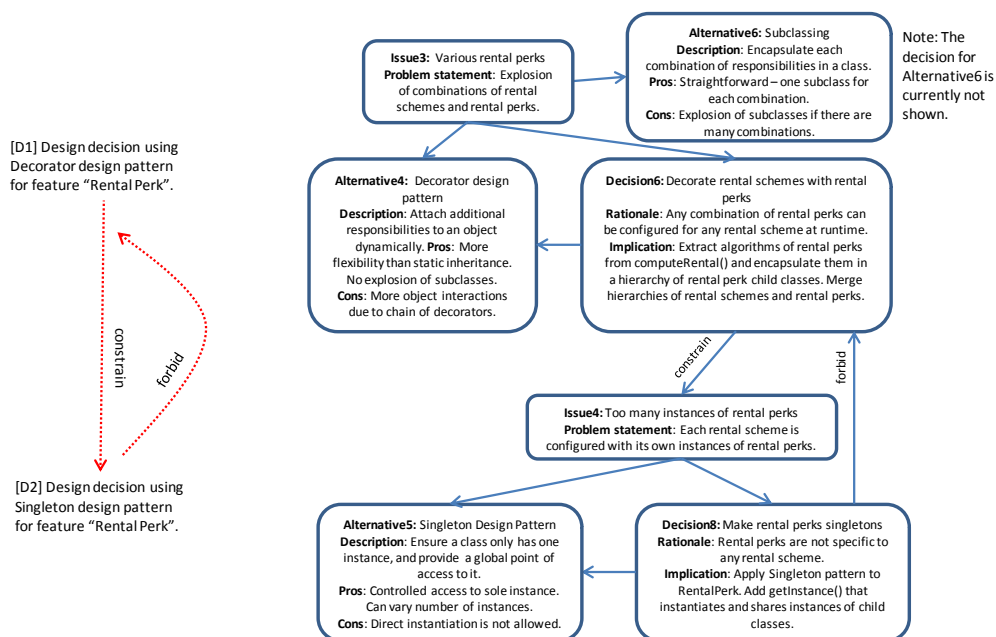
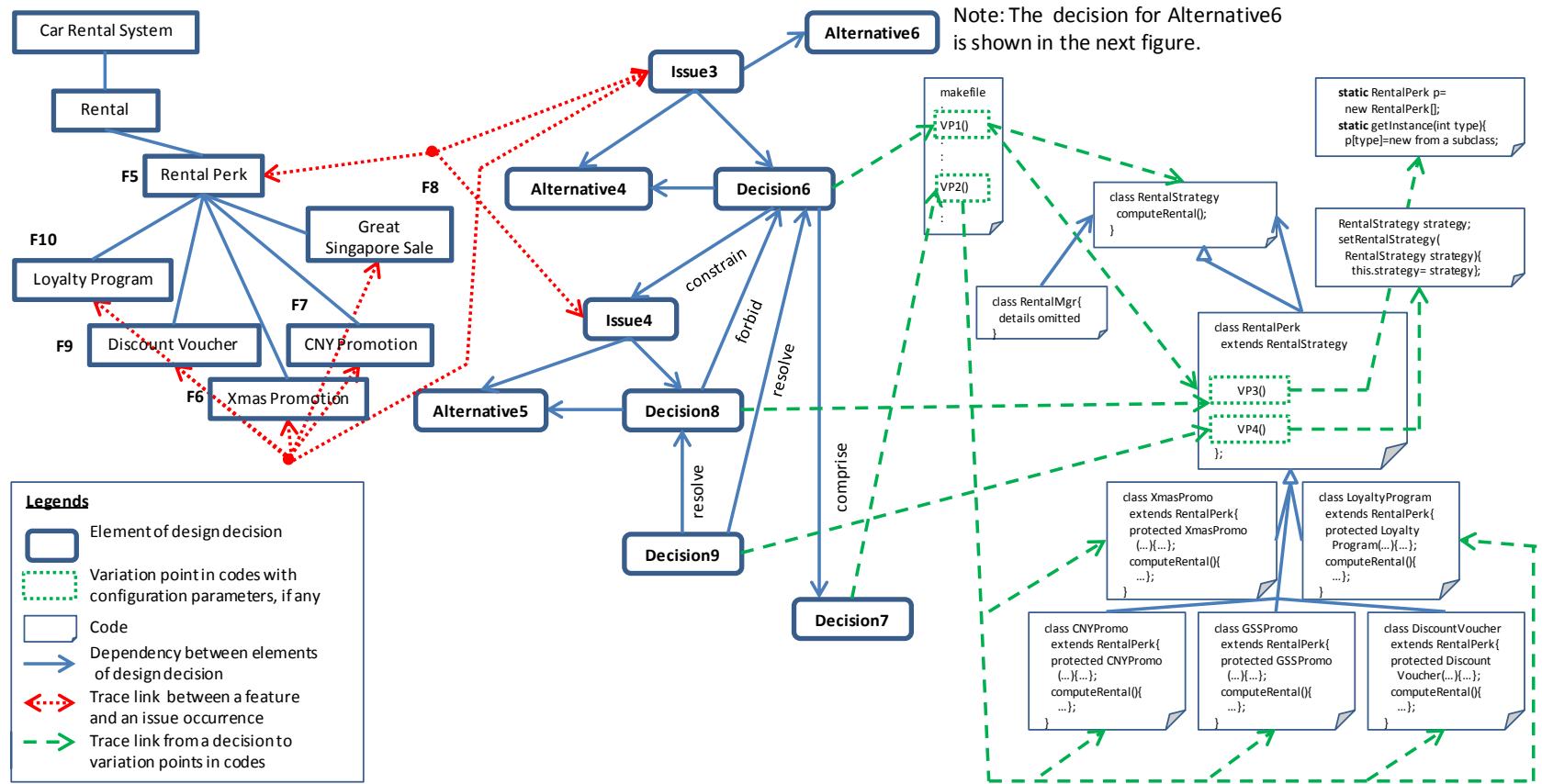


Fig. 6. Modeling of decisions with their related elements (without trace links).



Note: The decision for Alternative6 is shown in the next figure.

Fig. 7. Sample DDM with trace links from features to code.

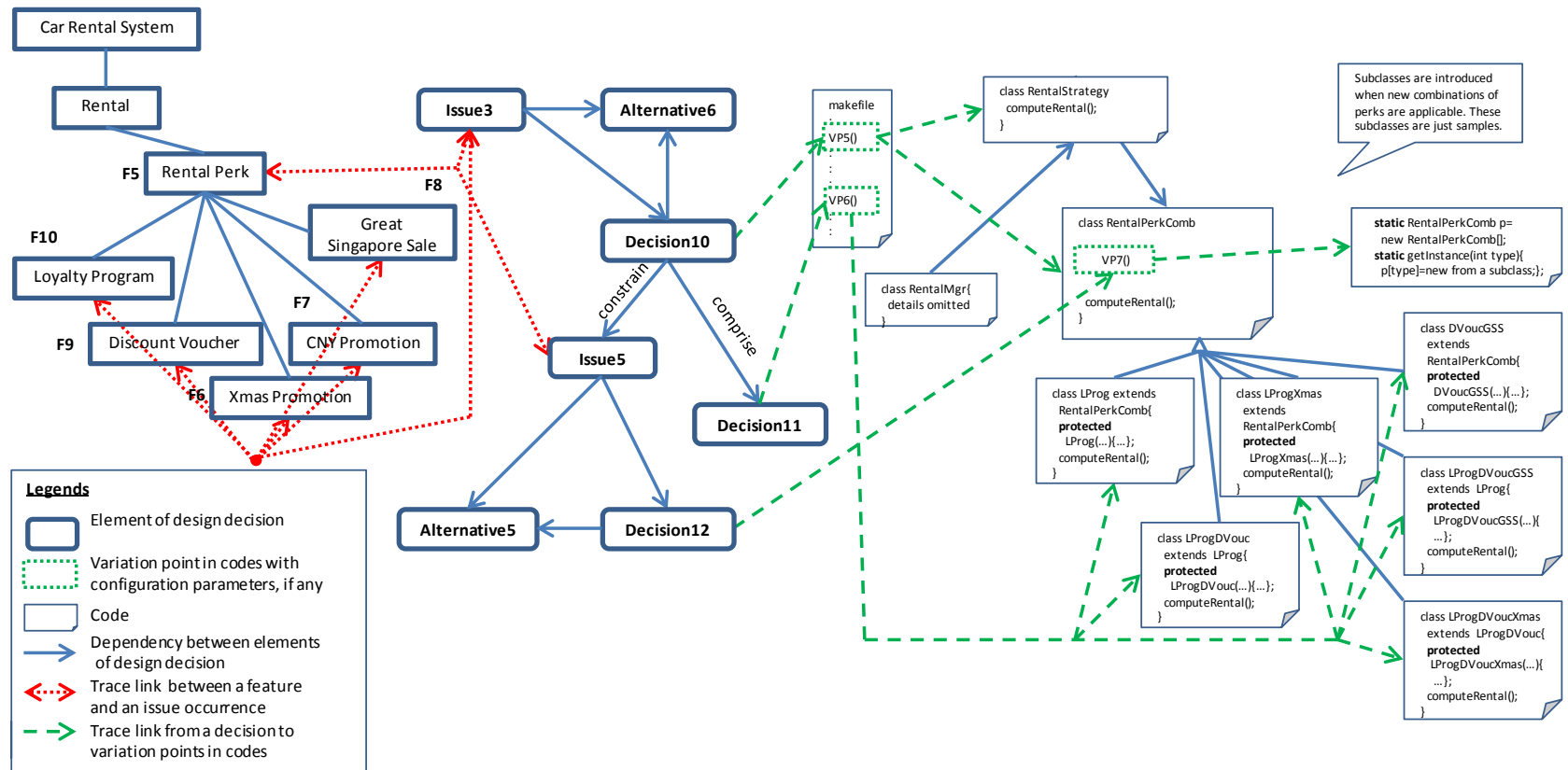


Fig. 8. Sample DDM with trace links from features to code (the alternative solution for Issue3).

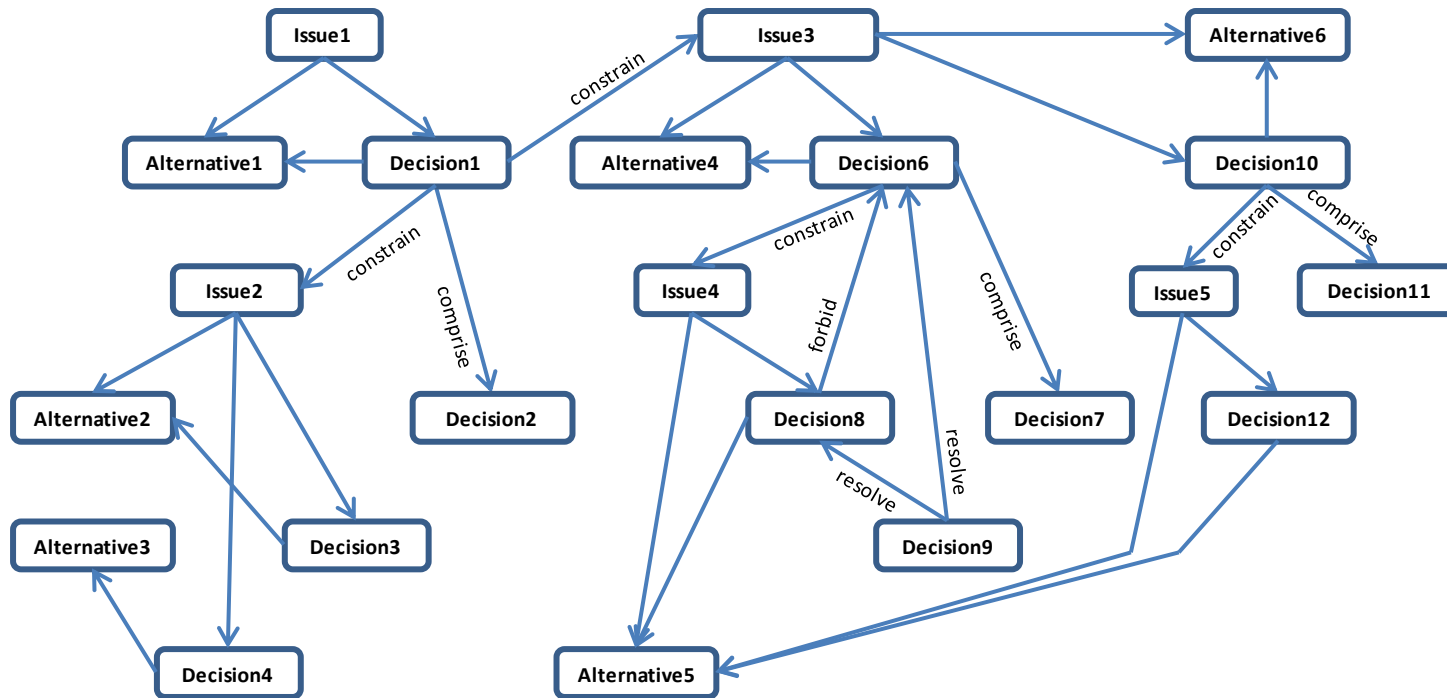


Fig. 9. Overview of the elements in the DDM of the complete example (trace links omitted).

Chapter 3 Formalization of Abstract Syntax of DDM for Single Systems

This chapter analyzes the running example in section 2.2 and formalizes the elements of DDM, the dependencies between them, and the trace links from features to the impacted variation points in code. A set of traceability rules are specified to enforce the integrity of DDM. The additional challenges to be addressed by the developers of IDEs that adopt my model are highlighted.

3.1 Elements of DDM

Based on the metamodel in **Fig. 5**, the key elements are issue occurrence, alternative, and decision. As explained in section 2.1, they capture the occurrences of design issues, the alternative design solutions considered, and the candidate design decisions along with their impacts on code. Without formally capturing this information, the traceability from features to code is incomplete; the design decisions behind the implementation cannot be explicitly reasoned and evolved.

An **issue occurrence** is formalized as a 2-tuple, $i = (n, ps)$ where

n = name

ps = problem statement

An **alternative** is formalized as a 4-tuple, $a = (n, as, pr, cn)$ where

n = name

as = alternative solution

pr = pros of alternative

cn = cons of alternative

A **decision** is formalized as a 4-tuple, $d = (n, rt, ip, vps)$ where

n = name

rt = rationale behind the decision

ip = implication of the decision

$vps = \{vp_1, vp_2 \dots, vp_n\}$ where $n \geq 1$ is a set of *impacted* variation points in code

A **variation point** is formalized as a 2-tuple, $vp = (n, ps)$ where

$n = \text{name}$

$ps = (p_1, p_2 \dots, p_m)$ where $m \geq 0$ is a sequence of *input* parameters that configures the variation point on specific ways in reusing code

The above scheme is used to formalize issue occurrences, alternatives, decisions, and impacted variation points in code as shown in **Fig. 7** and **Fig. 8**. A few examples needed in this section are given for various element types. Refer to Appendix A for the complete formalization.

Issue occurrences, $I = \{i_3, i_4, i_5\}$

Alternatives, $A = \{a_4, a_5, a_6\}$

Decisions, $D = \{d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}\}$

Variation Points, $VP = \{vp_1, vp_2, vp_3, vp_4, vp_5, vp_6, vp_7\}$

$i_3 = (\text{"Various rental perks"}, \text{"Explosion of combinations of rental schemes and rental perks"})$

$a_4 = (\text{"Decorator design pattern"}, \text{"Attach additional responsibilities to an object dynamically."}, \text{"More flexibility than static inheritance. No explosion of subclasses."}, \text{"More object interactions due to chain of decorators."})$

$d_6 = (\text{"Decorate rental schemes with rental perks"}, \text{omitted}, \text{omitted}, \{vp_1\})$

$d_7 = (\text{"Extensibility of rental perks"}, \text{omitted}, \text{omitted}, \{vp_2\})$

$vp_1 = (\text{"VP1"}, ())$

$vp_2 = (\text{"VP2"}, ())$

$vp_3 = (\text{"VP3"}, ())$

$vp_4 = (\text{"VP4"}, ())$

For a given decision (e.g. d_7), the impact on the variation points (vps) needs to be captured. A mechanism is required to *map from the decision to the applicable variation points and the specific parameters, if any, of each variation point*. Tool developers need to address this mapping mechanism in

tool implementation. The following sample mappings are provided for the impact of d_6 , d_7 , d_8 , and d_9 :

d_6 maps to vp_1 ;

d_7 maps to vp_2 ;

d_8 maps to vp_3 ;

d_9 maps to vp_4 .

