A FOUNDATION FOR BUILDING STABLE ANALYSIS PATTERNS

by

Haitham Safwat Hamza

A THESIS

Presented to the Faculty of

The Graduate Collage at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Mohamed Fayad

Lincoln, Nebraska

August. 2002

A FOUNDATION FOR BUILDING STABLE ANALYSIS PATTERNS

Haitham Safwat Hamza, M.S.

University of Nebraska, 2002

Advisor: Mohamed Fayad

Software analysis patterns are believed to play a major role in improving the quality of software products, while simultaneously reducing cost and condensing lifecycles. However, there are still problems that need to be solved in contemporary analysis patterns, such as the lack of stability, the lack of proper abstraction levels, and inadequate documentation. These problems tremendously diminish the reusability and the effectiveness of analysis patterns.

In this thesis, the main problems facing analysis patterns are investigated. As a solution to these problems, the following remedies are proposed: (1) A new concept, *stable analysis patterns,* to accomplish stability and to achieve the proper abstraction level. (2) A new documentation template, to improve understanding and to enhance communicating patterns among software developers. (3) A pattern language contains eight patterns, to document the process of building stable analysis patterns. (4) Five stable analysis patterns, to demonstrate the usage of the proposed approach. (5) Eight essential properties that cover the main qualities of any analysis pattern, to evaluate the samples of both today's analysis patterns and the proposed stable analysis patterns. Stable analysis patterns demonstrate effectiveness by satisfying all of the proposed essential properties.

Acknowledgement

I am enormously grateful to my advisor Dr. Mohamed Fayad, for his generous support during the work of this thesis. His wholehearted feedback and encouragement have been always the motivation for completing this work. He taught me that patience and optimism are the keys for success. I also thank him for the many fertile discussions and suggestion that built this work from ground zero.

I would like to thank Dr. Jitender Deogun and Dr. Peter Revesz for taking the time to serve on my defense committee.

Many thanks to my friends at University of Nebraska-Lincoln for taking the time to read and comment on my work. Special thanks to my best friend Ahmed Mahdy for his support and his willingness to always listen and discuss this work. I would like also to thank David Naney for his effort in copyediting this thesis.

Finally, I most want to thank my lovely parents for their continuous support. Without their love and care I have never got the energy for continuing this work. I am indebted to my perfect brother and my best friend Mohamed, his care and guidance have helped me to overcome many tough times. Special thanks to my brothers: Kamal, Faisal, Abd Elbaset and Abd Elrahman, for their support.

Contents

1 Introduction……………………………………………………. 1

1.1 Software Patterns: Background and Definition…………………………………1

1.2 Software Analysis Patterns……………………………………………………...3

1.3 General Classification of Analysis Patterns……………………………………. 4

1.4 Research Objective ……………………………………………………………..5

1.5 Thesis Outline…………………………………………………………………...6

2 Challenges Facing Today's Analysis Patterns……………… 7

2.1 Introduction……………………………………………………………………….. 7

2.2 Classification of Today's Analysis Patterns……………………………………7

2.3 The Main Challenges Facing Analysis Patterns and Their Current Solutions…………………………………………………………………….. 9

2.3.1 Stability Problem……………………………………………………… 9

2.3.2 Abstraction Level Problem……………………………………………….10

2.3.3 Documentation Problem…………………………………………………12

3 Software Stability Concepts…………………………………14

3.1 Introduction……………………………………………………………………..14

3.2 Software Stability Related Work………………………………………………….14

3.3 Software Stability Paradigm……………………………………………………15

3.3.1 Enduring Business Themes (EBTs)……………………………………16
3.3.2 Business Objects (BOs)……………………………………………...17
3.3.3 Industrial Objects (IOs)……………………………………………...17
3.4 Identification Heuristics……………………………………………………18
3.4.1 Top-Down Identification Approach……………………………………18
3.4.2 Bottom-Up Identification Approach……………………………………19
3.5 Example of Applying Software Stability Concepts……………………………..19
3.5.1 Case I- Computer Trading…………………………………………..20
3.5.2 Case II- Buying a House…………………………………………….21
3.5.3 Bidding on a Football Team…………………………………………22
3.5.4 Compound System…………………………………………………..23
4 Stable Analysis Patterns………………………………….25
4.1 Introduction………………………………………………………………..25
4.2 Stable Analysis Patterns Concepts…………………………………………26
4.3 Stable Analysis Patterns Template………………………..…………………26
4.4 Stable Analysis Patterns Example…………………………………………28
4.4.1 Pattern Name…………………………………………………………28
4.4.2 Context………………………………………………………………28
4.4.3 Problem………………………………………………………………28
4.4.4 Forces………………………………………………………………..28
4.4.5 Pattern Structure and Participants……………………………………29
4.4.6 CRC- Cards………………………………………………………….32

4.4.7 Applicability……………………………………………………………………33

4.4.7.1 Case Study I: Negotiation in buying a car…………………………………34

4.4.7.2 Case Study 2: Content Negotiation using Composite Capability/ Preference Profile (CC/PP)…………………………………………………...51

4.4.8 Related Patterns…………………………………………………………………...69

4.4.9 Possible Extensions………………………………………………………………69

5 A Pattern Language For Building Stable Analysis Patterns……………………………………………...71

5.1 Introduction……………………………………………………………………………71

5.2 Pattern Language Overview…………………………………………………………72

5.3 Building Stable Analysis Pattern Language Description…………………………76

5.3.1 Pattern 1.1- Efficient Usable Analysis Model………………………………76

5.3.2 Pattern 1.2- Software Stability Model………………………………………77

5.3.3 Pattern 2.3- Identify The Problem……………………………………………..82

5.3.4 Pattern 2.4- Identify Enduring Business Themes …………………………86

5.3.5 Pattern 2.5- Identify Business Objects………………………………………91

5.3.6 Pattern 2.6- Proper Abstraction Level………………………………………98

5.3.7 Pattern 3.7- Build Stable Analysis Patterns………………………………...104

5.3.8 Pattern 3.8- Use Stable Analysis Patterns………………………………… 108

6 Evaluation of Stable Analysis Patterns……………………...113

6.1 Introduction……………………………………………………………………………113

6.2 Essential Properties of Analysis Patterns…………………………………………113

6.3 Experience Group Evaluation……………………………………………………….116

6.4 Analogy Group Evaluation………………………………………………………….119

6.5 Stable Analysis Pattern Group Evaluation………………………………………..123

6.6 Conclusions………………………………………………………………………….127

7 Conclusions and Future Work……………………………….128

References……………………………………………………131

List of Figures

1.1 Analysis patterns in the development cycle………………………………….. 3

3.1 Software Stability Model (SSM) architecture……………………………….. 16

3.2 Computers trading stable model…………………………………………… 21

3.3 House trading stable model………………………………………………... 22

3.4 Bidding on a football team stable model…………………………………... 23

3.5 Combined stable models…………………………………………………… 24

4.1 AnyNegotiation pattern object diagram…………………………………… 30

4.2 Party package………………………………………………………………. 31

4.3 Context package……………………………………………………………. 32

4.4 Media package……………………………………………………………... 32

4.5 Negotiation system object diagram………………………………………... 34

4.6 Stability model for the negotiation in buying a car case study………………. 35

4.7 Use case diagram for the negotiation in buying a car case study……………. 36

4.8 Place a Bid sequence diagram……………………………………………... 47

4.9 Negotiate Car Price sequence diagram…………………………………….. 48

4.10 Prepare Contract sequence diagram……………………………………….. 49

4.11 Sign Contract sequence diagram…………………………………………... 49

4.12 Negotiate Price state transition diagram…………………………………… 50

4.13 Sign Contract state transition diagram……………………………………….. 50

4.14 Possible scenario of content negotiation using CC/PP………………………. 51

4.15 The stability model of the content negotiation case study…………………… 52

4.16 Use Case diagram for content negotiation case study……………………….. 53

4.17 Send HTTPRequest sequence diagram………………………………………. 65

4.18 Retrieve Profile sequence diagram…………………………………………… 65

4.19 Adapt Content sequence diagram…………………………………………….. 66

4.20 Negotiate Content sequence diagram………………………………………… 67

4.21 Retrieve Profile state transition diagram……………………………………... 68

2.22 Negotiate Content state transition diagram…………………………………… 68

4.24 Negotiation system with NegotiationStyle package………………………….. 70

4.25 NegotiationStyle package class diagram……………………………………... 70

5.1 Description of the pattern language …………………………………………….. 73

5.2 The pattern language chart………………………………………………………. 75

5.3 The relation between SSM elements…………………………………………. 79

5.4 Example of the mapping between EBTs, BOs, and IOs……………………… 80

5.5 *Account* pattern provided by Martin…………………………………………….. 84

5.6 Examples of accounts without entries………………………………………... 85

5.7 Example of entries without accounts…………………………………………. 85

5.8 The steps for identifying the EBTs of the problem…………………………... 88

5.9 The steps for identifying the BOs of the problem……………………………. 94

5.10 Proper abstraction level……………………………………………………. 101

5.11 The relationship between the EBTs and the BOs of the problem…………….. 105

5.12 *AnyAccount* pattern class diagram……………………………………………. 107

5.13 *AnyEntry* pattern class diagram………………………………………………. 108

5.14 Copy Machine account object diagram………………………………………. 110

5.15 Hotmail Account object diagram……………………………………………... 112

6.1 Account pattern provided by Martin………………………………………….. 117

6.2 Resource rental pattern…………………………………………………….. 119

6.3 Instantiation of resource rental pattern for a library service………………….. 120

6.4 AnyAccount pattern class diagram…………………………………………… 124

6.5 AnyEntry pattern class diagram………………………………………………. 125

6.6 AccountWithEntry pattern……………………………………………………. 125

List of Tables

5.1 Summary of the pattern language…………………………………………….. 74

6.1 Experience group evaluation summary……………………………………….. 118

6.2 Analogy group evaluation summary………………………………………….. 122

6.3 Stable analysis pattern group evaluation summary…………………………… 126

7.1 Quick reference to the proposed five stable analysis patterns………………... 129

Chapter 1

Introduction

1.1 Software Patterns: Background and Definition

Software development is a costly and a time consuming process, yet, the quality of the ultimate product is not guaranteed. Recently, software engineers have realized that a considerable portion of time and cost are spent in solving problems that are similar or identical to problems they have solved in earlier projects. These facts have lead the software engineering community to devote a significant effort to capture solutions to recurring problems in all levels of software development. Consequently, the notation of software patterns has emerged.