3.2 Dependencies between Elements of DDM

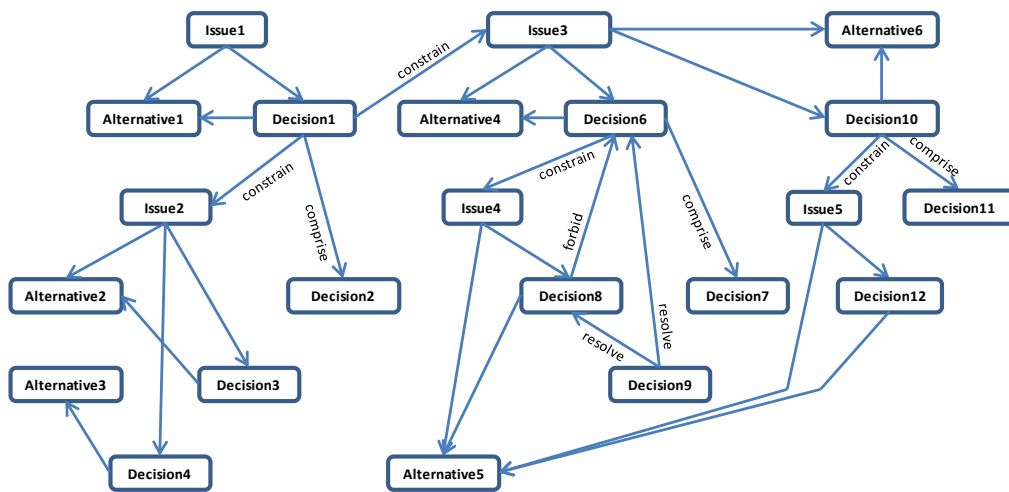


Fig. 10. Overview of the relationships in the DDM of the complete example (trace links omitted).

Fig. 7 and Fig. 8 also shows additional dependencies required in the running example beyond my reference metamodel in [4]. Fig. 10 (reproduced from Fig. 9 for ease of reference) shows the dependencies that exist for the complete running example. The various types of dependency collectively embody the rules that define the integrity of DDM. The following subsections analyze the types of dependency and formalize these rules with samples given for the running example. I refer to these rules as *traceability rules* that must be enforced for the integrity of DDM.

3.2.1 Issue occurrence-alternative Association

The relationship captures the *alternative solutions* considered for the issue occurrences. A solution is generic – not specific to the context of any issue occurrence. An issue occurrence may be solved by one or more alternative solutions. Different solutions address the issue occurrence in different ways; a suitable way is to be selected for the issue occurrence.

An issue occurrence-alternative association is formalized as a 2-tuple,

$ia = (i, a)$.

E.g. Issue occurrence-alternative associations,

$IA = \{(i_3, a_4), (i_3, a_6), (i_4, a_5), (i_5, a_5)\}$

Traceability Rule 1: Co-existence of issue occurrences and alternatives in issue occurrence-alternative association.

$i_j \in I \Rightarrow \exists a: (i_j, a) \in IA$

3.2.2 Issue occurrence-decision Association

The relationship captures the *binding of a candidate decision* to an issue occurrence. As an alternative solution associated with a decision is a generic solution that can possibly address multiple issue occurrences, this issue occurrence-decision association binds a candidate decision to a specific issue occurrence. This association captures the instantiation of an alternative solution to the context of an issue occurrence. Furthermore, for a given issue occurrence, one of the candidate decisions must be selected.

An issue occurrence-decision association is formalized as a 2-tuple,

$id = (i, d)$.

The selection of a candidate decision is formalized as predicate *selected*.

E.g. Issue occurrence-decision associations,

$ID = \{(i_3, d_6), (i_3, d_{10}), (i_4, d_8), (i_5, d_{12})\};$
 $selected(d_6); selected(d_8); selected(d_{12}).$

Traceability Rule 2: Co-existence of issue occurrences and candidate decisions in issue occurrence-decision association.

$$\mathbf{i_j \in I \Rightarrow \exists d \in D: (i_j, d) \in ID}$$

$$\mathbf{i_j \in I \Rightarrow \exists! d \in D: ((i_j, d) \in ID \text{ AND } \text{selected}(d))}$$

3.2.3 Decision-alternative Association

The relationship captures the *contextualization of generic solutions* for an issue occurrence that arises due to one or more specific features. A decision justifies, with rationale, the choice of an alternative solution. As an alternative solution is generic (e.g. design pattern), it has to be contextualized for the issue occurrence. Hence, a decision also captures the specific way the alternative solution is applied, by identifying the impacted variation points accordingly.

A decision-alternative association is formalized as a 2-tuple,

$$da = (d, a).$$

E.g. Decision-alternative associations,

$$DA = \{(d_6, a_4), (d_8, a_5), (d_{10}, a_6), (d_{12}, a_5)\}$$

Traceability Rule 3: Co-existence of decisions and alternatives in decision-alternative association.

$$\mathbf{(i_j, d_k) \in ID \Rightarrow \exists! a: ((i_j, a) \in IA \text{ AND } (d_k, a) \in DA)}$$

3.2.4 Comprise Association

The relationship captures the *compositions among decisions*. A decision may “comprise” other decisions. The “comprise” association represents that one decision is made of one or more decisions. The “whole” decision should also precede its “part” decisions when applied. d_1 “comprise” d_2 and d_3 implies that d_1 is made of d_2 and d_3 ; d_1, d_2 and d_3 can be seen collectively as a single composite decision that should be taken or dropped together. d_1 should also

precede d_2 and d_3 when applied, while ordering between d_2 and d_3 does not matter.

A comprise association is formalized as a 2-tuple,

$dd_{comprise} = (d_j, d_k)$ where $d_j \neq d_k$.

E.g. Comprise associations,

$DD_{comprise} = \{(d_6, d_7), (d_{10}, d_{11})\}$

Traceability Rule 4: Co-existence and precedence of whole and part decisions in comprise association.

$(d_j, d_k) \in DD_{comprise} \Rightarrow (d_j \Leftrightarrow d_k) \text{ AND } \text{precede}(d_j, d_k)$

where predicate **precede**(d_j, d_k) means implication of d_j precedes that of d_k .

Traceability Rule 5: Transitivity in comprise associations.

$(d_j, d_k) \in DD_{comprise} \text{ AND } (d_k, d_l) \in DD_{comprise} \Rightarrow (d_j, d_l) \in DD_{comprise}$.

3.2.5 Constrain Association

The relationship captures the *constraints between issue occurrences and decisions*. A decision may give rise to other issue occurrences; these issue occurrences arise in the context of the decision. Hence, the decision “constrains” the issue occurrences and their associated decisions. The “constrain” association represents that one or more issue occurrences arise in the premise of a decision. d_1 “constrains” i_2 implies that i_2 arises in the premise of d_1 ; if d_1 is dropped, then i_2 becomes irrelevant.

A constrain association is formalized as a 2-tuple,

$di_{constrain} = (d, i)$.

E.g. Constrain associations,

$DI_{constrain} = \{(d_6, i_4), (d_{10}, i_5)\}$

Traceability Rule 6: Co-existence and precedence of decisions and the issue occurrences they raise in constrain association.

$$(d, i) \in DI \Rightarrow (d \Rightarrow i) \text{ AND precede}(d, i).$$

Traceability Rule 7: Transitivity of constrain and issue occurrence-decision associations.

$$(d_j, i_k) \in DI_{\text{constrain}} \text{ AND } (i_k, d_l) \in DO \Rightarrow (d_j, d_l) \in DD_{\text{constrain}}.$$

where $DD_{\text{constrain}}$ is the set of derived decision-decision “constrain” associations.

$$(d_j, d_l) \in DD_{\text{constrain}} \Rightarrow (d_j \Rightarrow d_l) \text{ AND precede}(d_j, d_l).$$

3.2.6 Forbid and Resolve Associations

The relationships capture the *conflicts between decisions and their resolutions*. A *conflict* between two decisions occurs if their implications cannot be applied concurrently in harmony. It must be resolved by compromising either or both of the implications of the conflicting decisions. Such compromise in implications is called *resolution*; it makes it possible for both decisions to be applied concurrently.

The “forbid” association represents the prevention by another decision of a decision from being applied. Decision d_2 “forbids” decision d_1 implies that the implication of d_2 conflicts with that of d_1 ; d_2 is not possible unless the implications of d_1 and/or d_2 are worked around by the resolution (also a decision) d_3 . The “resolve” association represents the resolution of a “forbid” conflict. d_3 “resolves” conflict of d_2 “forbids” d_1 implies that d_3 makes it possible for both d_1 and d_2 to co-exist.

The forbid and resolve associations are formalized as 2-tuples and should exist in triplets as follow.

$$dd_{\text{forbid}} = (d_k, d_j) \text{ where } d_j \neq d_k$$

$$dd_{\text{resolve1}} = (d_r, d_j) \text{ where } d_r \neq d_j; dd_{\text{resolve2}} = (d_r, d_k) \text{ where } d_r \neq d_k$$

E.g.

Forbid associations, $DD_{\text{forbid}} = \{(d_8, d_6)\}$,
 Resolve associations, $DD_{\text{resolve}} = \{(d_9, d_8), (d_9, d_6)\}$

Traceability Rule 8: Co-existence of decisions in forbid and resolve associations.

$$(d_k, d_j) \in DD_{\text{forbid}} \Rightarrow \exists d_r: \{(d_r, d_j), (d_r, d_k)\} \subseteq DD_{\text{resolve}}$$

Traceability Rule 9: Precedence of decisions in forbid associations.

$$(d_k, d_j) \in DD_{\text{forbid}} \Rightarrow \text{precede}(d_j, d_k).$$

Traceability Rule 10: Precedence of decisions in resolve associations.

$$(d_r, d_j) \in DD_{\text{resolve}} \text{ AND } (d_r, d_k) \in DD_{\text{resolve}} \Rightarrow \\ \text{precede}(d_j, d_r) \text{ AND } \text{precede}(d_k, d_r).$$

3.3 Trace Links

The running example shows the trace links between features, the associated decisions (actually via issues) in DDM, and the impacted variation points in code. The trace links are captured to support the traceability of features and variability in design.

3.3.1 Feature-issue occurrence Trace

The relationship traces between features and DDM as part of end-to-end traceability from features to variation points in code.

A decision may be taken directly for one or more features. A decision may also be taken indirectly via comprise, constrain, forbid, and resolve associations. Decisions that are neither directly nor indirectly taken for some features are still included for tracing as they represent design variability.

A feature may be associated with one or more issue occurrences while an issue occurrence may be associated with zero or more features.

A feature is formalized as a 1-tuple, $f = (n)$ where $n = \text{name}$.

A feature-issue occurrence trace is formalized as a 2-tuple, $fi = (f, i)$. It is bidirectional. A set of feature-issue occurrence traces is a symmetric relation.

As illustrated in **Fig. 7** and **Fig. 8**,

```
Features F = {f5, f6, f7, f8, f9, f10}; f5 = ("Rental Perk")
(other features omitted)

Feature-issue occurrence traces FI = {(f5, i3), (f5, i4), (f5, i5),
(f6, i3), (f7, i3), (f8, i3), (f9, i3), (f10, i3)}
```

3.3.2 Decision-code Trace

The relationship traces between DDM and the variation points in code as part of the end-to-end traceability from features to code. The code is in the form of reusable code fragments which can be class, interface, attribute, operation, statement, or a part of statement.

In **Fig. 7** of section 2.2, RentalPerk is a class, RentalPerk.strategy is an attribute, RentalPerk.setRentalStrategy is an operation, and this.strategy = strategy is a statement of RentalPerk, and extends RentalStrategy is a part of statement of RentalPerk.

A decision may be associated with one or more variation points while a variation point may be associated with zero or more decisions.

The decision-code traces of a decision is formalized as a set of variation points, $vps = \{vp_1, vp_2 \dots, vp_n\}$ where $n \geq 1$. It is captured as the fourth element of the 4-tuple formalization of decision in section 3.1.

As illustrated in **Fig. 7**, d_6 's impacted variation points, $vps = \{vp_1\}$.

Chapter 4 Impacts of Design Decisions for Single Systems

Building on the formalization in Chapter 3, this chapter analyzes and formalizes the required rules and logics on the impacts of design decisions. The impact can be on other elements of DDM and the trace links to features and variation points. It also highlights the additional challenges to be addressed by tool developers, most of them can be attributed to the enforcement of the traceability rules.

4.1 Order in Applying the Implications of Decisions

The dependencies among the elements of DDM dictate the order of applying the implications of decisions. This order will also evolve as the elements of DDM and their dependencies are evolved. Traceability Rules 4, 6, 9, and 10 dictate ordering via predicate *precede* (introduced in section 3.2). In fact, each of the above traceability rules dictates the ordering in some way.

Note that Traceability Rule 2 does not dictate ordering among the candidate decisions for an issue occurrence; it however requires that one decision is selected among the candidate decisions. The candidate decisions that are not selected for an issue occurrence are omitted from ordering.

Consider only the following elements and dependencies from the running example:

$$\begin{aligned} I &= \{i_3, i_4\} \\ A &= \{a_4, a_5, a_6\} \\ D &= \{d_6, d_7, d_8, d_9\} \\ IA &= \{(i_3, a_4), (i_3, a_6), (i_4, a_5)\} \\ ID &= \{(i_3, d_6), (i_4, d_8)\} \\ DA &= \{(d_6, a_4), (d_8, a_5)\} \\ DD_{\text{comprise}} &= \{(d_6, d_7)\} \\ DI_{\text{constrain}} &= \{(d_6, i_4)\} \\ DD_{\text{forbid}} &= \{(d_8, d_6)\} \\ DD_{\text{resolve}} &= \{(d_9, d_8), (d_9, d_6)\} \end{aligned}$$

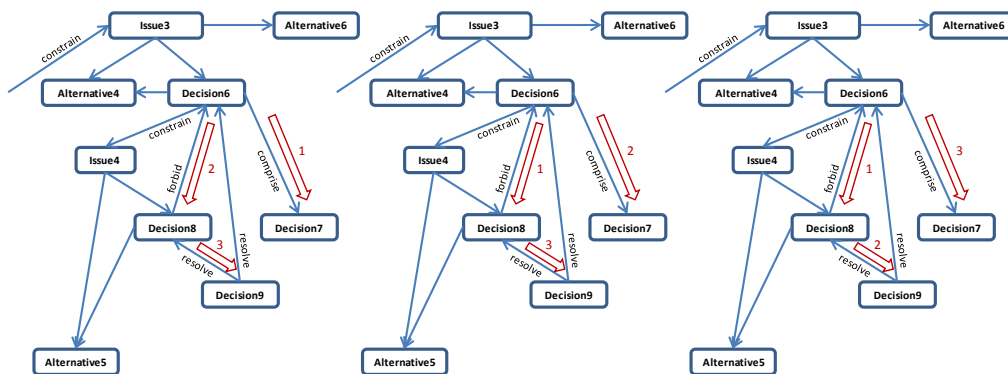


Fig. 11. Sample compliant chains for applying the implications of decisions.

As illustrated in **Fig. 11**, the following *chains of application* of decisions comply with the traceability rules:

$d_6-d_7-d_8-d_9$ or

$d_6-d_8-d_7-d_9$ or

$d_6-d_8-d_9-d_7$.