The popularity of software patterns is ascribed mainly to the wide acceptance of the GoF book [10], which introduces many patterns as solutions to different recurring software design problems. Thereafter, many software patterns have came forward to cover the different levels of software development phases.

Though the concept of patterns is ubiquitous, many different software engineering definitions exist for the word "Pattern". In [3], the architect, Christopher Alexander, has defined "Pattern" as follows:

"Each pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Another definition of the word "Pattern" is given by Gamma et al in [10]:

"A pattern presents the essence of the solution for a recurring problem in a specific context"

In [22], Fowler has defined patterns as:

"An idea that has been useful in one practical context and will probably be useful in others"

The ultimate goal of patterns is to help to assuage software complexity at the different phases in the software life cycle [5]. As a result, different types of software patterns have emerged. One possible classification of software patterns is on the development phase in which they might be used. This classification method classifies patterns into different categorizes: analysis patterns, design patterns, implementation patterns, and testing patterns, etc. Another classification is the one given in [11], which classifies patterns into three main categories: architectural patterns, design patterns, and idioms. However, this classification does not consider the new emerging patterns types such as analysis patterns and testing patterns.

1.2 Software Analysis Patterns

Analysis patterns are conceptual models that model the knowledge of the problem domain. They help to understand the problem itself rather than to show how to design the solution. By having conceptual models that model recurring problems, analysis can be made faster, more accurate, and more cost effective than it is today. Figure 1.1 shows the use of analysis patterns in the overall development of software systems.

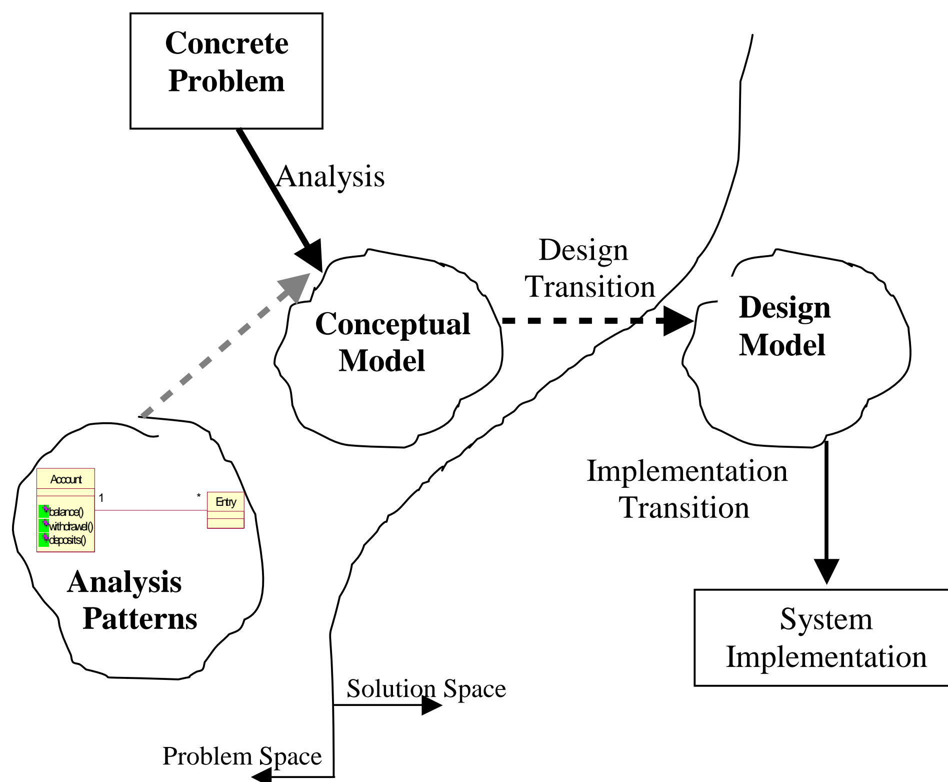


Figure 1.1: Analysis patterns in the development cycle.

Even though software developers are aware of the tremendous importance of the analysis phase [5,6,7,8], limited work has been done in developing analysis patterns, compared to the work that has been done in developing design patterns. The work done by Martin

[22], and the analysis patterns and languages presented in the procedures of the Pattern Languages of Programs (PLoP) are considered to be the only existing works that focus on analysis patterns.

1.3 General Classifications of Analysis Patterns

The product of an analysis pattern is a conceptual model that describes the problem under consideration. The level of abstraction determines how broadly the pattern can be used. However, too much abstraction can be a negative. There is a trade off between the flexibility and reusability of the model on one hand, and complexity of understanding and reuse of the model on the other. The level of abstraction plays a major role in this equation. A low level of abstraction will reduce the reusability of the model; adapting it will no longer be an easy task, but the resulting model will be relatively simple when compared to the same model designed with a higher level of abstraction. Conversely, too much abstraction will lead to models that are too complex. Therefore, the optimum solution is a conceptual model that is simple enough be easily used and understood, but with a level of abstraction high enough to make it general and hence more reusable.

One classification of analysis patterns is based on their applicability. Generally, analysis patterns can be divided into two types: domain specific analysis patterns and cross-domain (domain-less) analysis patterns.

Domain specific patterns are patterns that model problems that appear in a specific context. These problems should not occur outside that context; the generality requirement

of the pattern will not hold. Cross-domain (domain-less) patterns are patterns that can be used in any context.

1.4 Research Objectives and Thesis Contributions

The following table summarizes the objectives of our research, and it shows the proposed approach and our contributions to accomplish each objective.

Question to Answer	Proposed Solution, Approach and Contributions	Chapter
What are the main challenges facing today's analysis patterns?	Study the main challenges that face today's analysis patterns, and show their current solutions.	2
How can we achieve stability features while building software analysis patterns?	Propose the novel concept of *Stable Analysis Patterns*.	4
How can we achieve the proper level of abstraction while building analysis patterns?	Use software stability concepts.	4
How can we better describe and document analysis patterns in order to improve their reusability?	Propose a novel documentation template that covers the main aspects and issues of analysis patterns.	4
How can stable analysis patterns be used?	Propose 13 domain-less stable analysis patterns.	4, 5, and 6.
What are the main steps required for building stable analysis patterns?	Develop a pattern language for building stable analysis patterns.	5

1.5 Thesis Outline

The thesis is organized as follows:

In chapter 2, the main challenges facing today's analysis patterns, the consequences of these challenges, and the current solutions for these problems will be discussed in detail. In chapter 3, the related work in software stability, and the basic concepts of the software stability model (SSM) will be discussed. In chapter 4, the novel concept of stable analysis patterns, the novel documentation template, and an example of stable analysis patterns are presented in detail. Chapter 5 will introduce the pattern language that documents the steps in building stable analysis patterns. The proposed eight essential properties and the evaluation of today's analysis patterns and the proposed stable analysis patterns will be discussed in chapter 6.

Chapter 2

Challenges Facing Today's Analysis Patterns

2.1 Introduction

Software analysis patterns are believed to contribute vastly in improving the quality of software products, while simultaneously reducing their development cost. However, analysis patterns have not realized their full potential. Today's analysis patterns are insufficiently mature to be effectively utilized. Understanding the causes of this immaturity is the first step in achieving efficient and reusable analysis patterns.

In this chapter, the classifications of today's analysis patterns are given. The main challenges facing today's analysis patterns and their current solutions are discussed in detail.

2.2 Classification of Today's Analysis Patterns

One possible classification for today's analysis patterns is based on the approach of construction [12, 13]. Generally, different building approaches categorize analysis patterns into three main groups:

Group I People in this group use their experience to build analysis patterns. Simply, patterns are produced during the course of specific projects. Since no one can be an expert in all fields, domain experts often produce domain specific patterns, even if the problem modeled occurs in many other contexts.

Group II People in this group use to build their analysis patterns based on analogy. According to this group, patterns that model complete systems in one context are reused by making an analogy between the pattern and the new application. Thus, by analogy, they change the names of the pattern's classes to be relevant to the new application. Even though this group believes that analysis patterns should be built in a way the makes them reused to model the same problem regardless of its context. However, the way they choose to approach this goal makes them end up building templates rather than building patterns.

Group III That is our group, our approach is based on the software stability concepts [19]. By analyzing the problem in terms of its EBTs and the BOs, the resultant pattern models the core knowledge of the problem. The goal of this approach is stability. As a result, these stable patterns could be used to model the same problem regardless its context.

This classification will be used in the evaluation of analysis patterns in Chapter 6.

2.3 The Main Challenges Facing Analysis Patterns and Their Current Solutions

There are three main factors that control the reusability and the effectiveness of analysis patterns: The stability of the pattern, its abstraction level, and its description (documentation). Based on these three factors, the main challenges that face today's analysis patterns can be summarized in the following three problems:

2.3.1 Stability Problem

Analysis patterns model the core knowledge of the problem. Since the core knowledge of any problem is independent of the problem, context patterns are expected to be highly stable. Thus, a pattern that models a specific problem should be easily reused to model that same problem regardless of its context. Developing patterns that lack stability features will result in multiple analysis patterns for the same problem, a situation that contradicts the objective behind analysis patterns.

One of the major causes of analysis pattern instability is the common problem of mixing analysis and design while developing software systems. Software analysts believe that analysis models should model the problem knowledge and have nothing to do with the design or the implementation issues of the system. However, in practice, the analysis of the problem is usually made with some design and implementation issues in mind. The resultant pattern will adhere to the specific design, forcing the developers to remodel the problem whenever they change their design approaches. Therefore, separating analysis models from design and implementation issues is one of the keys for accomplishing

pattern stability and generality. However, today, there is no common approach that shows analysts how to do so.

2.3.2 Abstraction Level Problem

The level of abstraction determines how broadly the pattern can be used. There is a trade off between the flexibility and the reusability of the pattern on one hand, and the complexity of understanding and the reuse of that pattern on the other hand. The optimum solution is to have patterns that are easy to use and to understand, while at the same time having a sufficient level of abstraction to make them general and hence reusable.

A high abstraction level will result in complex models that are unattractive for reuse. On the other hand, a low abstraction level will result in building domain specific patterns for domain-less problems contradicting the concepts of analysis patterns.

Several different approaches currently exist to handle the abstraction problem. Some of these approaches are summarized below:

- People who extract analysis patterns strictly from experience believe that it is unsafe to further abstract patterns generated within certain projects to make them reusable in other contexts [22]. They argue that the patterns resulted from extended debate and that the patterns have been tested and validated only in the current project. Therefore, according to them, there is no guarantee that these patterns can be successfully reused in other contexts. Consequently, in their view, each developer should carry the full responsibility in judging whether or not it is

appropriate to reuse the pattern. This approach can diminish the reusability of the pattern for many reasons. For instance, time restraints in the development of the new product might limit the amount of time available for the developer to read, understand, and decide on existing patterns as suitable models for his problem. Another possible situation is that the resultant pattern from specific projects usually contains domain specific constraints that tie the problem to a specific domain. Realizing these constraints and adapting the pattern to fit into the new domain might not be a more attractive solution than remodeling the problem from scratch.

- Another approach for handling the abstraction level is to take the experience one step further than previous attempts. People using this approach believe that analysis models resulting from projects are valuable, yet they might be useless if they are not presented in a way that makes the use of the pattern in different contexts possible. The patterns are rewritten with generality in mind, trying to resolve all domains specific constraints in the pattern. Even though the concept sounds good, the problem with this approach is that they usually model systems as a whole, not specific problems. For instance, one might find a pattern that models the rental problem. The pattern is at an appropriate level of abstraction, making it reusable for any rental property, however, these patterns are not complete because they can never catch all the situations that might need to be modeled in the renting of some other properties. This approach of accomplishing abstraction levels results in the building of templates instead of the building of patterns.

2.3.3 Documentation Problem

Analysis patterns are conceptual models. Conceptual models are difficult to understand by their very nature. Therefore, efficient descriptions of such models are greatly desired. Unlike design patterns, analysis patterns have no standard forms that can be used to document them. In addition, the difference between the nature of the analysis and the design makes the usage of the existing known design patterns forms insufficient. Insufficient pattern description jeopardizes the understanding, and hence the reuse of the pattern.

Many different approaches are currently used for analysis patterns documentation. These approaches can be grouped into three main approaches:

1. *Narrative Approach.* In this approach, patterns are not documented in a defined form. Instead, they are described in a straight, narrative style. Patterns presented by Martin in [22] are good examples of this approach. The main problem with this approach is that it does not show all the aspects of the pattern. In addition, it is difficult to understand, and to use.
2. *Design Patterns Forms Approach.* According to this approach, patterns are documented using most, and sometimes all of the elements defined in the known design patterns forms, such as the forms proposed in [10], and [11]. Patterns presented in [27] and [9] are typical examples of this approach. The main problems with this approach are summarized below:
   - Some of the elements are not relevant for describing analysis patterns. For instance, using the element "solution" to present the pattern's structure can

cause confusion between the analysis (as the problem domain), and the design (as the solution space).

- Elements such as: “implementation”, and “Sample Code” are irrelevant when we talk about analysis patterns.
- Design patterns forms are insufficient for describing conceptual models. Conceptual models need more illustration in order to be understood.

3. *“Build-Mine” Approach.* In this approach, patterns are documented in forms that are introduced by the pattern’s author. These forms might be a mixture of elements defined in other forms. However, they always contain some new elements introduced by the author. Patterns presented in [1] are good examples for this approach. In [1], the only new field is the design field, which is very important in documenting analysis patterns. However, this form uses the element “solution”, which is irrelevant for analysis patterns and, it is not sufficient for capturing all the pattern’s aspects.

Chapter 3

Software Stability Concepts

3.1 Introduction

Achieving stability while building software system will tremendously improve the quality and reduce the cost of developing software products. Even though for many years software stability has been the focus of a large number of groups in the software engineering community there still exists no mature methodology that can be standardized for achieving software stability.

In this chapter, the related work in software stability, the concept of the Software Stability Model (SSM), and examples of using these concepts will be introduced.

3.2 Software Stability Related Work

Software stability has been the focus of many researchers among the software community [7,15,26]. These works differ in their definition of and approaches used to accomplish stability.

In [26], the relation between stability and software design is investigated. This work defines stability as a measure of the difficulty in changing a module, not a measure of the possibility that a module will change. It hinges the stability of the system to the dependency between the different design modules in the system. In addition, it proposes several metrics for measuring the stability of the system, based on this dependency concept.

Another approach for accomplishing stability is the work done by Coplien [15]. This work introduces the concept of *commonality and variability* in software engineering. One of the main goals of this approach is to create a design that contributes to reuse and ease of change. The main idea behind this approach is to show how the identification of the common and variable characteristics of the different versions of a system can vastly contribute to reducing the time required to develop high quality software systems.

3.3 Software Stability Paradigm

The stability approach used in this thesis is the Software Stability Model (SSM) introduced in [19]. Figure 1 shows the architecture of the SSM.

In the SSM, the model of the system is viewed as three layers: the Enduring Business Themes (EBTs) layer, the Business Objects (BOs) layer, and the Industrial Objects (IOs) layer. Each class in the system is classified into one of these three layers, according to its nature.

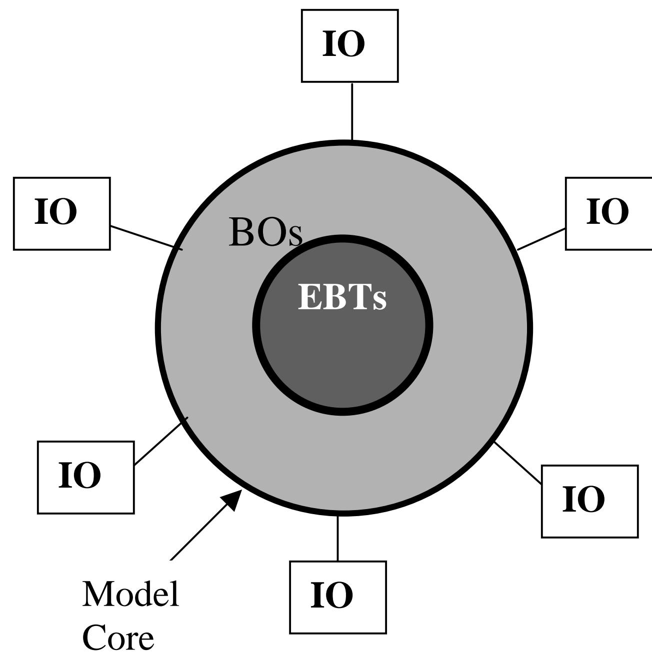


Figure 3.1: Software Stability Model (SSM) Architecture.

3.3.1 Enduring Business Themes (EBTs)

EBTs are the classes that present the enduring and basic concepts of the system. Therefore, they are extremely stable and form the nucleus of the SSM. There are a few properties that characterize EBTs [17]. These properties are:

- *Externally and Internally Stable.* EBTs present the core knowledge of the system. Consequently, they are stable over time and should never change, externally or internally.
- *Conceptual Objects.* EBTs represent concepts; they are not tangible.
- *Commonality to the Domain.* EBTs are common to all of the applications of the domain that they have been identified in.

3.3.2 Business Objects (BOs)

BOs are the classes that map the EBTs of the system into more concrete objects. BOs are tangible and externally stable, but they are internally adaptable. For instance, human beings are BOs. They are externally stable, but they can change internally (humans can get married, or become ill). There are some specific properties that characterize BOs [17]. These properties are:

- *Externally Stable and Internally Adaptable.*
- *Semi-tangible Objects.* Compared to the EBTs, the BOs of the system are tangible, however, they are not concrete objects, therefore, they are considered semi-tangible objects.
- *Commonality to the Domain.* BOs are common to all of the applications of the domain that they have been identified in.