Any of these chains will consistently impact, via variation points, on the same set of code. Any other ordering may result in unexpected impact on code. As a counterexample, if d_9 precedes d_6 , VP3 configured by d_6 would not be included when it is required by d_9 .

A tool requires an **ordering mechanism** to *analyze all the applicable precedence between the decisions and propose the chains of application*. As the number of applicable precedence increases, the number of possible chains combinatorially explodes. These chains must comply with the ordering dictated by the applicable traceability rules at all times; they must adapt accordingly as decisions and dependencies evolve. Furthermore, the sheer number of possible chains is a cognitive challenge when evolving decisions and dependencies, the ordering mechanism should mitigate that by *recommending the preferred chain based on some prioritization scheme*. For instance, the prioritization scheme can assign different weights to different

types of association; the preferred chain can be a chain that complies with the traceability rules with weight as an additional ordering criterion.

Assume a prioritization scheme that assigns descending weights to Constrain with Forbid (4), Comprise (3), Constrain (2), and Resolve (1) associations, the preferred chain could be:

d₆-d₈-d₉-d₇

Without such an ordering mechanism, the implications of decisions cannot be automatically sequenced in the right order to correctly affect their impacts on variation points. Without a prioritization scheme, it is cognitively complicated for the domain engineers to evaluate impacts when evolving decisions and their dependencies. The next 3 sections analyze the impacts on the chains of application as decisions and their dependencies evolve.

4.2 Evolution of Decision and its Ripple

The implication of a decision is “hard-wired”. As the decision itself is evolved, the implication may also change in terms of the impact on the variation points in code. The change in the implication of a decision on the variation points may further impact its dependant decisions. Such changes in implications and their orderly propagation can be complicated. Consider chain d₆-d₇-d₈-d₉ for the samples below.

4.2.1 Evolution of Decision

A change in the “hard-wired” part of the implication of a decision results in changes, via variation points, in code.

An evolved d₇,

d₇' = (“Extensibility of rental perks”, *omitted*, *omitted*, {vp₂', vp₈'}) where vp₂' is an evolved vp₂ and vp₈' is a newly introduced variation point.

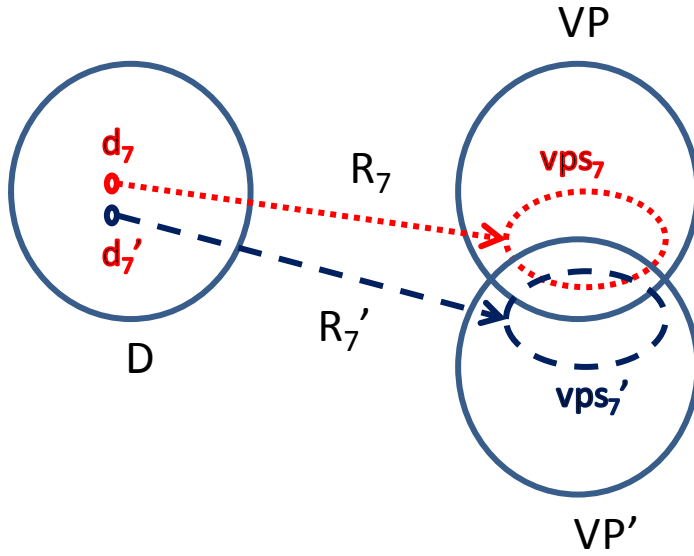


Fig. 12. Sample mappings from decisions to variation points for the evolution of a decision.

As illustrated in **Fig. 12**, the mapping of d_7 to vps_7 (variation points) can be formalized as a relation R_7 from D to VP where D is the set of all decisions; VP is the set of all variation points (shared by all the decisions). However, R_7 does not cater to the evolution of d_7 . A new relation R_7' is required to map from d_7' to a new vps_7' . The evolution of R_7 to R_7' is formalized below:

$$(d_7, vps_7) \in R_7, (d_7', vps_7') \in R_7' \text{ where}$$

vps_7' is the set of variation points for d_7' where $vps_7' \in VP'$.

As $vps_7 \neq vps_7'$, the variation points (and hence code) are impacted as d_7 is evolved to d_7' .

4.2.2 Ripple

So, the evolution of a decision impacts vps (i.e., its set of variation points). As vps is a premise of the dependent decisions, this change in vps may invalidate that premise; requiring dependent decisions to be individually assessed for impacts along the chain. Traceability Rules 4, 6, 9, and 10 dictate the decision-decision precedence, predicate *precede* (introduced in section 3.2) has further implication as specified in Rule 11 below.

Traceability Rule 11: Ripple of the evolution of a decision to its descendants.

evolve(d_j) AND precede(d_j, d_k) => assess(d_k)

where predicate **evolve(d_j)** means d_j is evolved; and predicate **assess(d_k)** means d_k is evaluated for impact and may result in evolve(d_k).

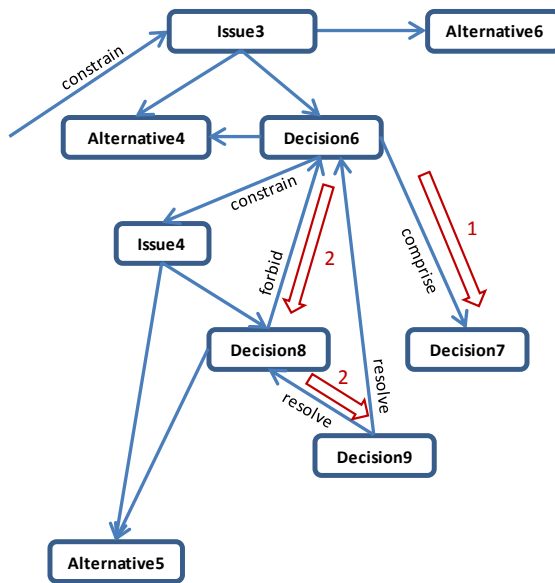


Fig. 13. Sample ripples for the evolution of a decision.

As illustrated in **Fig. 13**, applying Traceability Rule 11 on chain d₆-d₇-d₈-d₉, the possible “waves” of impacts that must be evaluated are d₆-d₇ and d₆-d₈-d₉. I refer to such a “wave” of impact from a decision onto its dependent decisions as a *ripple*. One possible result is *ripples across multiple dependent decisions*. The impact of these ripples must be manually assessed. A worse result is that the *premise of some dependent decision becomes invalid, requiring some form of redesign*: use of a new alternative solution, removal of the subject decision, etc. Such redesign may also cause more ripples. Traceability Rule 11 enables the automated identification of the potential impacts when evolving a decision, minimizing possible misses if assessed manually.

4.3 Addition/removal of Elements of DDM

As part of the maintenance of a software system, the elements of DDM may be evolved (as discussed in section 4.2), added or removed as the design for the features changes. These changes must comply with the traceability rules with their ripples properly evaluated. Such an action may cause DDM to be incomplete, requiring other actions to mend it.

4.3.1 Issue occurrence-alternative Association

Based on Traceability Rule 1, an issue occurrence should have at least one alternative solution that can solve it.

Assuming $i_j \in I$, $a_k \in A$, $(i_j, a_k) \in IA$:

- a_k can be removed individually, resulting in $a_k \notin A$, $(i_j, a_k) \notin IA$; DDM is incomplete until $\exists a \in A: (i_j, a) \in IA$.
- i_j can be removed individually, resulting in $i_j \notin I$, $(i_j, a_k) \notin IA$.

Assuming $i_j \notin I$: i_j can be added as an issue occurrence, resulting in $i_j \in I$; DDM is incomplete until $\exists a \in A: (i_j, a) \in IA$.

Assuming $i_j \in I$: a_k can be added as an alternative for i_j , resulting in $a_k \in A$, $(i_j, a_k) \in IA$. Note that a_k can be pre-existing or newly added.

4.3.2 Issue occurrence-decision Association

Based on Traceability Rule 2, an issue occurrence should have at least one decision that instantiates an alternative solution to solve it.

Assuming $i_j \in I$, $d_k \in D$, $(i_j, d_k) \in ID$:

- d_k can be removed individually, resulting in $d_k \notin D$, $(i_j, d_k) \notin ID$; DDM is incomplete until $\exists d \in D: (i_j, d) \in ID$, $\exists !d \in D: ((i_j, d) \in ID \text{ AND } \text{selected}(d))$.
- i_j can only be removed together with d_k , resulting in $i_j \notin I$, $d_k \notin D$, $(i_j, d_k) \notin ID$.

Assuming $i_j \notin I$: i_j can be added as an issue occurrence, resulting in $i_j \in I$; DDM is incomplete until $\exists d \in D: (i_j, d) \in ID, \exists! d \in D: ((i_j, d) \in ID \text{ AND selected}(d))$.

Assuming $i_j \in I, d_k \notin D$: d_k can be added as a decision addressing i_j , resulting in $d_k \in D, (i_j, d_k) \in ID$; DDM is incomplete until $\exists! d \in D: ((i_j, d) \in ID \text{ AND selected}(d))$.

4.3.3 Decision-alternative Association

Based on Traceability Rule 3, a decision should instantiate exactly one alternative solution if it addresses an issue occurrence. Otherwise, it does not require an alternative solution.

Assuming $d_j \in D, (i, d_j) \in ID, a_k \in A, (d_j, a_k) \in DA$:

- a_k can only be removed together with d_j , resulting in $d_j \notin D, a_k \notin A, (d_j, a_k) \notin DA$.
- d_j can be removed individually, resulting in $d_j \notin D, (d_j, a_k) \notin DA$.

Assuming $a_k \notin A$: a_k can be added as an alternative solution, resulting in $a_k \in A$.

Assuming $a_k \in A, d_j \notin D$: d_j can be added as a decision adopting a_k , resulting in $d_j \in D, (d_j, a_k) \in DA$.

4.3.4 Comprise Association

Based on Traceability Rule 4, d_j and d_k must exist together if one comprises the other and vice-versa.

Assuming $d_j \in D, d_k \in D, (d_j, d_k) \in DD_{\text{comprise}}$: d_j must be removed if d_k is removed and vice-versa, resulting in $d_j \notin D, d_k \notin D, (d_j, d_k) \notin DD_{\text{comprise}}$.

Assuming $d_j \in D, d_k \notin D$: d_k can be added as a part of d_j , resulting in $d_k \in D, (d_j, d_k) \in DD_{\text{comprise}}$.

4.3.5 Constrain Association

Based on Traceability Rules 6 and 7, d_j constrains d_l via i_k .

Assuming $d_j \in D$, $i_k \in I$, $d_l \in D$, $(d_j, i_k) \in DI_{\text{constrain}}$, $(i_k, d_l) \in ID$, then $(d_j, d_l) \in DD_{\text{constrain}}$:

- d_l can be removed individually, resulting in $d_l \notin D$, $(i_k, d_l) \notin ID$, $(d_j, d_l) \notin DD_{\text{constrain}}$; DDM is incomplete until $\exists! d \in D: (i_k, d) \in ID$.
- d_j can only be removed together with i_k and d_l resulting in $d_j \notin D$, $i_k \notin I$, $d_l \notin D$, $(d_j, i_k) \notin DI_{\text{constrain}}$, $(i_k, d_l) \notin ID$, $(d_j, d_l) \notin DD_{\text{constrain}}$.

Assuming $d_j \in D$, $i_k \notin I$: i_k can be added as an issue occurrence arises due to d_j , resulting in $i_k \in I$, $(d_j, i_k) \in DI_{\text{constrain}}$; DDM is incomplete until $\exists! d \in D: (i_k, d) \in ID$.

Assuming $d_j \in D$, $i_k \in I$, $(d_j, i_k) \in DI$: d_l can be added to address i_k , resulting in $d_l \in D$, $(i_k, d_l) \in ID$, $(d_j, d_l) \in DD_{\text{constrain}}$.

4.3.6 Forbid and Resolve Associations

Based on Traceability Rule 8, a forbid association exists with 2 resolve associations.

Assuming $\{d_j, d_k, d_r\} \subseteq D$, $(d_k, d_j) \in DD_{\text{forbid}}$, $\{(d_r, d_j), (d_r, d_k)\} \subseteq DD_{\text{resolve}}$:

- d_r can be removed individually, resulting in $d_r \notin D$, $(d_r, d_j) \notin DD_{\text{resolve}}$, $(d_r, d_k) \notin DD_{\text{resolve}}$; DDM is incomplete until $\exists d \in D: \{(d, d_j), (d, d_k)\} \subseteq DD_{\text{resolve}}$.
- d_k can only be removed together with d_r , resulting in $d_k \notin D$, $d_r \notin D$, $(d_k, d_j) \notin DD_{\text{forbid}}$, $(d_r, d_j) \notin DD_{\text{resolve}}$, $(d_r, d_k) \notin DD_{\text{resolve}}$.
- d_j can only be removed together with d_k and d_r , resulting in $d_j \notin D$, $d_k \notin D$, $d_r \notin D$, $(d_k, d_j) \notin DD_{\text{forbid}}$, $(d_r, d_j) \notin DD_{\text{resolve}}$, $(d_r, d_k) \notin DD_{\text{resolve}}$.

Assuming $d_j \in D$, $d_k \notin D$: d_k can be added to conflict with d_j , resulting in $d_k \in D$, $(d_k, d_j) \in DD_{\text{forbid}}$; DDM is incomplete until $\exists d \in D: \{(d, d_j), (d, d_k)\} \subseteq DD_{\text{resolve}}$.

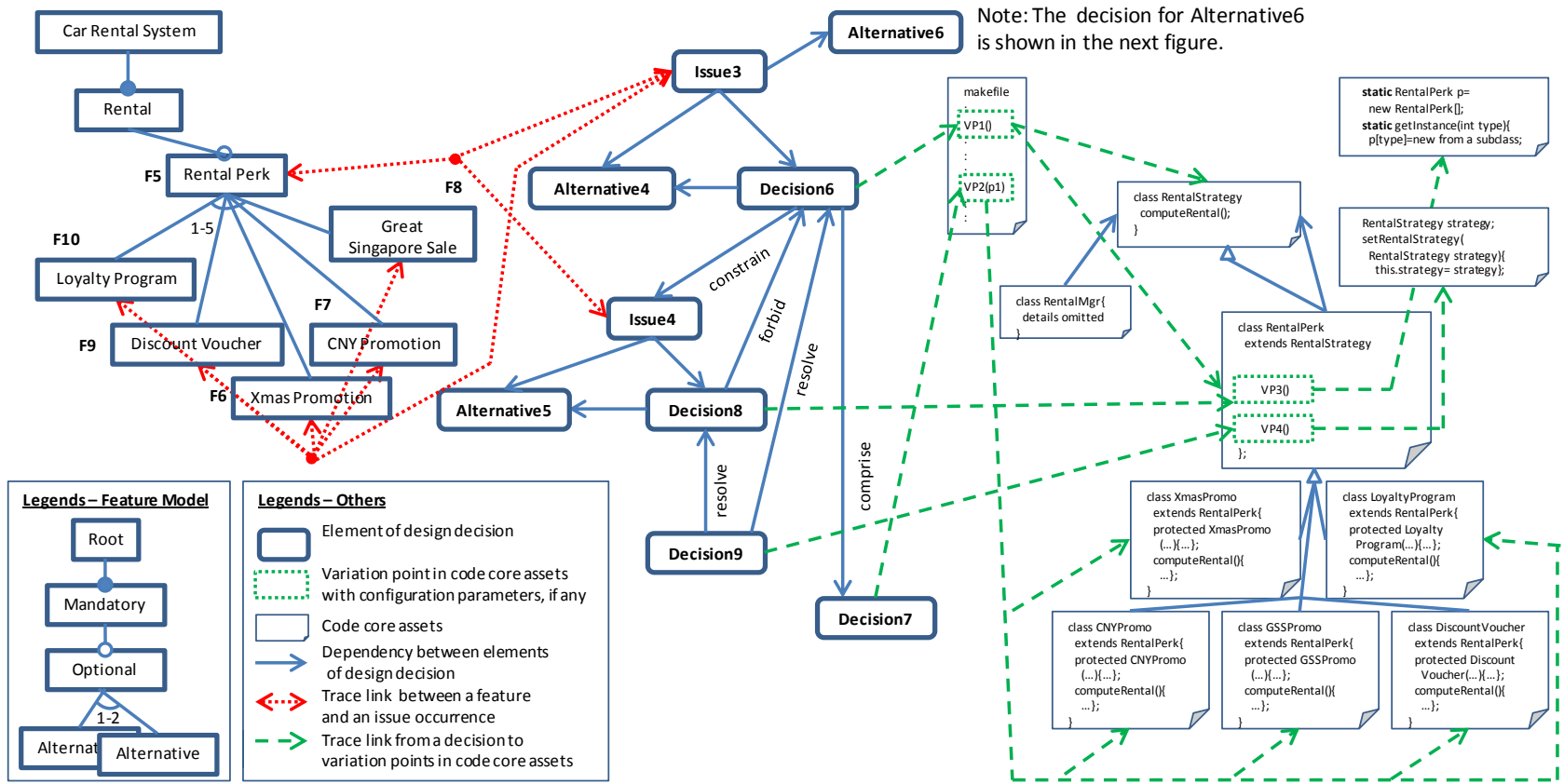
Assuming $d_j \in D$, $d_k \in D$, $(d_k, d_j) \in DD_{\text{forbid}}$: d_r can be added to resolve (d_k, d_j) , resulting in $d_r \in D$, $\{(d_r, d_j), (d_r, d_k)\} \subseteq DD_{\text{resolve}}$.

Chapter 5 Extension for Software Product Lines

This chapter extends the formalization of design decisions in Chapter 3 and Chapter 4 for the SPL situation. The running example in section 2.2 is also suitably extended here. As compared to a single system, an SPL has features which are mandatory or variant (i.e., either optional or alternative). This variability in features is a new dimension to be supported in my model. The core of this dimension is that variant features and their associated design information can emerge or vanish as they are selected or deselected for an SPL application. The challenges for my model include how variant features are represented, how variability in features is incorporated into DDM, and how variability in features is supported by the variation points in code core assets.

5.1 Extension of the Running Example

Fig. 14 and **Fig. 15** extend the sample DDM in **Fig. 7** and **Fig. 8** for the SPL situation. The original fragments of feature tree are now replaced with fragments of feature model, providing for the presence of variant features. For instances, features Rental, Rental Perk, and Xmas Promotion are now labeled respectively as mandatory, optional and alternative. An optional or alternative feature is only present in an SPL application if it is selected for the SPL application. The elements of DDM now provide for the emergence or the vanishing of variant features via the feature-issue occurrence traces. The variation points in code core assets now provide for the emergence or the vanishing of decisions in DDM. For instances, VP2 and VP6 are now configurable via a parameter which indicates the selection of the alternative features under Rental Perk.



Note: The decision for Alternative6 is shown in the next figure.

Fig. 14. Sample DDM with trace links from features to code core assets (extended for SPL).

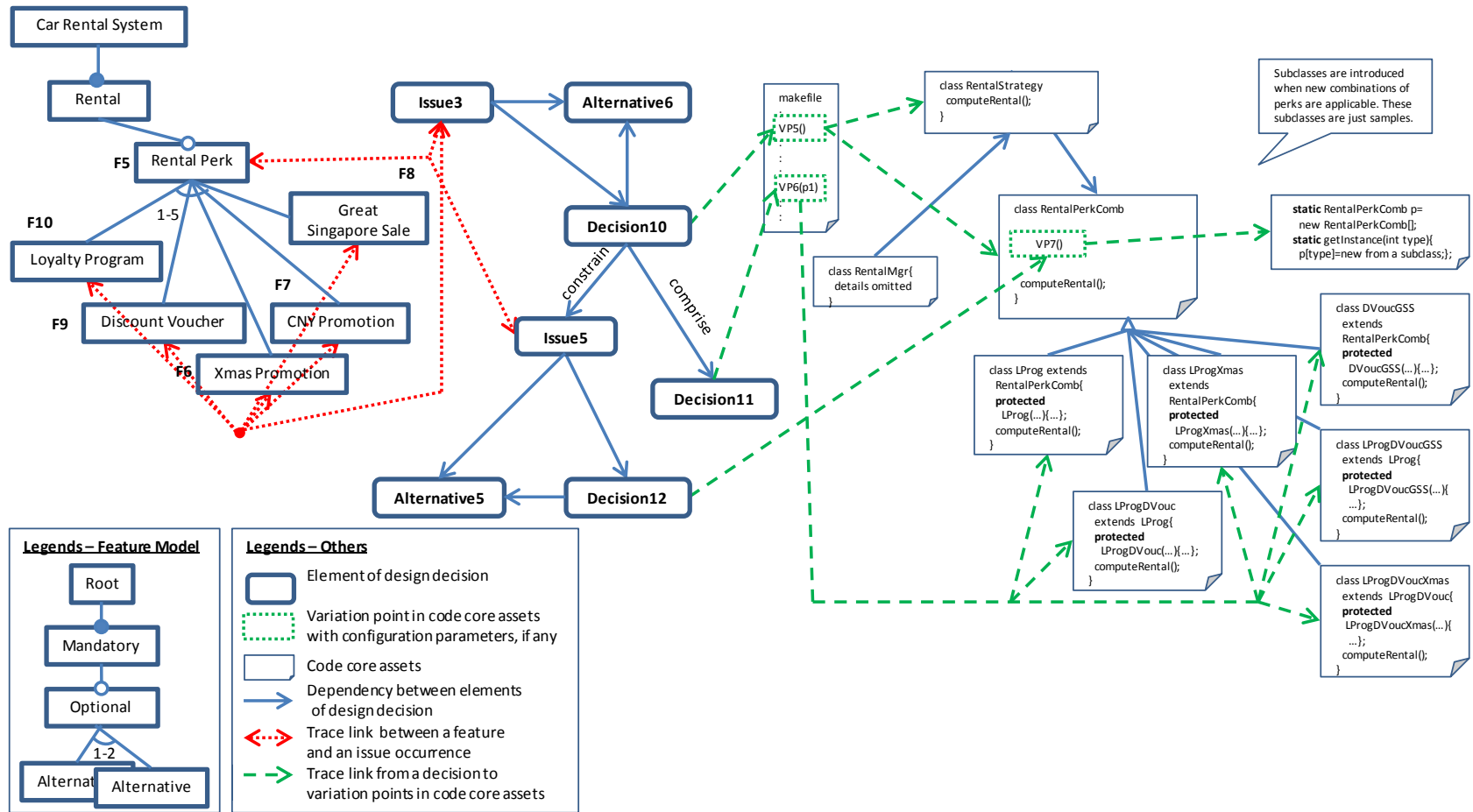


Fig. 15. Sample DDM with trace links from features to code core assets (the alternative solution for Issue3) (extended for SPL).

5.2 Extension of the Abstract Syntax

The abstract syntax for single systems as specified in Chapter 3 is generally applicable to the SPL situation. The following subsections identify the required extensions.

5.2.1 Scoping of DDM based on Feature Configuration

For a single system, all the features in its feature tree are applicable at the same time. The issue occurrences that arise in all these features as well as other elements of DDM associated with these issue occurrences are also applicable at the same time. That is, the traceability rules should be applied to check the integrity of DDM as a whole.

In the SPL situation, a feature model specifies the variability in features and implies a set of feature configurations. A feature configuration is a valid set of features for the feature model. For each feature configuration, a subset of the issue occurrences and other elements of DDM associated with these issue occurrences are applicable at the same time. That is, the traceability rules should be applied to check the integrity of DDM on a per feature configuration basis.

I refer to the identification of the elements of DDM which are within the scope of consideration for integrity check for a given feature configuration as *scoping*. So, for the SPL situation, DDM should be *scoped* first before the traceability rules are applied. Since a feature model represents several feature configurations, DDM must be repeatedly scoped for each feature configuration and checked for integrity – The required effort to manually conduct such checking also increases proportionally.

In order to support scoping, a set of *scoping rules* are introduced in the following sections.

5.2.2 Elements of DDM

5.2.2.1 Scoping of Elements of DDM

Among the elements of DDM, the issue occurrences and the decisions may be scoped in or out for a given feature configuration. The alternatives are not affected as they are generic design solutions that are not specific to any features. Predicate *in_scope* is added to the abstract syntax to represent the scoping in of an element of DDM.

Scoping Rule 1: Scoping of issue occurrences and decisions.

$i \in I, d \in D$

$in_scope(i)$

$in_scope(d)$

5.2.2.2 Scoping in Variation Points in Code Core Assets

For single systems, a mechanism is required to map from a decision to the applicable variation points and the specific parameters, if any, of each variation point. In the SPL situation, the impact of a decision on the variation points may vary as the variant features associated with the issue occurrence change due to feature selection. The mechanism has to be enhanced to account for the variant features associated with the issue occurrence. The following sample mappings are provided for the impact of d_6 , d_7 , d_8 , and d_9 :

d_6 maps to vp_1 ;

d_7 maps to vp_2 and its parameter p_1 ;

d_8 maps to vp_3 ;

d_9 maps to vp_4 .

Note that vp_1 corresponds to f_5 which is an optional feature. If f_5 is selected, the code configured by vp_1 is included. Parameter p_1 of vp_2 is newly introduced to provide for the alternative features f_6 through f_{10} . Depending on the selection of these alternative features, vp_2 is configured via its p_1 to include the relevant code.

5.2.3 Dependencies between Elements of DDM

If an issue occurrence is in scope, it follows that the decisions that solve the issue occurrence are also in scope. If a decision is in scope, it follows that the decisions that it comprises are also in scope; it also follows that the issue occurrences that it gives rise to and constrains are also in scope. If two conflicting decisions are in scope, it follows that the resolution (also a decision) of the conflict is also in scope.

Scoping Rule 2: Transitivity of scoping in issue occurrence-decision association.

$$\mathbf{i \in I, d \in D}$$

$$\mathbf{in_scope(i) \text{ AND } (i, d) \in ID \Rightarrow in_scope(d)}$$

Scoping Rule 3: Transitivity of scoping in comprise association.

$$\mathbf{d_j \in D, d_k \in D}$$

$$\mathbf{in_scope(d_j) \text{ AND } (d_j, d_k) \in DD_{comprise} \Rightarrow in_scope(d_k)}$$

Scoping Rule 4: Transitivity of scoping in constrain association.

$$\mathbf{i \in I, d \in D}$$

$$\mathbf{in_scope(d) \text{ AND } (d, i) \in DI \Rightarrow in_scope(i)}$$

Scoping Rule 5: Transitivity of scoping in forbid and resolve associations.

$$\mathbf{d_j \in D, d_k \in D, d_r \in D}$$

$$\mathbf{in_scope(d_k) \text{ AND } in_scope(d_j) \text{ AND } (d_k, d_j) \in DD_{forbid} \text{ AND } \{(d_r, d_j), (d_r, d_k)\} \subseteq DD_{resolve} \Rightarrow in_scope(d_r)}$$

5.2.4 Trace Links

If a variant feature in the feature model is selected to be in a feature configuration, it is then in scope. If a feature is in scope, it follows that the issue occurrences that arise in the design of the feature is also in scope.

Scoping Rule 6: Transitivity of scoping in feature-issue occurrence trace.

$f \in F, i \in I$

$\text{in_scope}(f) \text{ AND } (f, i) \in \text{FI} \Rightarrow \text{in_scope}(i)$

5.3 Extension of the Impacts of Design Decisions

After adjusting for the scoping of DDM as specified in section 5.2, the impacts of design decisions for single systems as specified in Chapter 4 is generally applicable to the SPL situation. The following subsections identify the required extensions.

5.3.1 Evolution of Decision and its Ripple

The implication of a decision changes as the selection of the variant features associated with the issue occurrence it solves changes. In contrast to the “hard-wired” implication in section 4.2, the implication of a decision due to the selection of variant features can be planned and configured via parameters.

A change in the selection of the variant features associated with the issue occurrence that a decision solves results in changes, via variation points, in code. Such evolution can be planned and be easily affected by taking the associated variant features as an input parameter of the decision.

Take the feature-issue occurrence traces for i_3 ,

$\text{FI}_3 = \{(f_6, i_3), (f_7, i_3), (f_8, i_3), (f_9, i_3), (f_{10}, i_3)\}$.

Also, $(i_3, d_6) \in \text{ID}$ and $(d_6, d_7) \in \text{DD}_{\text{comprise}} \Rightarrow (i_3, d_7) \in \text{ID}$. A change in the associated variant features of i_3 may result in the evolved feature-issue occurrence traces for i_3 ,

$\text{FI}_3' = \{(f_6, i_3), (f_7, i_3), (f_8, i_3)\}$ where f_9 and f_{10} are disassociated from i_3 .

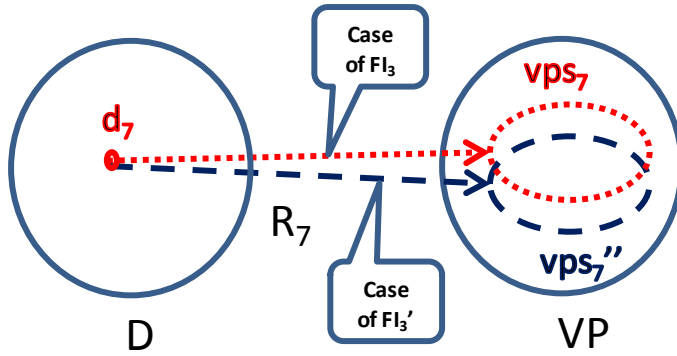


Fig. 16. Sample mappings from features to variation points for the evolution of decisions (extended for SPL).

As illustrated in **Fig. 16**, the mapping of d_7 (with FI_3 taken as input parameter) to vps (variation points) can be formalized as relation R_7 from D to VP where D is the set of all decisions; VP is the set of all variation points (shared by all decisions). The evolution of d_7 as FI_3 changes is formalized below:

$$\{(FI_3, vps_7), (FI_3', vps_7'')\} \subseteq R_7 \text{ where}$$

FI_3 and FI_3' are instances of input parameter of d_7 ;

vps_7 and vps_7'' are instances of vps of d_7 where $vps_7 \in VP$, $vps_7'' \in VP$.

As $vps_7 \neq vps_7''$, variation points (and hence code) are impacted.

Since the evolution of decision due to variant features is planned (as discussed above), there is no need to consider the ripples as in the case of the evolution of the decision itself (as discussed in section 4.2.2).

Chapter 6 Validation by Usage Examples

This chapter validates DDM and the impacts of design decisions by means of usage examples. Using step by step illustration, I demonstrate the applicability of the rules and/or the logics from the formalization in Chapter 3 and Chapter 4 in:

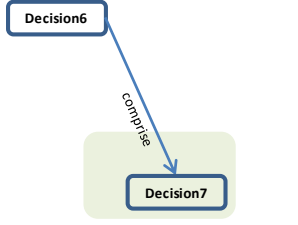
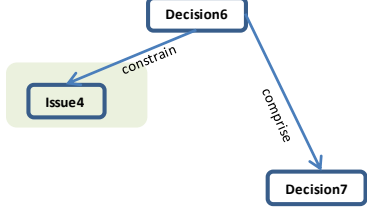
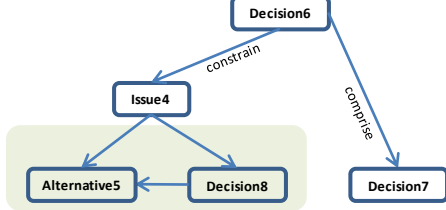
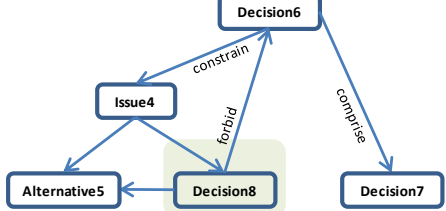
- **Constructing** the DDM in **Fig. 7** from section 2.2 from scratch given an existing feature model and code.
- Showing how the **ordering mechanism** and **prioritization scheme** help in understanding the impacts of design decisions for the constructed DDM.
- Evolving the constructed DDM with *salient evolution scenarios*.