3.3.3 Industrial Objects (IOs)

IOs are the classes that map the BOs of the system into physical objects. For instance, the BO "Agreement" can be mapped in real life as a physical "Contract", an IO. There are a few properties that characterize IOs [17]. These properties are:

- *Unstable.* Since IOs present the physical objects of the system, they can be replaced, added, or even removed from the system without affecting the core of the system.

- *Concrete Objects.* The IOs are fully tangible objects. They usually represent real world objects.
- *Commonality to the Domain.* IOs form the periphery of the SSM model. IOs can be completely different from one application to another, even within the same domain.

3.4 Identification Heuristics

Even thought the concepts of software stability have shown to be promising in achieving stable software systems [21], identifying the EBTs and the BOs of a problem is not a straightforward process. Moreover, long experience within the field does not guarantee accurate EBT and BO definitions [18].

Consequentially, a set of heuristics has been recently defined to facilitate the task of using the SSM concepts [18]. Currently, there are two identification heuristics used for identifying the EBTs and the BOs of a problem: the Top-Down Identification approach, and the Bottom-Up identification approach. The two approaches are summarized below:

3.4.1 Top-Down Identification Approach

In this approach, the analyst starts by identifying the IOs of the system, which are usually easy to find, and then abstracts the system from the conceptual level down to the physical level.

Beginning with the overall system, the concepts involved in this system are developed. This should result in the naming of several subunits of the original system. Each subunit

is then broken down, using the same approach. Once the physical objects making up the system (IOs) are reached, stop and move back one level. The result is a collection of "Objects" that form good candidates for the BOs and the EBTs of the system.

3.4.2 Bottom-Up Identification Approach

In this approach, the classical model of the problem is built without considering stability concepts. The IOs of the model are identified. Next, related IOs are grouped and the common concepts that relate them are defined. Grouping should continue until no further concepts can be found. The result is a collection of "Objects" (concepts) that form the EBTs and the BOs of the system. The final step is to separate the EBTs and the BOs by examining each object against the main characteristics of EBTs and BOs.

3.5 Example of Applying Software Stability Concepts

Several case studies have been performed recently, in order to illustrate the use of SSM concepts [2,21, 24]. This section shows how software stability concepts can be used to model real problems. Three case studies, originally presented in [2], are shown. At first glance, the three case studies presented here appear to be unrelated. However, they all share common objects. Using the software stability concepts of EBTs and BOs will help to extract these commonalities as a core for modeling the three applications.

3.5.1 Case I- Computer Trading

The requirement of this case study is to model the concept of computer trading. The stable model for this application is shown in Figure 3.2. The core of the model consists of four EBTs (Trading, Negotiation, Inspection, and Bidding), and four BOs (Bid, Agreement, Features, and User). It is clear that none of the EBTs or BOs are limited to the current application, computer trading. For instance, "Trading" is an object that remains stable externally and internally; it is not tied to computer trading or any other specific application. Whenever we need to trade *anything* this object should be there. The same concept applies to the other three EBTs. Likewise, looking at the BOs of this model, there is nothing specific in this model that is ultimately tied to the computer trading problem. They are all related to the generic concept of trading, however, they differ from the EBTs in that they are externally stable, but can be internally changed. For instance, the result of any trade is an agreement, however, the physical nature of this agreement can differ from one application to another. The contract used to buy a computer is completely different from the contract used to buy a house, yet, in the end, they are both considered to be agreements.

In the SSM, the variety of physical representations of BOs, as in the agreement case, is handled through the IOs of the system. For example, the actual implementation of the agreement in any trading application is an IO in that system. This IO can completely differ from one system to another, although the BO "Agreement" is the same.

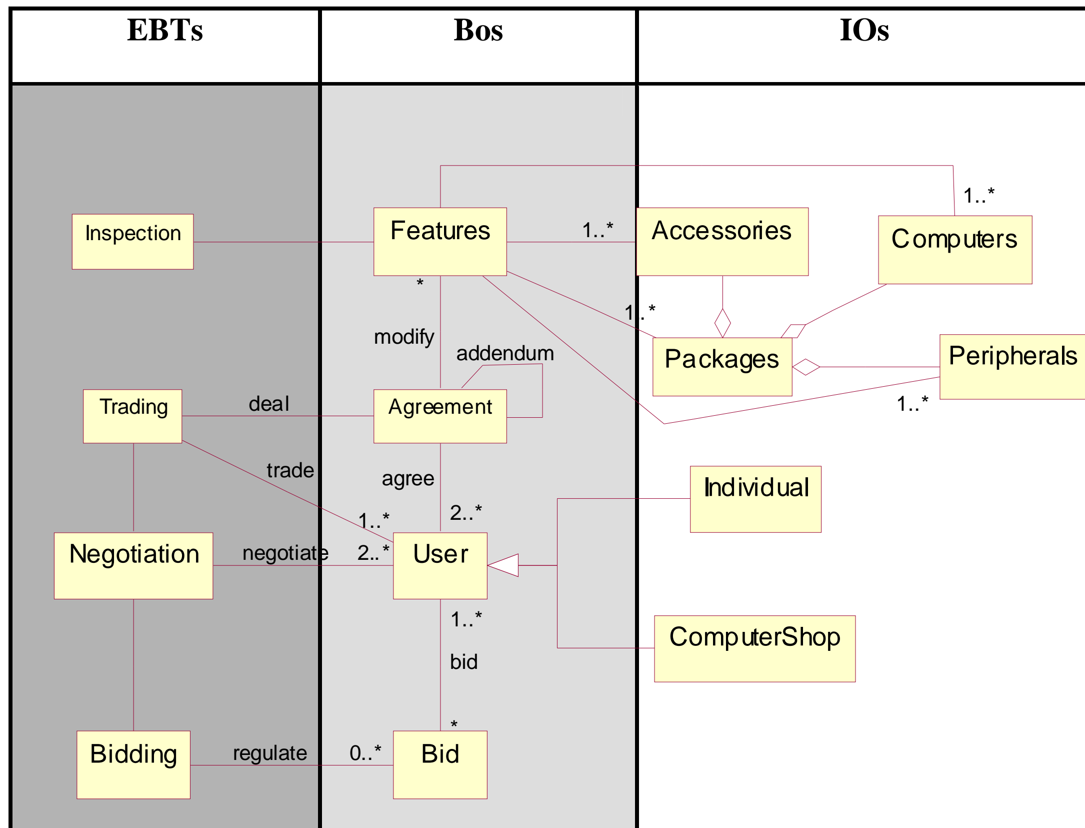


Figure 3.2**:** Computers Trading Stable Model

3.5.2 Case II- Buying a House

Buying a house might look different from trading a computer, however they are both forms of the concept, trading. Therefore, it is expected that many objects will be common between both applications. These objects are the EBTs and the BOs of the system. Figure 3.3 shows the stable model of this case study.

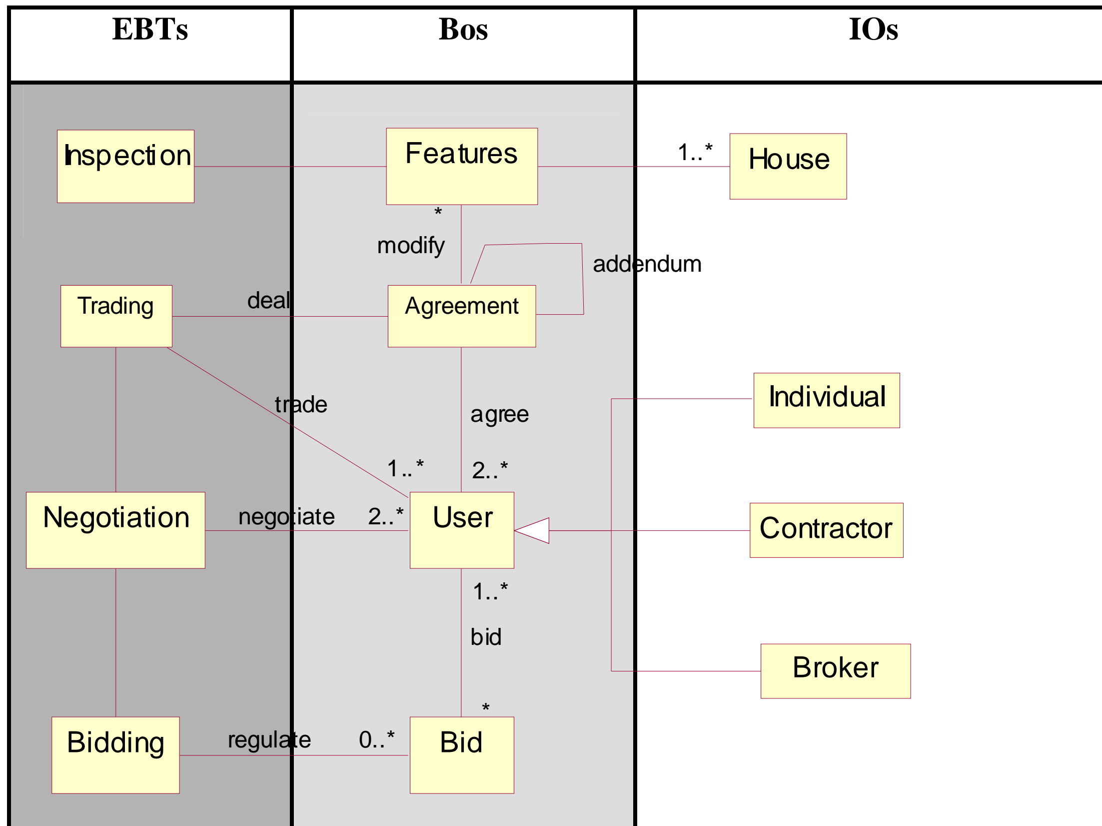


Figure 3.3: House Trading Stable Model

3.5.3 Case III- Bidding on a Football Team

Without using software stability concepts, the concept of buying a football team might look completely unrelated to the concept of buying a computer or a house. However, software stability concepts will help to view the problem at a more abstract level. From the SSM point of view, buying a football team can also be viewed as trading. It is unnecessary to consider the object that is to be physically traded; the football team case appears as just another collection of IOs. The same model provided in Figure 3.3 can be used in this case study, with the need only to define this new set of the IOs. Figure 3.4 shows the stable model of this case study.