These usage examples represent means of validating my proposed formalization by manual inspection. It provides the confidence on the practicality of using the formalization in a design support IDE.

I do not provide usage examples for the SPL situation. What sections 6.1, 6.2, and 6.3 illustrate is equivalent to one feature configuration of an SPL. The same rigor is required for each feature configuration of the SPL.

6.1 Construction of DDM

Step	Rule & Logic	Evolution of DDM
1	<p>Assume Decision1 to exist. It gives rise to Issue3.</p> <p>→ Issue3 is added and associated with Decision1 using a Constrain association.</p>	
2	<p><i>Traceability Rule 1</i> requires an issue occurrence to have at least one alternative. Its logic specifies how to add an issue occurrence to complete DDM.</p> <p>→ Alternative4 & Alternative6 are added and associated with Issue3.</p>	
3	<p><i>Traceability Rule 2</i> requires an issue occurrence to have at least one decision. Its logic specifies how to add a decision to complete DDM.</p> <p>→ Decision6 is added and associated with Issue3.</p>	
4	<p><i>Traceability Rule 3</i> requires a decision to have exactly one alternative if it addresses an issue occurrence. Its logic specifies how to add an alternative to complete DDM.</p> <p>→ Alternative4 is associated with Decision6.</p>	

Step	Rule & Logic	Evolution of DDM
5	Decision7 is a composite part of Decision6. → Decision6 is associated with Decision7 using a Comprise association.	 <pre> graph TD D6[Decision6] -- comprise --> D7[Decision7] </pre>
6	Decision6 also gives rise to Issue4. → Issue4 is added and associated with Decision6 using a Constrain association.	 <pre> graph TD D6[Decision6] -- constrain --> I4[Issue4] D6 -- comprise --> D7[Decision7] </pre>
7	Applying <i>Traceability Rules 1, 2 & 3</i> as in steps 2, 3 & 4. → Alternative5 is added and associated with Issue4. → Decision8 is added and associated with Issue4. → Alternative5 is associated with Decision8.	 <pre> graph TD D6[Decision6] -- constrain --> I4[Issue4] I4 -- constrain --> A5[Alternative5] I4 -- constrain --> D8[Decision8] D8 --> A5 D6 -- comprise --> D7[Decision7] </pre>
8	The implication of Decision8 conflicts with that of Decision6 in code. → Decision8 is associated with Decision6 using a Forbid association.	 <pre> graph TD D6[Decision6] -- constrain --> I4[Issue4] D6 -- forbid --> D8[Decision8] D6 -- comprise --> D7[Decision7] I4 -- constrain --> A5[Alternative5] I4 -- constrain --> D8 D8 --> A5 </pre>

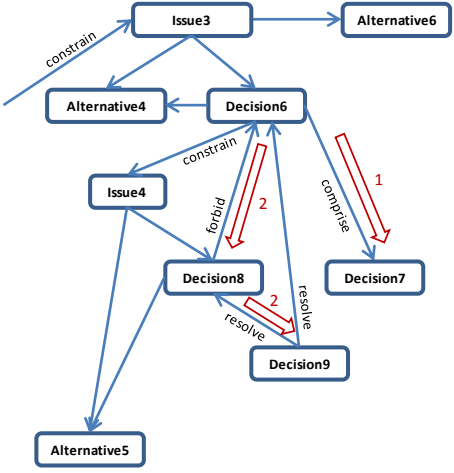
Step	Rule & Logic	Evolution of DDM
9	<p><i>Traceability Rule 8</i> requires a conflict between decisions to be resolved by a resolution decision. The implication of Decision9 resolves the conflict between Decision8 & Decision6 in code.</p> <p>→ Both Decision8 & Decision6 are associated with Decision9 using two Resolve associations.</p> <p>This completes the construction of DDM.</p>	<pre> graph TD D1[Decision1] -- constrain --> I3[Issue3] I3 --> A6[Alternative6] I3 --> D6[Decision6] A4[Alternative4] --> D6 I4[Issue4] -- constrain --> D6 I4 -- forbid --> D8[Decision8] A5[Alternative5] --> D8 D6 -- comprise --> D7[Decision7] D9[Decision9] -- resolve --> D6 D9 -- resolve --> D8 </pre>

6.2 Understanding the Impacts of DDM

Step	Rule & Logic	Evolution of DDM
10	<p><i>Traceability Rules 4, 6, 9, and 10</i> dictate the ordering of the application of decisions via predicate <i>precede</i>. An ordering mechanism that complies with these rules generates the compliant chains of applications for an instance of DDM.</p>	<p>Assume the instance of DDM in section 6.1. The compliant chains of application are:</p> <ul style="list-style-type: none"> • d6-d7-d8-d9 (Note: d6 is Decision6) • d6-d8-d7-d9 • d6-d8-d9-d7

Step	Rule & Logic	Evolution of DDM
11	<p>Cognitive challenge arises due to the combinatorial explosion of the number of chains as the number of applicable precedence increases. A prioritization scheme recommends the preferred chain.</p> <p>→ One simple prioritization scheme is to assign different weights to various types of association.</p>	<p>Assume a prioritization scheme that assigns descending weights to Constrain with Forbid (4), Comprise (3), Constrain (2), and Resolve (1) associations, the preferred chain could be:</p> <ul style="list-style-type: none"> d6-d8-d9-d7
12	Same as step 10.	<p>To consider the candidate decisions d6 and d10 for issue occurrence i3, expand DDM to include d1 and d6 through d12 in Fig. 7 and Fig. 8. The compliant chains of applications are:</p> <ul style="list-style-type: none"> d1-d6-d7-d8-d9 d1-d6-d8-d7-d9 d1-d6-d8-d9-d7 d1-d10-d11-d12 d1-d10-d12-d11 <p>As observed in the above chains, d6 and d10 are mutually exclusive.</p>

6.3 Evolution of DDM

Step	Rule & Logic	Evolution of DDM
13	<p><i>Traceability Rule 11</i> identifies the ripples due to the evolution of a decision. Each ripple must be evaluated and the impact handled accordingly.</p>	<p>If d8 is evolved, the ripples are:</p> <ul style="list-style-type: none"> d8-d9 <p>If d6 is evolved, the ripples are:</p> <ul style="list-style-type: none"> d6-d7 d6-d8-d9 
14	<p>To remove a decision, the following rules apply.</p> <p><i>Traceability Rule 2:</i> If a decision associated with an issue occurrence is removed, the model is incomplete till another decision is associated with the issue.</p> <p><i>Traceability Rule 3:</i> If a decision associated with an issue occurrence is removed, the issue occurrence can continue to exist.</p> <p><i>Traceability Rule 4:</i> If a decision in a Comprise association is removed, the other decision in the association must be removed.</p>	<p><i>Note: Braces “{}” below reference the applicable rules from the “Rules & Logic” column.</i></p> <p>If d9 is removed:</p> <ul style="list-style-type: none"> another decision must replace d9 {8a}. <p>If d8 is removed:</p> <ul style="list-style-type: none"> d9 must be removed {8b}. another decision must replace d8 {2}. <p>If d6 is removed:</p> <ul style="list-style-type: none"> d8 & d9 must be removed {8c}. d7 must be removed {4}. i4 & d8 must be removed {6 & 7}.

Step	Rule & Logic	Evolution of DDM
	<p><i>Traceability Rules 6 & 7:</i> If a decision that constrains another decision via an issue occurrence is removed, the issue occurrence and the constrained decision must be removed.</p> <p><i>Traceability Rule 8a:</i> If a decision that resolves the conflict between two other decisions is removed, the model is incomplete till another decision that resolves the conflict exists.</p> <p><i>Traceability Rule 8b:</i> The forbidding decision of two conflicting decisions can be removed together with the decision that resolves the conflict.</p> <p><i>Traceability Rule 8c:</i> The forbidden decision of two conflicting can only be removed together with the forbidding decision and the decision that resolves the conflict.</p>	
15	<p>To remove an issue occurrence, the following rules apply.</p> <p><i>Traceability Rule 1:</i> If an issue occurrence is removed, the associated alternative can continue to exist.</p> <p><i>Traceability Rule 2:</i> If an issue occurrence is removed, the</p>	<p>If i4 is removed:</p> <ul style="list-style-type: none"> d8 must be removed {2}. <p>If i3 is removed:</p> <ul style="list-style-type: none"> d6 must be removed {2}. <ul style="list-style-type: none"> d8 & d9 must be removed (cf. Step 14). d7 must be removed (cf. Step 13).

Step	Rule & Logic	Evolution of DDM
	associated decision must be removed.	<ul style="list-style-type: none"> • i4 & d8 must be removed (cf. Step 14).
16	<p>To remove an alternative, the following rules apply.</p> <p><i>Traceability Rule 1:</i> If an alternative associated with an issue occurrence is removed, the model is incomplete till another alternative is associated with the issue occurrence.</p> <p><i>Traceability Rule 3:</i> If an alternative associated with a decision is removed, the decision must be removed.</p>	<p>If a5 (i.e., Alternative5) is removed:</p> <ul style="list-style-type: none"> • d8 must be removed {3}. • d9 must be removed (cf. Step 14). • another decision must replace d8 (cf. Step 14). • another alternative must replace a5 {1}. <p>If a4 is removed:</p> <ul style="list-style-type: none"> • d6 must be removed {3}. <ul style="list-style-type: none"> • d8 & d9 must be removed (cf. Step 13). • d7 must be removed (cf. Step 14). • i4 & d8 must be removed (cf. Step 14). <p>If a6 is removed:</p> <ul style="list-style-type: none"> • none.

Chapter 7 Verification by Formal Method

This chapter describes how formal method can be used to specify and verify the abstract syntax of DDM, instances of DDM, and feature configurations of instances, as well as to derive information from instances of DDM.

Note that I discuss together both the situations for single systems and SPLs. This is done by using a feature model to represent the feature variability in an SPL; and to represent the features of a single system using a feature model with only mandatory features (i.e., effectively a feature tree).

7.1 Use of Formal Method

The verification of the abstract syntax and its instances can be conducted through formal verification and/or manual inspection. Manual inspection is usually adopted in typical software development lifecycles. For DDM, it may include activities like peer review of the abstract syntax, peer review of the code of the support tool, and unit and system testing of the support tool. As these techniques demand human effort and skills, they are conducted with best effort which tends to be error-prone and non-exhaustive. In fact, it is practically impossible to manually cover all possible scenarios of the abstract syntax and the support tool.

Formal verification takes a very different approach. The structure and/or behavior of the test subject have to be specified in a formal language so that it can be verified formally using techniques like theorem proving. Once a specification is formally verified, all the possible scenarios are completely covered. As compared to manual inspection, in cases where formal verification is feasible, the latter can precisely and completely verify the test subject. This characteristic is the main motivation behind my proposal to formally verify DDM.

7.2 Alloy as a Formal Method Tool

In order to verify a test subject using formal method, it must firstly be possible to specify its structure and/or behavior without overly elaborate effort that negates the potential gains from formal verification. The approach used must also be computationally economical so that instances of verification test can be conducted within bearable time and be regressed as frequently as the test subject evolves.

In formal verification, test subjects are typically specified in a combination of predicate logic and first order logic. These representations vary in terms of expressiveness but are generally sufficient to capture structure and behaviors. The main problem lies in the computation of first order logic which is *undecidable* – It is impossible to compute if an assertion is valid, i.e. holds true for every possible assignment.

Alloy [7] is a structural modeling language based on first-order logic, for expressing complex structural constraints and behavior. The Alloy Analyzer is a constraint solver that provides fully automatic simulation and checking. There are two primary use cases. Firstly, as a *model checker*, it formally verifies a model against an abstract syntax and some properties. If the model is invalid, counterexamples are provided to help refine the model. Secondly, as a *model finder*, it formally derives a model that complies with an abstract syntax and some specified constraints, if any.

Alloy works around the undecidability of first order logic by introducing the notion of *scope* to limit the size of state space considered. This makes the earlier computation *tractable* within a scope of concern. The main compromise is that Alloy does not verify outside the specified scope. This is however mitigated, as claimed by the creator of Alloy, the *Small Scope Hypothesis* where most bugs can be found within small scopes. Furthermore, Alloy does more than a theorem prover in verifying an abstract syntax or its models; it goes a step further in suggesting counterexamples that help in debugging.

7.3 Overall Verification Approach using Alloy

This section describes the overall verification approach using Alloy. Alloy supports constructs like signature, relation, predicate, function, formula, fact, assertion, etc.

The abstract syntax of DDM is specified in Alloy. For instances:

- the decision as a signature
- the dependencies among the elements of DDM as relations
- the rules that enforce the integrity of DDM as predicates, arities of relations, etc.
- the rules that govern the impacts of decisions of DDM as functions, formulae, etc.
- the rules that scope the elements of DDM as predicates.

In order to reason about feature configurations, the abstract syntax of feature model (FM) is also specified in Alloy. With the abstract syntaxes specified, the instances of DDM and FM and any additional constraints are also specified in Alloy.

The above strategy makes it possible to formally reason on properties that encompass feature configurations (of an instance of FM) and/or design decisions (of an instance of DDM). The strategy can first be applied on the single system situation and then the SPL situation of the running example. This should identify issues which help debug and refine the formalization. The refined strategy can then be applied to the single system situation and then the SPL situation of an industry case study to show that it can scale up from the running example.

To show the infeasibility of exhaustive manual inspection of all possible verification scenarios, the complexity involved in various verification tasks (e.g., comparing planned against supported feature configurations) can be computed for comparison where applicable.

7.4 Specification and Verification of DDM and its Instances

Fig. 17 illustrates the approach for specifying and verifying the abstract syntax (including rules) of DDM (on the left of the figure) and its instances (on the right of the figure).

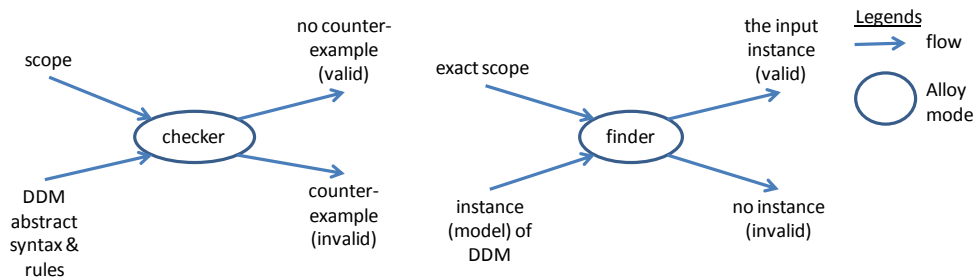


Fig. 17. Scheme for verifying the abstract syntax of DDM and its instances.

The following steps are to be performed:

1. Formally specify the abstract syntax (including rules) of DDM in Alloy language.
2. Formally verify this specification for a sufficiently large scope using Alloy as a model checker.
3. If there are no counterexamples, this abstract syntax is valid within the scope.
4. If there are counterexamples, this abstract syntax is invalid. They are used to refine the abstract syntax towards a valid one.
5. Formally specify an instance of DDM in Alloy language.
6. Formally verify this instance of DDM against the abstract syntax of DDM for an exact scope (of this instance) using Alloy as a model finder.
7. If the input instance is found, this instance is a valid model of DDM. That is, the design represented by the instance is verified to be consistent.
8. If no instance is found, this instance is an invalid model of DDM.

For step 8 above, the scheme cannot enumerate issues that result in the invalidity as counterexamples so that the instance can be refined towards a valid model of DDM using Alloy.

7.5 Specification and Verification of Feature Model and its Instances

Although the formalization of FM is not a subproblem of this thesis, it is required for supporting the reasoning related to feature configurations in section 7.6. This formalization can be based on [5] which proposes an abstract syntax for FM in Alloy. **Fig. 18** illustrates the approach for specifying and verifying the abstract syntax of FM (on the left of the figure) and its instances (on the right of the figure).

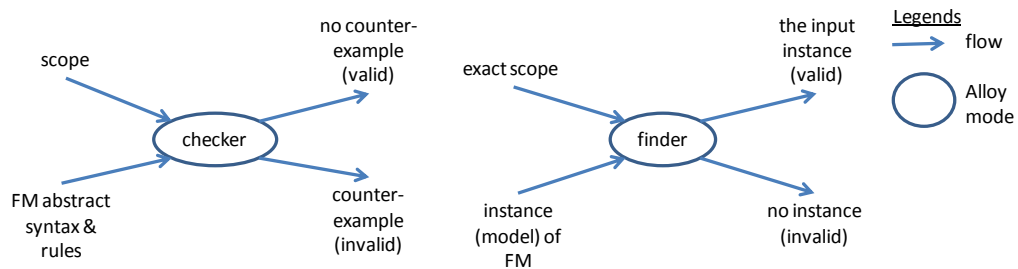


Fig. 18. Scheme for verifying the abstract syntax of FM and its instances.

The following steps are to be performed:

1. Formally specify the abstract syntax (including rules) of FM in Alloy language.
2. Formally verify this specification for a sufficiently large scope using Alloy as a model checker.
3. If there are no counterexamples, this abstract syntax is valid within the scope.
4. If there are counterexamples, this abstract syntax is invalid. They are used to refine the abstract syntax towards a valid one.
5. Formally specify an instance of FM in Alloy language.

6. Formally verify this instance of FM against the abstract syntax of FM for an exact scope (of this instance) using Alloy as a model finder.
7. If the input instance is found, this instance is a valid model of FM. That is, the design is verified to be consistent.
8. If no instance is found, this instance is an invalid model of FM.

For step 8 above, the scheme cannot enumerate issues that result in the invalidity as counterexamples so that the instance can be refined towards a valid model of FM using Alloy.

7.6 Comparison of Planned vs. Supported Feature Configurations

This section devises the scheme to address the third subproblem specified in section 2.1. **Fig. 19** illustrates the approach for comparing planned against supported feature configurations.

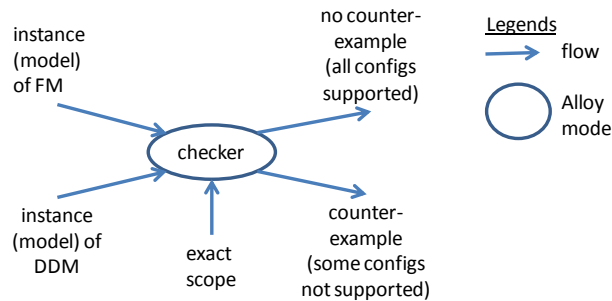


Fig. 19. Scheme for comparing planned against supported feature configurations.

A feature model implies a set of valid feature configurations, say FC_{fm} , which are *planned*. An instance of DDM *supports* a set of valid feature configurations, say FC_{ddm} , which are constrained by the design represented by the instance.

The following steps are to be performed:

1. Given an instance of DDM and an instance of FM.
2. Formally derive FC_{ddm} from the instance of DDM.

3. Formally derive FC_{fm} from the instance of FM.
4. Formally compare FC_{fm} and FC_{ddm} for an exact scope (of the instances of DDM and FM) using Alloy as a model checker.
5. If $FC_{fm} \subseteq FC_{ddm}$, all the planned feature configurations are supported by the design.
6. Otherwise, some planned feature configurations are not supported by the design. The issues are enumerated as counterexamples so that they can be resolved either by:
 - a. constraining FC_{fm} further by adding feature dependencies in the instance of FM.
 - b. expanding FC_{ddm} by refining the design in the instance of DDM.

7.7 Derivation of Information for a Feature Configuration from DDM

This section devises the scheme to address the fourth subproblem specified in section 2.1. As DDM is already equipped with the traceability capability from features through code, the main challenge is the derivation of the possible combinations of design decisions for a given feature configuration.

The following steps are to be performed:

1. Given an instance of DDM, an instance of FM, and a feature configuration FC_1 .
2. Let F_1 be the set of features that are in the scope of FC_1 .
Given $FC_1, \forall f \in F_1: \text{in_scope}(f)$
3. Using Scoping Rule 6, let I_1 be the set of issues that are in the scope of FC_1 .
Given $F_1, \forall i \in I_1: \text{in_scope}(i)$
4. Using Traceability Rule 2, let ID_1 be the set of issue occurrence-decision associations in the scope of FC_1 .
Given $I_1, \forall (i, d) \in ID_1: i \in I_1$

5. Let D_1 be the set of possible combinations of decisions in the scope of FC_1 .

Given $ID_1, \forall D \in D_1: (\exists !i \in I_1: ((i, d) \in ID_1))$

6. For each combination of decisions, $D \in D_1$, the Traceability Rules can be applied to derive the other elements of DDM that are in the scope of this combination of decisions.

7.8 Verification of Instances of DDM for the Addition and Removal of Elements of DDM

The elements of an instance of DDM may be added or removed as the design is evolved. These changes are more drastic as compared to the evolution of decisions. A change (e.g. the removal of an issue occurrence) could potentially *invalidate* other elements of the instance of DDM (e.g. the decision for the issue occurrence and other elements that depend on the decision). Hence, after one or more changes to the instance of DDM, it should be:

- formally verified that the model is still consistent – against the abstract syntax and rules of DDM. The inconsistencies, if any, are enumerated.
- formally verified that the planned feature configurations are still intact. The inconsistencies, if any, are enumerated.

The schemes for verifying the above using Alloy are described in section 7.4 and section 7.6 respectively.

Chapter 8 Implementation of Support IDEs

This chapter recommends to the tool developers how the key salient features of support IDEs adopting DDM can be implemented. I first summarize the additional challenges to be addressed by the tool developers, address them in my proposed solution, and then plan for a prototype tool.

Note that, in the same approach in Chapter 7, I discuss together both the situations for single systems and SPLs. SPL-specific considerations are highlighted accordingly.

8.1 Challenges for Tool Developers

The metamodel in **Fig. 5** of section 2.2 also incorporate the *additional dependencies* required: comprise, constrain, forbid, and resolve. Note that the enforcement of the 11 *traceability rules* for the integrity of DDM and trace links is already accounted for by the abstract syntax. There is no need for the tool developers to separately address them.

As discussed in section 3.1, a *mapping mechanism* is required to map from a decision to the applicable variation points and the specific parameters of each variation point. (SPL-specific) As discussed in section 5.2.2.2, the mechanism also has to account for the variant features associated with the issue occurrence that the decision addresses.

As discussed in section 4.1, an *ordering mechanism* is required to analyze all the applicable precedence between the decisions and propose the chains of application. It should also recommend the preferred chain based on some *prioritization scheme*.

As discussed in section 4.2.2, the solution should automatically identify potential impacts due to *ripples of the evolutions of the “hard-wired” part of decisions*.

(SPL-specific) As for the *scoping rules* introduced for the abstract syntax in section 5.2, they are already taken into account by DDM. There is no need for the tool developers to separately address them.

8.2 Solutions to Challenges

8.2.1 Metamodel for DDM

This section proposes a metamodel for DDM. It is adapted and enhanced from [11] and [4] that discuss architectural design decisions in single systems.

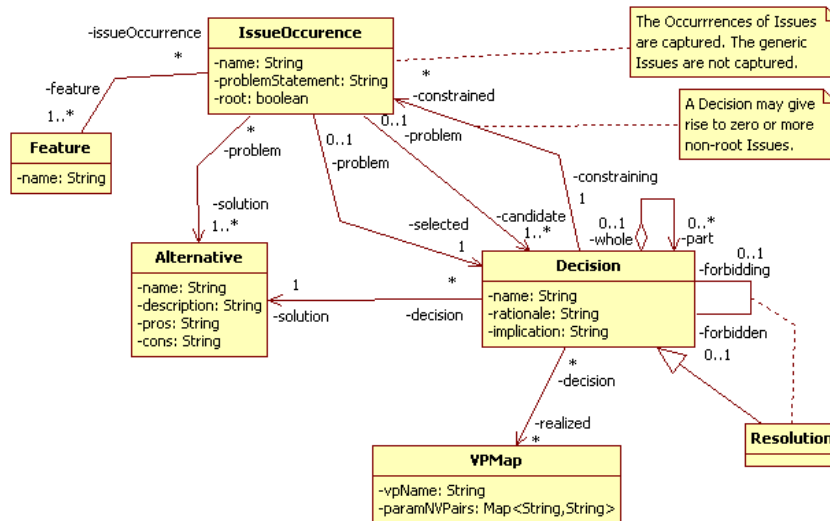


Fig. 20. Metamodel for DDM.

Fig. 20 illustrates an enhanced metamodel that takes into consideration the additional dependencies as identified in section 8.1. A `root` attribute is added to `IssueOccurrence` in order to differentiate between root and non-root issue occurrences. A non-root issue occurrence arises in the context of (i.e., constrained by) another decision; a root issue occurrence is one which is not constrained by another decision. There must be at least one such root issue occurrence in an instance of DDM. The “comprise” association is represented by the whole-part association of `Decision`. The “constrain” association is represented by the `constraining-constrained` association from

Decision to IssueOccurrence. The “forbid” association is represented by the forbidding-forbidden association of Decision. The “resolve” association is represented by the association class Resolution which inherits from Decision. As for the trace links, the feature-issue occurrence trace is represented by the feature-issueOccurrence association from IssueOccurrence to Feature. The decision-code trace is changed to the decision-realized association from Decision to VMap. VMap captures the mapping from a Decision to its impacted variation points. VMap.paramNVPairs captures the information required to configure parameters of a variation point in code.

8.2.2 Mapping Mechanism

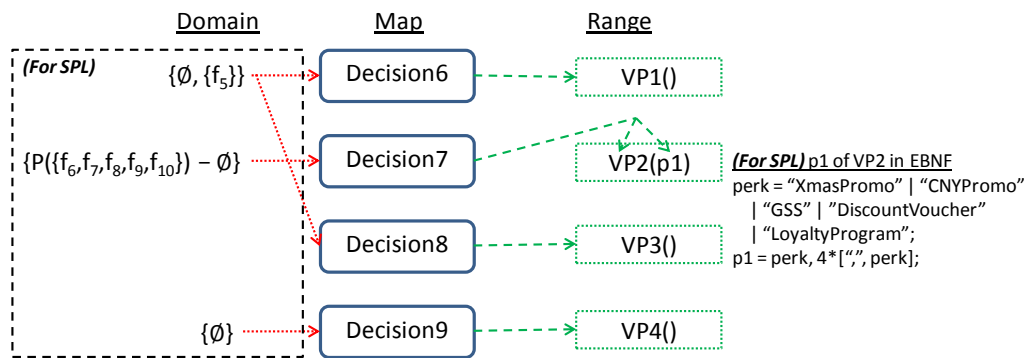


Fig. 21. Sample mappings for some decisions of the running example.

Fig. 21 above illustrates the required mappings for some decisions of the running example for both the situations for single systems and SPLs. It shows the role of a decision in mapping from the domain (i.e., the selected variant features via issue occurrences) to the range (i.e., the applicable variation points and the configuration of their parameters). As the domain accounts for variant features, it is only applicable to the SPL situation. The “p1” parameter is also introduced to account for the impact of the variant features on VP2.

For each decision, the corresponding mapping specifies:

- the set of valid feature configurations (It would be $\{\emptyset\}$ for the case of a single system)
- for each feature configuration, the set of impacted variation points and the value of each applicable parameter

The above specification is captured as `VPMAP` objects.

Based on the feature selection, the mapping mechanism computes for each decision:

- the impacted variation points
- the value of each applicable parameter of the variation points

If a specific parameter is impacted by multiple decisions, the impacts (i.e., values) are combined.

8.2.3 Variability Technique

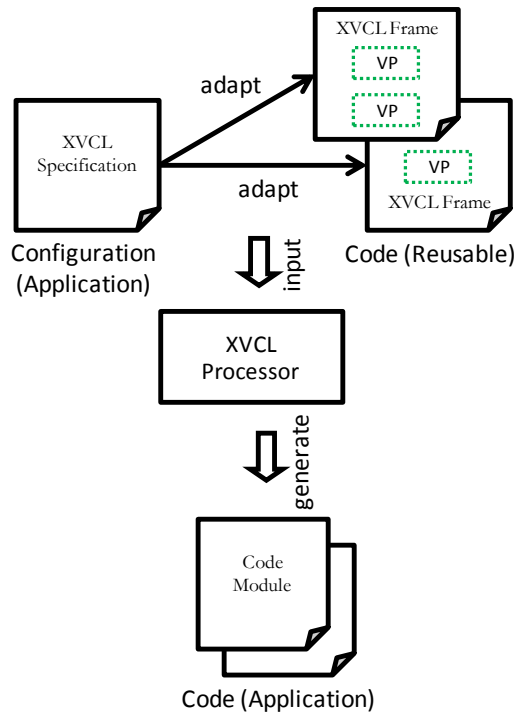


Fig. 22. XVCL as a variability technique.

A variability technique is required to include/exclude code based on the selection of variant features (for SPL) and candidate design decisions. I choose XML-based Variant Configuration Language (XVCL) [5] for its generic support on various formats of core assets and its independence from the syntaxes of programming languages.

Fig. 22 illustrates the use of XVCL as a mechanism to capture the configuration parameters required to assemble the application code from the reusable code as impacted by the decisions. An XVCL specification specifies how a set of XVCL frames are to be used as code templates. It can be parameterized to allow for variations in the use of these code templates. The required parameters for the applicable decisions are captured in $VPMap$ of Fig. 20. An XVCL specification is instantiated with parameter values and interpreted by the XVCL Processor to assemble the application code.

8.2.4 Metamodel for Feature Model

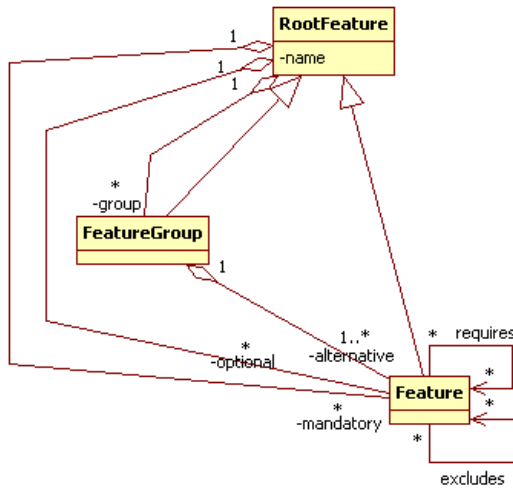


Fig. 23. Metamodel for feature model of FODA.

Fig. 23 illustrates a metamodel compatible with feature model of FODA [9]. It shall be the basis for my proposed prototype tool to support feature model.

8.2.5 Ordering Mechanism and Prioritization Scheme

To propose the chains of application of decisions, *rooted directed acyclic graph* (DAG) can be used as the ordering mechanism where the decisions are the vertices and the precedence dependencies between decisions are the edges. The topological sorts of such DAG are then the possible chains. To recommend the preferred chain among the possible chains, weighted vertices and edges can be used to accumulate the relative importance of a subgraph rooted at a vertex. The preferred chain would then be the one produced by prioritizing vertex weight.

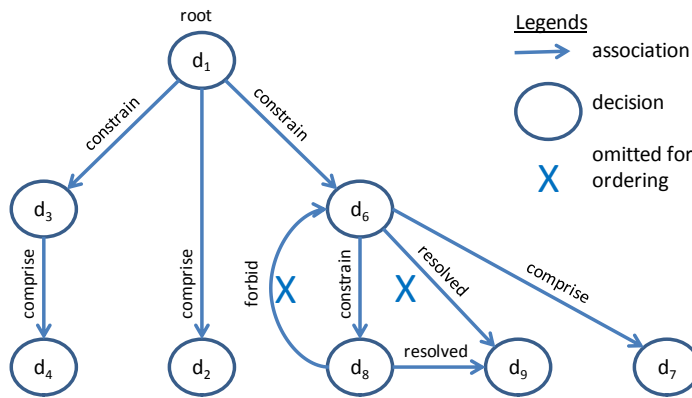


Fig. 24. Rooted directed acyclic graph for the ordering mechanism.