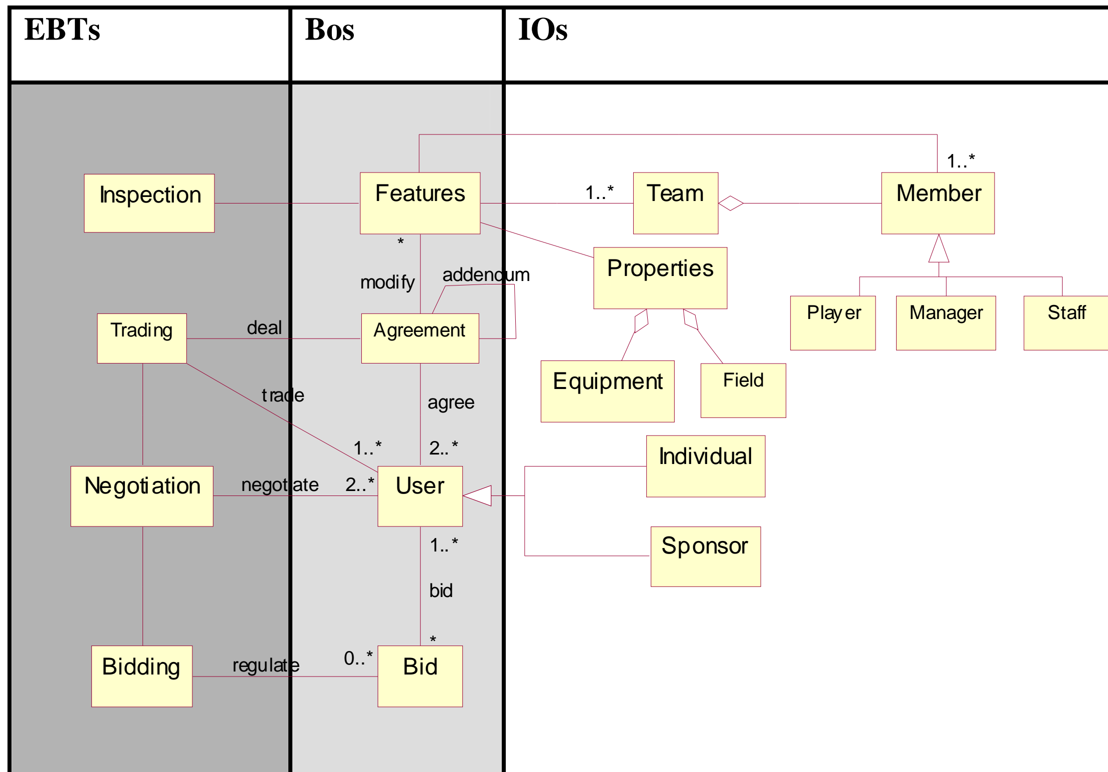


Figure 3.4: Bidding on a Football Team Stable Model

3.5.4 Compound System

Figure 3.5 shows the compound model resulting from the merging of the three stable models from the three described case studies. From the shown Figure, it is clear how these three different applications share a common core. In the level of the EBTs and the BOs, all three of these applications are viewed as trading systems, regardless of what each application physically trades. Moving down to the physical level (IO level), these three applications diverge, since each application has its own appropriate IOs.

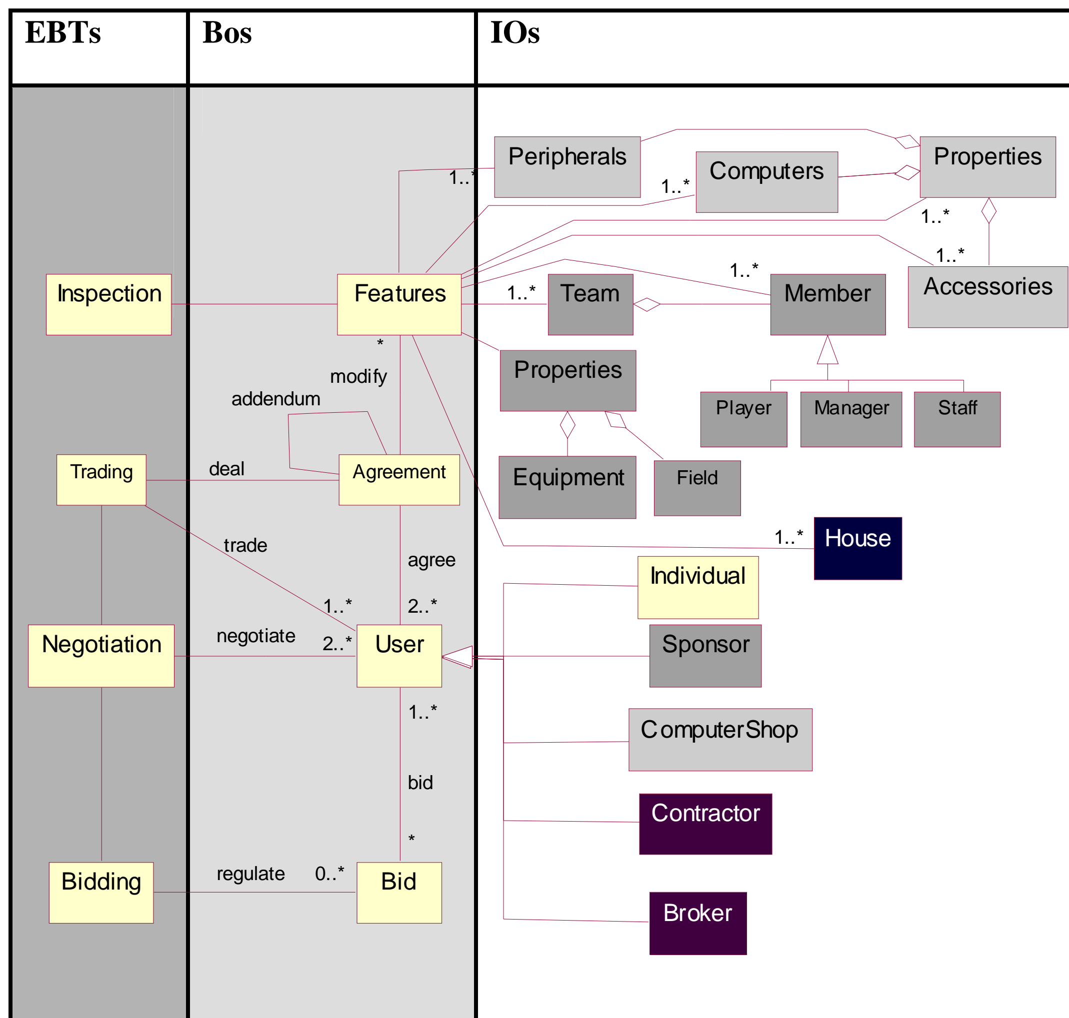


Figure 3.5: Combined Stable Models

Chapter 4

Software Stable Analysis Patterns

4.1 Introduction

Analysis patterns are conceptual models that model the core knowledge of the problem. Therefore, it is expected that the pattern that models a specific problem should be easily and successfully reused to model the same problem, regardless of the context in which the problem appears. In reality, this is not always the case. In fact, many of today's analysis patterns model well-known problems that span many domains. Yet, using these patterns to model the same problem in different contexts, if it is possible, is not as easy as it should be. As a result, software developers frequently are forced to start their analysis from scratch.

Building analysis patterns with stability in mind will help in producing effective and reusable patterns. These patterns can be used to model the problem whenever the problem appears, independent of the context of the problem. In this chapter, we introduce the novel concept of *stable analysis patterns,* a novel documentation template, and a detailed example, building and utilizing stable analysis patterns.

4.2 Stable Analysis Patterns Concept

Software stability concepts have demonstrated great promise in the area of software reuse and lifecycle improvement. The concepts of EBTs and BOs have been shown to produce models that are both stable over time, and stable across various paradigm shifts within a domain or application context [16, 17, 18, 19].

By applying stability concepts to the notion of analysis patterns we propose the concept of Stable Analysis Patterns [12, 13]. The idea behind stable analysis patterns is to analyze the problem under consideration in terms of its EBTs and the BOs with the goal of increased stability and broader reuse. By analyzing the problem in terms of its EBTs and the BOs, the resultant pattern models the core knowledge of the problem. The goal of this concept is stability. As a result, these stable patterns can be used to model the same problem regardless its context.

4.3 Stable Analysis Pattern Template

Documenting analysis patterns is as important as building the patterns themselves. Lacking a clear description of a pattern, the reusability of the pattern will diminish, forcing software developers to reinvent the wheel, analyzing a well-known problem from scratch.

The proposed pattern description template is as follows:

- **Name:** Presents the name of the presented pattern.
- **Context:** Gives possible scenarios for the situations in which the pattern may recur.
- **Problem** Presents the problem the pattern concentrates on.

- **Forces** Illustrates the challenges and the constraints that the pattern needs to resolve.
- **Pattern Structure and Participants** Gives the class diagram of the pattern. It also introduces briefly each class and its role.
- **CRC- Cards** Summarizes the responsibility and collaboration of each participant. Each participant should have only one well defined responsibility in its CRC- Card. Participants with more than one responsibility should be presented with more than one CRC- Card, each CRC- Card will handle one of these responsibilities.
- **Applicability** Provides clear and detailed case studies for applying the pattern in different contexts. The following sub elements represent the required details.
    1. *Case Studies.* Shows the scenario of two or three cases studies from different contexts
    2. *Use Case Diagram.* Presents the different Use Cases and the Actors for each case study, and shows the relation between the different Use Cases, and the relation between these Use Cases and the Actors of the system.
    3. *Use Case Description.* Gives detailed description for each Use Case.
    4. *Behavior Diagram.* Presents the sequence diagrams and/or the state transition diagrams of the Use Cases.
- **Design Issues** Discuss the important issues required for linking the analysis phase to the design phase. <u>*"Future extension".*</u>
- **Formalization** Gives the formal description of the pattern. This is useful for verification purposes. Currently, we are using the Z specifications. <u>*"Future extension".*</u>

- **Related Patterns** Present the analysis and design patterns that are related to the proposed pattern.
- **Possible Extensions** Shows how this pattern can be extended to capture new situations in different contexts.
- **Known Uses** Gives examples of the use of the pattern within existing systems.

4.4 Stable Analysis Pattern Example

4.4.1 Pattern Name: *AnyNegotiation*

This pattern is required to model the negotiation problem regardless of the context it might appear in, hence the name *AnyNegotiation* is chosen.

4.4.2 Context

Negotiation is a general concept that has many applications within many different contexts. For instance, the trading of property usually involves negotiation (e.g. buying a house or a car). Negotiation might also appear in solving social and political conflicts. In addition, negotiation can also play a role in scientific applications. For instance, content negotiation in Internet applications, where the client and server perform negotiations in order to obtain the appropriate contents that fit the client preferences and capabilities

4.4.3 Problem

How to build a model for the negotiation concept, such that this model can be used to model the negotiation problem in any context?

4.4.4 Forces

- Negotiation spans many contexts that are completely different in their natures.
- The Negotiation process can take place between two or more persons, persons and organizations or between two non-human entities, in each case the negotiator structure is completely different. How can we handle these different structures using a single model?
- The Negotiation entity can be an organization consists of many persons; each has his role in the negotiation process. For instance, there can be one individual who is responsible for negotiating financial issues, another who is responsible for negotiating issues related to management, and so on. Therefore, our pattern should be flexible enough to handle different negotiator structures.
- Negotiation can be conducted through one or more media simultaneously or consecutively, thus, the pattern should handle the use of multiple media.
- Negotiation can be performed on one or more affairs at the same time. For instance, negotiation that takes place in the buying and selling context usually involves more than one subject to be negotiated. For example, in buying a car, one can negotiates the price, the warranty, and so on. Therefore, the pattern should be flexible enough to handle such situations.
- The ultimate goal of any negotiation is to reach an agreement between the negotiators. However, the nature of this agreement varies tremendously from one application to another and from one context to another. An agreement that might be reached while negotiating a political conflict is completely different from that reached

while buying a car. Therefore, the pattern should be able to handle these wide variations.

4.4.5 Pattern structure and Participants

Figure 4.1 shows the object diagram of the *AnyNegotiation* pattern. The pattern contains classes and packages.

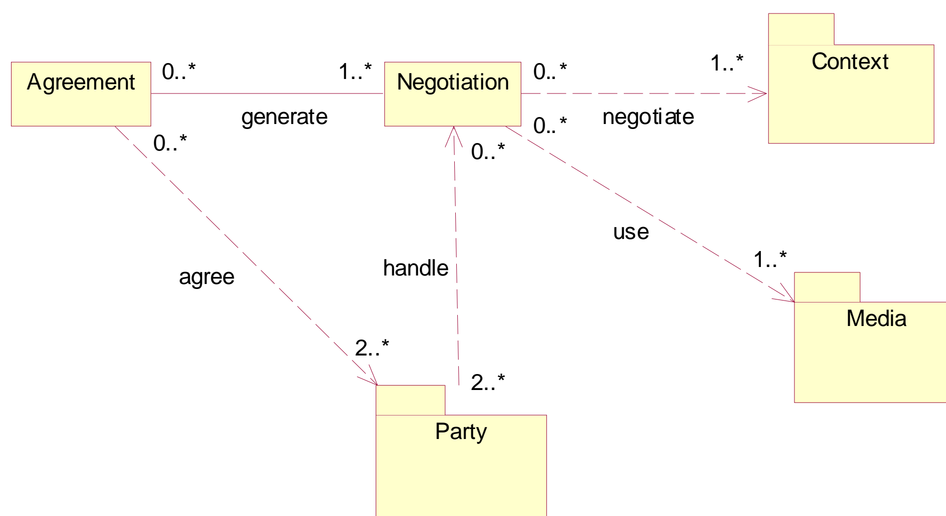


Figure 4.1: *AnyNegotiation* pattern object diagram

<u>Participants</u>

The participants of the *AnyNegotiation* pattern are:

<u>1. Classes</u>

- *Agreement.* Represents the result of the negotiation. Agreement is always present, whenever there is a negotiation. However, the nature of this agreement varies, depending on the context of the negotiation. For instance, a negotiation to buy a car

might results in a paper contract and a negotiation between two countries that have a conflict might result in a truce.

- *Negotiation.* Represents the negotiation process itself. This class contains the behaviors and attributes that regulate the negotiation process itself.

2. Packages

- *Party.* Represents the negotiation handlers. Party can be a person, organization, or a group with specific orientation. Figure 4.2 gives the class diagram of this package.
- *Media.* Represents the media through which the negotiation will take place. It is possible to use different medias to conduct a single negotiation. Figure 4.3 gives the class diagram of this package.
- *Context.* Represents the matters to be negotiated. It is possible to negotiate more than one affair within the same negotiation process. For instance, one can negotiate the price and the method of delivery when buying a product. Figure 4.4 gives the class diagram of this package.

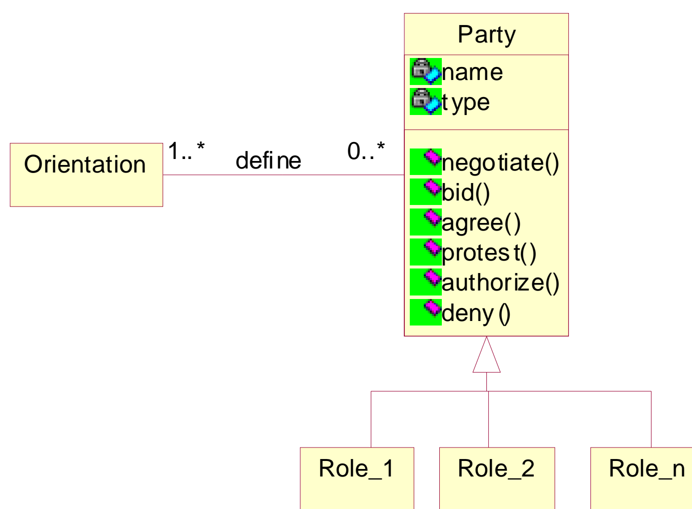


Figure 4.2: Party package

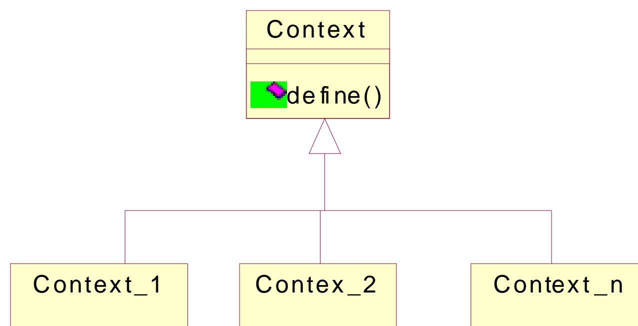


Figure 4.3: Context package

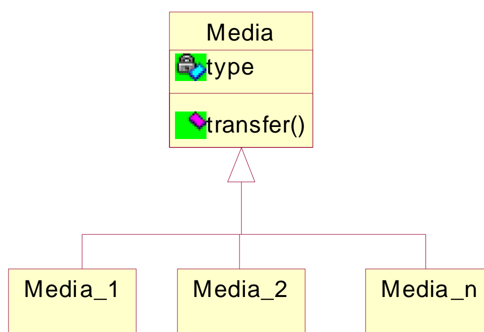


Figure 4.4: Media package

4.4.6 CRC- Cards

Negotiation (Negotiation Descriptor)		
Responsibility	**Collaboration**	
	Clients	**Server**
Describes the negotiation rules, and regulations to the negotiating parties.	- Agreement - Party - Media - Context	- Define rules

Party (Negotiation handler)		
Responsibility	**Collaboration**	
Performs and finalizes the negotiation.	**Clients**	**Server**
	- Negotiation - Agreement	- Negotiate - Approve

Agreement (Descriptor)		
Responsibility	**Collaboration**	
Describes agreement terms and conditions.	**Clients**	**Server**
	- Negotiation - Party	- Define agreement -

Media (Connector)		
Responsibility	**Collaboration**	
Communicates negotiation issues between negotiators.	**Clients**	**Server**
	- Negotiation	- Connect parties

Context (Motivator)		
Responsibility	**Collaboration**	
Defines the reason of the negotiation process.	**Clients**	**Server**
	- Negotiation	- Define the matter.