Fig. 24 illustrates the use of **rooted DAG** as the ordering mechanism where:

- decisions are the vertices
- precedence dependencies between decisions are the edges (forbid association & the corresponding resolved association from the forbidden decision are omitted)

Note: For readability, the “resolve” association is reversed in direction to become “resolved”.

The **topological sorts** of such DAG are then the possible chains.

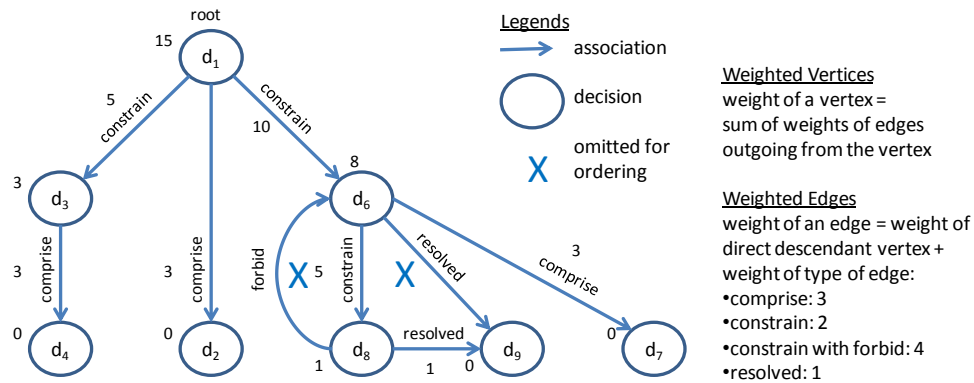


Fig. 25. Weighted rooted directed acyclic graph for the prioritization scheme.

Fig. 25 illustrates the use of **weighted vertices and edges** to accumulate the **relative importance of a subgraph** rooted at a vertex.

weight of a vertex = sum of weights of edges outgoing from the vertex

weight of an edge = weight of direct descendant vertex + weight of type of edge (comprise: 3, constrain: 2, constrain with forbid: 4, resolved: 1)

The **preferred chain** would then be the one produced by **prioritizing vertex weight**.

8.2.6 Ripple Mechanism

While there can be multiple ways in computing the ripples due to evolution of a decision, I propose a way that leverages on the already built **weighted rooted DAG**.

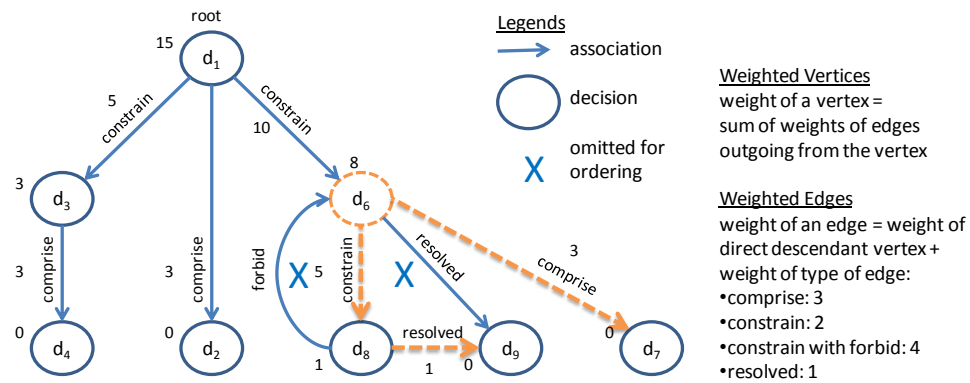


Fig. 26. Weighted rooted directed acyclic graph for the ripple mechanism.

Fig. 26 illustrates the computation of **ripples** as follows:

- Perform **topological sorting** for the subgraph rooted at the evolved vertex
- Transform the resultant topological sort into a **set of paths** from the evolved vertex
- Disregard any **duplicate paths**
- The **remaining paths** are the ripples

These ripples can then be **highlighted as the impacts** of an evolved decision. E.g., ripples for d_6 are shown in the figure.

8.3 Implementation Technologies

I highlight a few implementation technologies that facilitate the implementation of the proposed solutions in section 8.2.

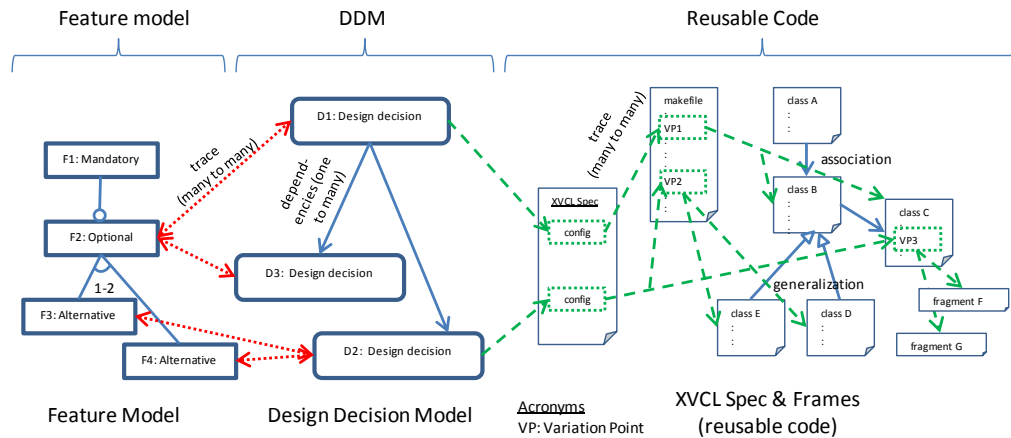


Fig. 27. Key artifacts to be managed by a typical SPL support IDE.

Fig. 27 illustrates the key artifacts to be managed by a typical SPL support IDE. These artifacts span feature model, DDM, and reusable code. By having only mandatory features on the feature model (equivalent to a feature tree), this IDE can be simplified to support single systems.

I propose the use of three key technologies: Domain-specific Language (DSL), Alloy, and XVCL.

DSL is used to code the metamodels and instantiate models for:

- feature model of FODA
- DDM
- trace links between features and DDM
- trace links between DDM and variation points in XVCL frames

Custom code on top of DSL implementation for:

- the mapping mechanism
- the ordering mechanism and prioritization scheme
- the ripple mechanism

Alloy is used for the formal verification of:

- the integrity of feature models of FODA
- the integrity of models of DDM based on the traceability rules
- the comparison of planned (of feature models of FODA) vs. supported (of models of DDM) feature configurations

XVCL specifications and frames are used to:

- capture fine-grained trace links from decisions to the impacted code
- automate the assembly of application code from the impacted code for selected variant features of an application

Chapter 9 Evaluation against Design Activities in Development Processes

This section analyzes the benefits the proposed DDM presents to the developers of single systems and SPLs by describing salient features of support IDEs built on top of the model. The tool is assumed to fulfill the criteria specified in Chapter 3, Chapter 4, and Chapter 5. It is evaluated against the design activities in the development processes of single systems and SPLs.

9.1 Benefits for the Design Activities of Single Systems

There are various types of development lifecycle for single systems. The more typical ones are the waterfall, incremental, and agile models. *Rational Unified Process* (RUP) [12] is a de facto process framework popularly used in the industry. Since it is inherently incremental and iterative, it can be tailored to support various process models. I hence discuss the benefits of DDM for the design activities of single systems by using RUP as a reference process model.

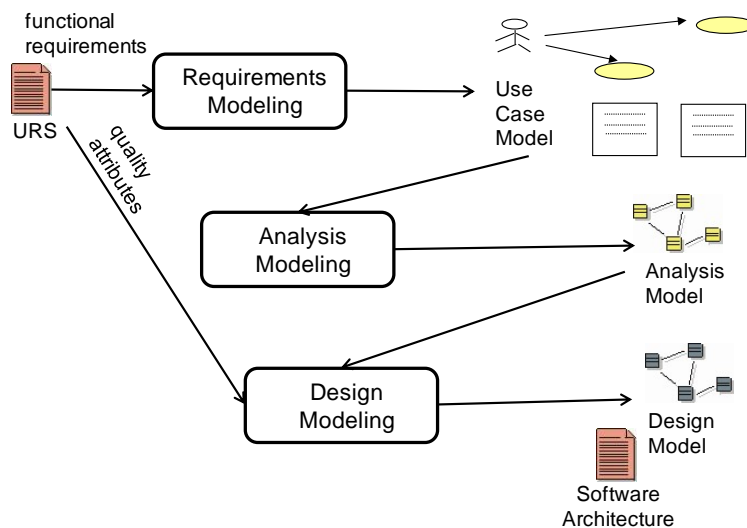


Fig. 28. The Analysis and Design workflow of Rational Unified Process.

Fig. 28 illustrates the *analysis and design workflow* of RUP comprising three modeling stages: requirements modeling, analysis modeling, and design modeling. In *requirements modeling*, the functional requirements are analyzed

and structured as use cases. The dependencies among the functional requirements are identified as relationships between use cases. The operation flows of each use case are described in terms of the interactions between the users and the system. In *analysis modeling*, the analysis objects are identified along with their state (attributes) and responsibilities (operations) without considerations for implementation. In *design modeling*, design strategies (which are captured as a part of the software architecture) are devised according to the operating environment in order to fulfill the quality attributes. The analysis objects are adapted according to the design strategies to become the design objects with full class details. Design issues occur as the analysis objects are adapted, these occurrences of design issues and their related design information can be captured using DDM. So, DDM is a means to formally document design information that are not usually captured in design modeling of RUP.

The first benefit of DDM is that the developers can *revisit the existing design information* for various features to understand the deliberations and rationales behind. In fact, an inexperienced developer can study the design information to learn on design approaches and techniques.

With the explicit dependencies specified between the elements of DDM, the second benefit is that the developers can *systematically evaluate the impact of evolution (addition, removal, and modification) of an element on other elements of DDM* as the design for features changes.

With the explicit trace links from the features through the elements of DDM to the variation points in code, the third benefit is that the developers can *systematically evaluate the impact of evolution of an element of DDM on various features and variation points in code*. Hence, DDM bridges between the features and the variation points in code; *enabling end-to-end traceability* that minimizes unintended errors during evolution, which are common given the complication involved.

9.2 Benefits for the Design Activities of SPLs

In the development of SPLs, domain engineering and application engineering are the key workflows.

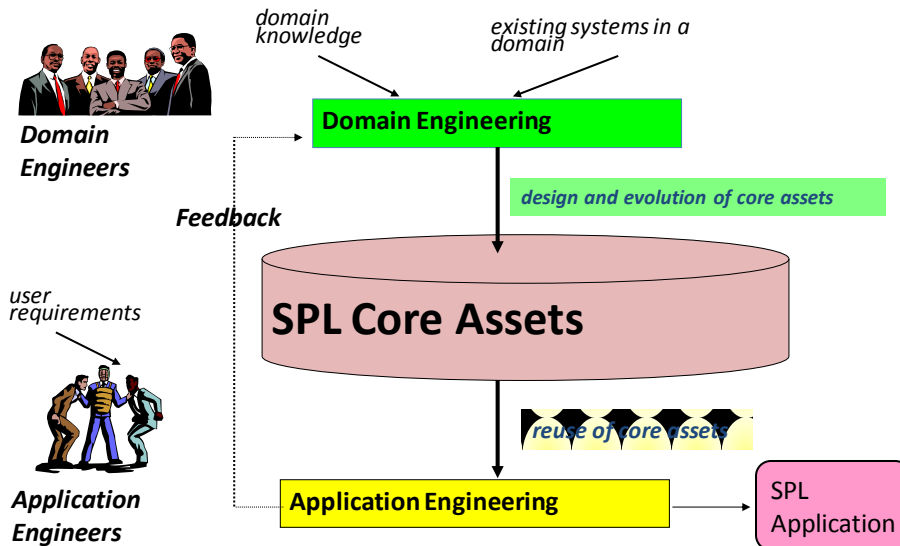


Fig. 29. The Domain Engineering and Application Engineering workflows of the development of SPL.

Fig. 29 illustrates the key workflows of the development of SPL. In *domain engineering*, the domain engineers analyze a few similar existing systems in a domain and construct reusable core assets. The core assets may include any artifacts that can be reused, e.g., requirements specification, software architecture, design specification, code, user documentation, test cases, etc. In this thesis, I focus on code core assets. The code core assets are designed to support the required features of the SPL. They are instrumented with some variability technique so that they can be reused during application engineering.

In *application engineering*, the application engineers analyze the user requirements for an SPL application and construct it by reusing and adapting the code core assets. The assembled code is finally tested against the user requirements of the SPL application. The SPL application may possibly be evolved and enhanced, these changes can be selectively absorbed into the code core assets by the domain engineers.

Similar to the situation of single systems, the design information for an SPL can also be formally documented using DDM that caters to variability in features. It retains *the design knowledge and decisions made for the features* of the SPL that are *beyond what is usually captured by the SPL architecture*. As such knowledge is now formally captured by DDM, my model benefits greatly those activities of SPL that require more granular design information than what component architecture can accommodate. These activities are enumerated below.

The three benefits for the situation of single systems, as discussed in section 9.1, also apply to the situation of SPLs. In domain engineering, these benefits are also extended to cater to the variability in features. For the first benefit, the design information for variant features can emerge or vanish according to feature selection. For the second and third benefits, the evaluation of impact also caters to the inclusion/exclusion of the design information associated with variant features, based on feature selection.

In application engineering, as an SPL application is instantiated from the core assets, a customized instance of DDM that includes only the design information for the selected variant features of the SPL application is instantiated. In fact, this application-specific instance of DDM is equivalent to that of a single system. The application engineers can hence enjoy the same traceability-enabled benefits of single systems, as discussed in section 9.1, in the context of application engineering. They can also *adapt the design information for the application features by evolving this application-specific instance of DDM*. As the fourth benefit, the use of the same representation to capture the design information for an SPL application as well as the SPL itself would *ease the future incorporation of application-specific adaptation back into the SPL*.

Chapter 10 Related Works

Apart from the design at the object level in section 2.1, a single system or an SPL may also specify its *architectural design* in terms of components and interfaces. An *architectural design decision* differs from the design decisions in this thesis in that it impacts components instead of objects. Architectural design is usually a component architecture represented in diagrams (e.g. the UML component diagram) and textual description. It may guide the developers on the design of code that interface with other components. Being coarse-grained, the implication of an architectural design decision is usually not directly traceable to its fine-grained objects.

In general software (i.e., single system) engineering, topics on architectural design decisions and related concepts are researched [4,11,17,18,15]. These works focus on the architectural design (e.g. loose coupling between components) with trace links from features to architectural artifacts. In particular, [4] proposes a metamodel for elements of architectural design decisions and their links to architectural artifacts. There is however no comprehensive and practical enough solutions on traceability of design decisions for design at the object level, going beyond design of component architecture; and current IDEs provide only limited support for traceability.

In SPL engineering, existing works [12,2,16,1,10,5] focus on the traceability between artifacts from various levels of abstraction, primarily features, components, and objects. The role of design decisions at the object level in bridging the problem space (features) and solution space (components and objects) is overlooked. [3] proposes a metamodel for capturing architectural design decisions and trace links from features to architectural description. These various models for architectural design do not address design at the object level, going beyond design of component architecture.

In SPL feature modeling, the dependencies among features further constrain valid feature configurations in addition to variability in features. In FODA [9], two *composition rules* are used to represent *requires* and *mutually exclusive*

dependencies between any two features. In FORM [10], additional composition rules are added to represent *mutual exclusion* and *mutual dependency* among variant features. Lee et al. [12] show that there are *operational dependencies* among features. For instance, a feature may have a *usage* dependency on another feature. They should be identified before designing core assets. Ferber et al. [5] propose a graphical representation for feature dependency which complements feature diagram of FODA. Apart from [5], the above works identify feature dependencies by analyzing features. The identified feature dependencies are *planned* and serve as an input to the design and realization of core assets – a top-down approach. While [5] does not restrict the identification approach, it briefly mentions the use of bottom-up approach where design in core assets is analyzed for feature dependencies. My work on DDM captures the design decisions for features in code. It also captures the dependencies, if any, among the design decisions. In the context of feature dependency, DDM can serve as a bottom-up structured means to identify feature dependencies that arise due to the design decisions.