4.4.7 Applicability

In practical applications, the *AnyNegotiation* pattern is used within the negotiation system. The negotiation system consists of the *AnyNegotiation* pattern in addition to the Bidding package. Figure 4.5 gives the class diagram of the negotiation system. The "Bidding" package represents the bidding process. For instance, when negotiating the price of a car, both the customer and the car dealer bid in the price and then the negotiation of this price will take place. The CRC- Card of the "Bidding" package is given below.

Bidding (Bidding Regulator)		
Responsibility	**Collaboration**	
To define the rules and requirements for posting a bid.	**Clients**	**Server**
	- Party - Media - Context - Authorization	- Place bid - Compromise bid - Remove bid

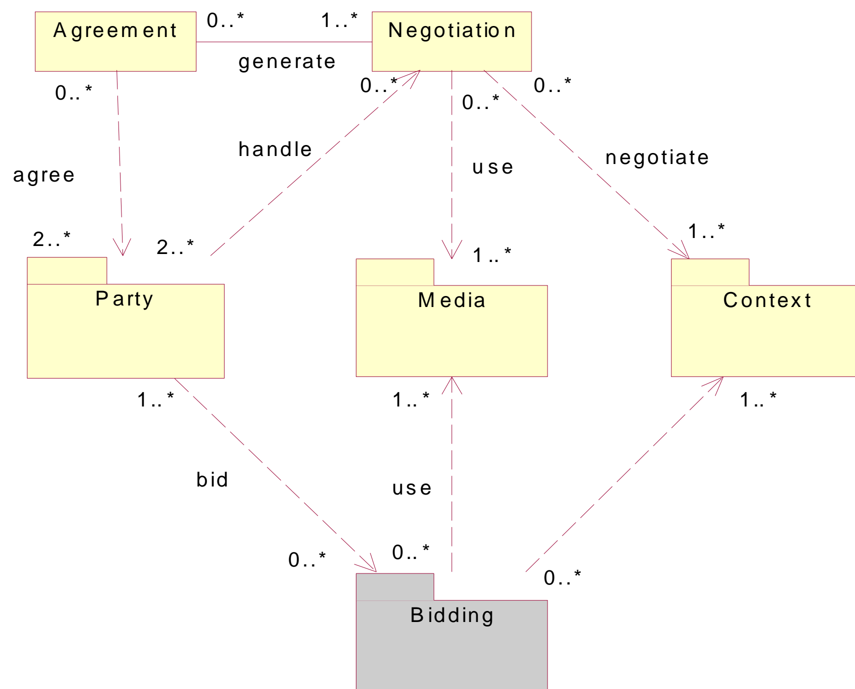


Figure 4.5: Negotiation system object diagram.

In order to demonstrate the use of the "AnyNegotiation" pattern in different contexts, two case studies are presented and the complete model for each case study is given.

4.4.7.1 Case Study 1: Negotiation to buy a car

Problem Description

This use case models the simple process of buying a car. In buying a car, a negotiation concerning the car's price and warranty usually take place. This case study shows how

to utilize the *AnyNegotation* pattern in the analysis of the problem and to build the stable model of the problem.

Problem Class Diagram

Figure 4.6 shows the stability model of the negotiation used in buying a car. The IOs added onto the top of the original *AnyNegotiation* pattern are colored in gray.

Note: The objective of this case study to demonstrate the usage of the proposed pattern, however, for simplicity, it dose not give the complete model for the problem. The complete model of this problem should contain other EBTs and BOs. For instance, we might add the EBTs: (Trading, Inspection, etc.), and the BOs: (Features, Item, etc.)

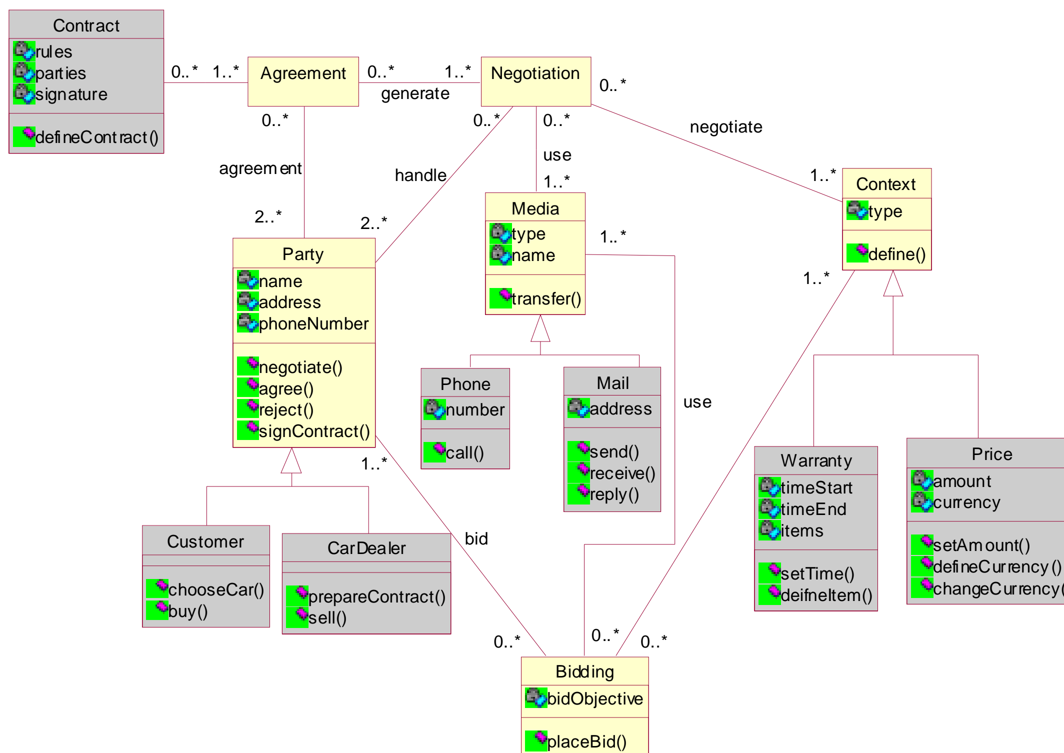


Figure 4.6: Stability model of the negotiation in buying a car case study.

8 Use Case Diagram

Figure 4.7 shows the use case diagram for the case study.

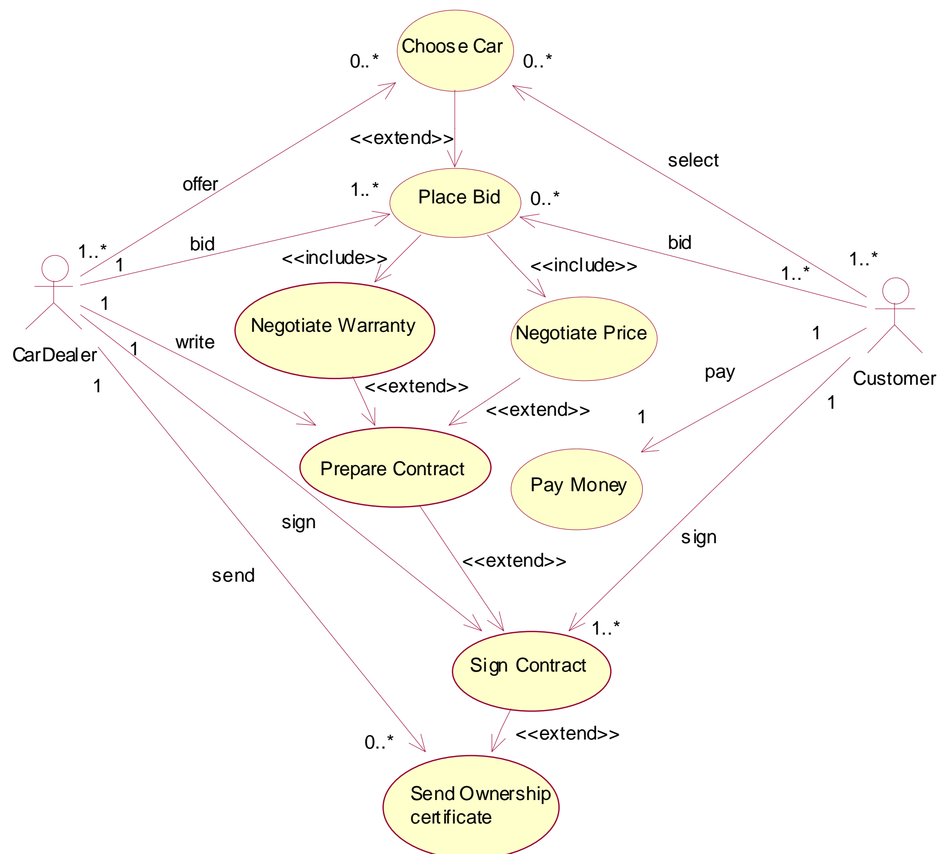


Figure 4.7: Use case diagram of the negotiation in buying a car case study.