In short, there is a lack of existing solutions that address both design decisions for design at the object level – beyond component architecture – as well as traceability from features to code in both single system and SPL engineering. My work enhances fine-grained reuse at the object level, beyond coarse-grained reuse at the component level, in the context of both single system and SPL engineering. My work also explicates feature dependencies due to design at the object level and the impact on feature configurations.

Chapter 11 Conclusion

This chapter concludes by summarizing my achievements and recommending future works.

11.1 Achievements

In this thesis, I discuss the decisions involved in the design for features and their importance in the traceability from features to code in both single system and SPL engineering.

I propose an abstract syntax (DDM) to document these decisions and also the trace links from features to code. I formalize DDM in terms of its elements, dependencies among elements, and trace links. I also specify a set of traceability rules for enforcing the integrity of DDM. Detailed logics are specified for the impacts due to the evolution of elements of DDM as design for features changes. In order to apply DDM on the SPL situation, I also devise a set of scoping rules that extends DDM to account for the variability in features.

I describe how formal method can be used to specify and verify the abstract syntax of DDM, the instances of DDM, and the feature configurations of the instances of DDM, and to derive information from instances of DDM. I also devise the schemes to perform formal verification using Alloy, a formal method tool. As a guideline to the tools developers, I suggest how the key salient features of support IDEs adopting DDM can be implemented.

I validate the usage of DDM and its impacts by means of usage examples. I also evaluate the benefits of the proposed DDM in the context of the design activities of both single system and SPL engineering.

11.2 Future Works

I have three major recommendations for future works that extend the works in this thesis.

The first recommendation is to extend my model to capture the aspect of quality attributes and their influence on the selection of design decisions. This aspect would include the derivation of optimal sets of design decisions for a single system or an SPL.

The second recommendation is to implement the abstract syntax in a formal method tool as proposed in Chapter 7. The implementation will serve to verify the abstract syntax so that the latter can be refined accordingly.

The third recommendation is to implement a support IDE that is proposed in Chapter 8. The implementation can be conducted incrementally and be tested against the running example. The complete implementation can then be tested against an industry case study.

Bibliography

1. Anquetil, N., Grammel, B., Galvao Lourenco da Silva, I., Noppen, J.A.R., Shakil Khan, S., Arboleda, H., Rashid, A., Garcia, A.: Traceability for Model Driven, Software Product Line Engineering, <http://purl.org/utwente/64994>, (2008).
2. Berg, K., Bishop, J., Muthig, D.: Tracing software product line variability: from problem to solution space. Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. pp. 182–191. South African Institute for Computer Scientists and Information Technologists, White River, South Africa (2005).
3. Capilla, R., Ali Babar, M.: On the Role of Architectural Design Decisions in Software Product Line Engineering. In: Morrison, R., Balasubramaniam, D., and Falkner, K. (eds.) Software Architecture. pp. 241–255. Springer Berlin Heidelberg, Berlin, Heidelberg.
4. Capilla, R., Zimmermann, O., Zdun, U., Avgeriou, P., Küster, J.M.: An Enhanced Architectural Knowledge Metamodel Linking Architectural Design Decisions to other Artifacts in the Software Engineering Lifecycle. In: Crnkovic, I., Gruhn, V., and Book, M. (eds.) Software Architecture. pp. 303–318. Springer Berlin Heidelberg, Berlin, Heidelberg (2011).
5. Ferber, S. et al.: Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. Proceedings of the Second International Conference on Software Product Lines. pp. 235–256 Springer-Verlag (2002).
6. Gheyi, R., Massoni, T., Borba, P.: A theory for feature models in alloy.
7. Jackson, D.: Software abstractions: logic, language and analysis. MIT Press, Cambridge, Mass (2006).
8. Jarzabek, S., Bassett, P., Zhang, H., Zhang, W.: XVCL: XML-based variant configuration language. Software Engineering, 2003. Proceedings. 25th International Conference on. pp. 810 – 811 (2003).

9. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. DTIC Document (1990).
10. Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*. 5, 143–168 (1998).
11. Kruchten, P.: *Building Up and Reasoning About Architectural Knowledge. Quality of Software Architectures*. Springer Berlin / Heidelberg (2006).
12. Kruchten, P.: *The rational unified process: an introduction*. Addison-Wesley, Boston (2004).
13. Lee, K., Kang, K.C.: Feature Dependency Analysis for Product Line Component Design. In: Bosch, J. and Krueger, C. (eds.) *Software Reuse: Methods, Techniques, and Tools*. pp. 69–85 Springer Berlin Heidelberg, Berlin, Heidelberg (2004).
14. Pashov, I., Riebisch, M.: Using feature modeling for program comprehension and software architecture recovery. *Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the*. pp. 406-417.
15. Ramesh, B., Dhar, V.: Supporting systems development by capturing deliberations during requirements engineering. *IEEE Transactions on Software Engineering*. 18, 498–510 (1992).
16. Sousa, A., Kulesza, U., Rummler, A., Anquetil, N., Moreira, A., Amaral, V., Araújo, J.: *A Model-Driven Traceability Framework to Software Product Line Development*. (2008).
17. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*. 80, 918–934 (2007).
18. Tyree, J., Akerman, A.: *Architecture Decisions: Demystifying Architecture*. *IEEE Software*. 22, 19–27 (2005).

Appendix A Formalization of the Running Example

This appendix formalizes the running example in section 2.2 using the abstract syntax specified in Chapter 3, Chapter 4, and Chapter 5.

A.1 Formalization for Single System

This section formalizes the running example as a single system. The scope includes issues Issue3 through Issue5, alternatives Alternative4 through Alternative6, decisions Decision6 through Decision12, and variation points VP1 through VP7.

Issue occurrences $I = \{i_3, i_4, i_5\}$

$i_3 =$ ("Various rental perks", "Explosion of combinations of rental schemes and rental perks")

$i_5 =$ ("Too many instances of rental perk combinations.", "Each rental scheme is configured with its own instances of rental perk combination.")

Alternatives $A = \{a_4, a_5, a_6\}$

$a_4 =$ ("Decorator design pattern", "Attach additional responsibilities to an object dynamically.", "More flexibility than static inheritance. No explosion of subclasses.", "More object interactions due to chain of decorators.")

$a_5 =$ ("Singleton Design Pattern.", "Ensure a class only has one instance, and provide a global point of access to it.", "Controlled access to sole instance. Can vary number of instances.", "Direct instantiation is not allowed.")

$a_6 =$ ("Subclassing", "Encapsulate each combination of responsibilities in a class.", "Straightforward - one subclass for each combination.", "Explosion of subclasses if there are many combinations.")

Decisions $D = \{d_6, d_7, d_8, d_9, d_{10}, d_{11}, d_{12}\}$

$d_6 =$ ("Decorate rental schemes with rental perks", "Any combination of rental perks can be configured for any rental scheme at runtime.", "Extract algorithms of rental perks from `computeRental()` and encapsulate them in a hierarchy of rental

```

perk child classes. Merge hierarchies of rental schemes and
rental perks.", {vp1})

d7 = ("Extensibility of rental perks", "Decouple other
classes from rental perk child classes.", "Add, modify or
remove rental perk child classes to/from rental perk
hierarchy.", {vp2})

d8 = ("Make rental perks singletons", "Rental perks are not
specific to any rental scheme.", "Apply Singleton pattern to
RentalPerk. Add getInstance() that instantiates and shares
instances of child classes.", {vp3})

d9 = ("Resolve conflict between Decision8 and Decision6.",
"Rental perk child classes have public constructors while
Singleton constructors should be protected or private. Cannot
initialize a RentalPerk instance with a RentalComp instance
via constructor.", "Make constructors of rental perk child
classes protected. Add setRentalStrategy() to initialize a
RentalPerk instance with a RentalStrategy instance.", {vp4})

d10 = ("Represent combinations of rental perks using
subclasses.", "Create a hierarchy of subclasses to represent
required combinations. Acceptable for small number of
combinations.", "Use one subclass for each combination of
rental perks.", {vp5})

d11 = ("Extensibility of rental perk combinations.",
"Decouple other classes from rental perk combination child
classes.", "Add, modify or remove rental perk combination
child classes to/from rental perk combination hierarchy.",
{vp6})

d12 = ("Share instances of rental perk combinations.",
"Rental perk combinations are not specific to any rental
scheme.", "Apply Singleton pattern to RentalPerkComb. Add
getInstance() that instantiates and shares instances of child
classes.", {vp7})

```

Variation Points VP = {vp₁, vp₂, vp₃, vp₄, vp₅, vp₆, vp₇}

```

vp1 = ("VP1", ())
vp2 = ("VP2", ())
vp3 = ("VP3", ())
vp4 = ("VP4", ())
vp5 = ("VP5", ())
vp6 = ("VP6", ())
vp7 = ("VP7", ())

```

Issue occurrence-alternative associations IA = {(i₃, a₄), (i₃, a₆), (i₄, a₅), (i₅, a₅)}

Issue occurrence-decision associations $ID = \{(i_3, d_6), (i_3, d_{10}), (i_4, d_8), (i_5, d_{12})\}$

selected(d_6)

selected(d_8)

selected(d_{12})

Decision-alternative associations $DA = \{(d_6, a_4), (d_8, a_5), (d_{10}, a_6), (d_{12}, a_5)\}$

Comprise associations $DD_{comprise} = \{(d_6, d_7), (d_{10}, d_{11})\}$

Constrain associations $DI_{constrain} = \{(d_6, i_4), (d_{10}, i_5)\}$

Forbid associations $DD_{forbid} = \{(d_8, d_6)\}$

Resolve associations $DD_{resolve} = \{(d_9, d_8), (d_9, d_6)\}$

Features $F = \{f_5, f_6, f_7, f_8, f_9, f_{10}\};$

$f_5 =$ ("Rental Perk")

$f_6 =$ ("Xmas Promotion")

$f_7 =$ ("CNY Promotion")

$f_8 =$ ("Great Singapore Sale")

$f_9 =$ ("Discount Voucher")

$f_{10} =$ ("Loyalty Program")

Feature-issue occurrence traces $FI = \{(f_5, i_3), (f_5, i_4), (f_5, i_5), (f_6, i_3), (f_7, i_3), (f_8, i_3), (f_9, i_3), (f_{10}, i_3)\}$

A.2 Formalization for SPL

This section extends the formalization in section A.1 for a single system. It assumes that all the variant features are selected for an SPL application.

in_scope(f_5)

in_scope(f_6)

in_scope(f_7)

in_scope(f_8)

in_scope(f_9)

in_scope(f_{10})

By applying the scoping rules specified in Chapter 5, the elements of DDM that are in scope can be derived.

Appendix B Source Code of the Running Example

This appendix lists the source code in Java programming language that are referred to by the running example in section 2.2. These classes are RentalStrategy.java, RentalPerk.java, XmasPromo.java, CNYPromo.java, GSSPromo.java, DiscountVoucher.java, and LoyaltyProgram.java.

RentalStrategy.java

```
package crs;

public abstract class RentalStrategy {
    public abstract float computeRental(
        Customer c, Vehicle v, int days, float undiscounted);
}
```

RentalPerk.java

```
package crs;

public abstract class RentalPerk extends RentalStrategy {
    public enum Type {XmasPromo, CNYPromo, GSSPromo,
        LoyaltyProgram, DiscountVoucher};
    private RentalStrategy strategy;
    private static RentalPerk[] p =
        new RentalPerk[Type.values().length];

    public static RentalPerk getInstance(Type t) {
        if (p[t.ordinal()] == null) {
            if (t == RentalPerk.Type.XmasPromo)
                p[t.ordinal()] = new XmasPromo();
            else if (t == RentalPerk.Type.CNYPromo)
                p[t.ordinal()] = new CNYPromo();
            else if (t == RentalPerk.Type.GSSPromo)
                p[t.ordinal()] = new GSSPromo();
            else if (t == RentalPerk.Type.LoyaltyProgram)
                p[t.ordinal()] = new LoyaltyProgram();
        }
    }
}
```

```

        else if (t == RentalPerk.Type.DiscountVoucher)
            p[t.ordinal()] = new DiscountVoucher();
        else
            return null;
    }
    return p[t.ordinal()];
}

public void setRentalStrategy(RentalStrategy s) {
    this.strategy = s;
}

public RentalStrategy getRentalStrategy() {
    return strategy;
}
}

```

XmasPromo.java

```

package crs;

public class XmasPromo extends RentalPerk {

    @Override
    public float computeRental(
        Customer c, Vehicle v,
        int days, float undiscounted) {
        // 20% off total charge.
        float discounted = 0.80f * undiscounted;
        System.out.println(
            "XmasPromo discounted amount is $" + discounted);

        float prevPrice = getRentalStrategy().computeRental(
            c, v, days, undiscounted);

        return (discounted < prevPrice)
            ? discounted : prevPrice;
    }
}

```

```
}  
}
```

CNYPromo.java

```
package crs;  
  
public class CNYPromo extends RentalPerk {  
  
    @Override  
    public float computeRental(  
        Customer c, Vehicle v,  
        int days, float undiscounted) {  
        // 30% off total charge.  
        float discounted = 0.70f * undiscounted;  
        System.out.println(  
            "CNYPromo discounted amount is $" + discounted);  
  
        float prevPrice = getRentalStrategy().computeRental(  
            c, v, days, undiscounted);  
  
        return (discounted < prevPrice)  
            ? discounted : prevPrice;  
    }  
}
```

GSSPromo.java

```
package crs;  
  
public class GSSPromo extends RentalPerk {  
  
    @Override  
    public float computeRental(  
        Customer c, Vehicle v,  
        int days, float undiscounted) {  
        // 1 day free for every 2 days.  
        float discounted = (float)(days - (days / 3))
```

```

        / (float)days * undiscounted;
System.out.println(
    "GSSPromo discounted amount is $" + discounted);

float prevPrice = getRentalStrategy().computeRental(
    c, v, days, undiscounted);

return (discounted < prevPrice)
    ? discounted : prevPrice;
}
}

```

DiscountVoucher.java

```

package crs;

public class DiscountVoucher extends RentalPerk {
    private float value = 0f;

    @Override
    public float computeRental(
        Customer c, Vehicle v,
        int days, float undiscounted) {
        // Offset previous price with value of voucher.
        float prevPrice = getRentalStrategy().computeRental(
            c, v, days, undiscounted);
        float discounted;
        if (prevPrice > getValue()) {
            discounted = prevPrice - getValue();
        }
        else {
            discounted = 0;
        }
        System.out.println("DiscountVoucher discounted amount
            is $" + discounted);

        return discounted;
    }
}

```



```

    }

    public float getValue() {
        return value;
    }

    public void setValue(float value) {
        this.value = value;
    }
}

```

LoyaltyProgram.java

```

package crs;

public class LoyaltyProgram extends RentalPerk {

    @Override
    public float computeRental(
        Customer c, Vehicle v,
        int days, float undiscounted) {
        // Offset discounted amount with loyalty points.
        float prevPrice = getRentalStrategy().computeRental(
            c, v, days, undiscounted);
        float discounted;
        if (prevPrice > c.getLoyaltyPoints()) {
            discounted = prevPrice - c.getLoyaltyPoints();
            c.setLoyaltyPoints(0);
        }
        else {
            discounted = prevPrice - (int)prevPrice;
            c.setLoyaltyPoints(
                c.getLoyaltyPoints() - (int)prevPrice);
        }
        System.out.println("LoyaltyProgram discounted amount
            is $" + discounted);
        System.out.println("Balance loyalty point amount is " +

```

```
        c.getLoyaltyPoints());  
  
    return discounted;  
    }  
}
```