<u>Use Case Description</u>

In this subsection, in order to show how to apply the proposed template, five use cases of the system are documented.

***Use Case Id*:** 1.0

Title: Place a Bid: car Price

Actors	Roles
Customer	Bidder
CarDealer	Bidder

Class	Supper Class	Attributes	Operations
Customer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() ChooseCar() Buy()
CarDealer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() PrepareContract() Sell()
Bidding	-	BidObjective	PlaceBid()
Price	Context	Type, Amount, Currency	Define() SetAmount() DefineCurrency() ChangeCurrency()
Phone	Media	Type, Name, Number	Transfer() Call()

EBTs: Bidding.

BOs: Party, Media, Context.

IOs: Customer, CarDealer, Phone, Price

Description:

1. The Customer contacts the CarDealer through the appropriate Media (e.g. through the Phone).
2. The Customer requests the Price of the car he needs from the CarDealer.
3. The CarDealer initiates Bidding by providing the price of the car the Customer wants.
4. The Customer receives the Price of the car he wants.

Use Case Id: 2.0

Title: *Negotiate Car Price*

Actors	Roles
Customer	Bidder, Negotiators
CarDealer	Bidder, Negotiators

Class	Supper Class	Attributes	Operations
Customer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() ChooseCar() Buy()
CarDealer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() PrepareContract() Sell()
Bidding	-	BidObjective	PlaceBid()
Negotiation	-	-	-
Price	Context	Type, Amount, Currency	Define() SetAmount() DefineCurrency() ChangeCurrency()
Phone	Media	Type, Name, Number	Transfer() Call()

EBTs: Bidding. Negotiation.

BOs: Party, Media, Context.

IOs: Customer, CarDealer, Phone, Price

Description:

1. The Customer starts his Bidding on the Price. The Customer proposed his desired Price, and sends it to the CarDealer.
2. The CarDealer receives the proposed Price from the Customer.
3. The CarDealer Negotiates the Price proposed by Customer.
4. The CarDealer starts his Bidding on the price proposed by the Customer. The CarDealer proposes new Price, and sends it the Customer.
5. The Customer Negotiates the new Price.
6. The Customer starts new Bidding. He will propose a new Price for the car, and sends it to the CarDealer.

Alternatives:

4.a The CarDealer agrees on the proposed Price by the Customer.

5.a The Customer agrees on the new Price of the car.

Use Case Id: 3.0

Title: *Negotiate Car Warranty*

Actors	Roles
Customer	Negotiators
CarDealer	Negotiators

Class	Supper Class	Attributes	Operations
Customer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() ChooseCar() Buy()
CarDealer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() PrepareContract() Sell()
Bidding	-	BidObjective	PlaceBid()
Negotiation	-	-	-
Warranty	Context	Type, TimeStart, TimeEnd, ItemsCovered	Define() SetTime() DefineItem()
Phone	Media	Type, Name, Number	Transfer() Call()

EBTs: Bidding. Negotiation.

BOs: Party, Media, Context.

IOs: Customer, CarDealer, Phone, Warranty

Description:

1. The CarDealer describes the Warranty offered with the selected car to the Customer, through the appropriate Media.
2. The Customer Negotiates the Warranty time period.
3. The Customer starts Bidding on Warranty time period. He proposes the desired time.
4. The CarDealer Negotiates the proposed Warranty time period.
5. The CarDealer starts his Bidding on the time period proposed by the Customer. The CarDealer proposes new time period, and sends it the Customer.
6. The Customer Negotiates the items covered by the offered Warranty.
7. The Customer starts Bidding on the items covered by the offered Warranty.
8. The CarDealer start Bidding on the propose Warranty by the Customer.

Alternatives:

2.a The Customer agrees on the Warranty time period.

5.a The CarDealer agrees on the proposed Warranty time period.

5.a The Customer agrees on the Warranty items.

7.a The CarDealer agrees on the proposed Warranty.

Use Case Id: 4.0

Title: *Prepare Contract*

Actors	Roles
CarDealer	Prepare contract, Send contract
Customer	Receive contract

Class	Supper Class	Attributes	Operations
CarDealer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() PrepareContract() Sell()
Customer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() ChooseCar() Buy()
Warranty	Context	Type, TimeStart, TimeEnd, ItemsCovered	Define() SetTime() DefineItem()
Price	Context	Type, Amount, Currency	Define() SetAmount() DefineCurrency() ChangeCurrency()
Mail	Media	Type, Name, Number	Transfer() Call()
Contract	Agreement	Rules, Parties, Signature	DefineContract()

EBTs: <none>

BOs: Party, Media, Context, Agreement.

IOs: CarDealer, Customer, Mail, Price, Warranty, Contract.

Description:

1. The CarDealer documents the final Price according to the Agreement with the Customer.
2. The CarDealer documents the final Warranty offer according to the Agreement with the Customer.
3. The CarDealer writes the official Contract.
4. The CarDealer sends the Contract to the Customer using the appropriate Media (e.g. the Mail).

Use Case Id: 5.0

Title: *Sign Contract*

Actors	Roles
Customer	Sign Contract
CarDealer	Sign Contract

Class	Supper Class	Attributes	Operations
Customer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() ChooseCar() Buy()
CarDealer	Party	Name, Address, PhoneNumber	Negotiate() Agree() Reject() SignContract() PrepareContract() Sell()
Mail	Media	Type, Name, Number	Transfer() Call()
Contract	Agreement	Rules, Parties, Signature	DefineContract()

EBTs: <none>

BOs: Party, Media, Context, Agreement.

IOs: CarDealer, Customer, Mail, Contract.

Description:

1. The CarDealer signs the final Contract.
2. The CarDealer sends the final Contract to the Customer using the appropriate Media (e.g. using the Mail).
3. The Customer receives the final Contract.
4. The Customer signs the final Contract.
5. The Customer sends the final Contract to the Customer using the appropriate Media (e.g. using the Mail).
6. The CarDealer and the Customer reach Agreement.

Behavior Diagrams

I. Sequence Diagrams

Use Case 1: *Place a Bid*

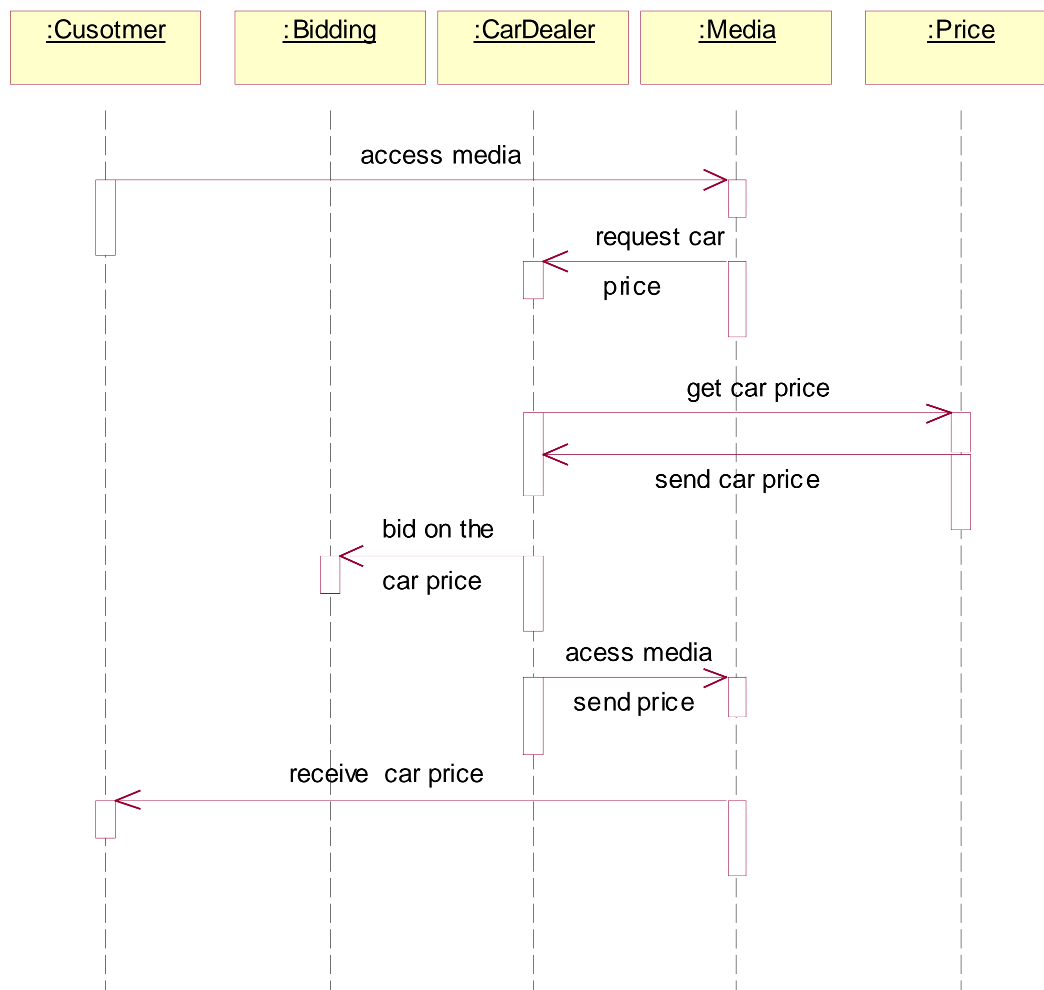


Figure 4.8: Place a Bid sequence diagram.

Use Case 2: Negotiate Car Price

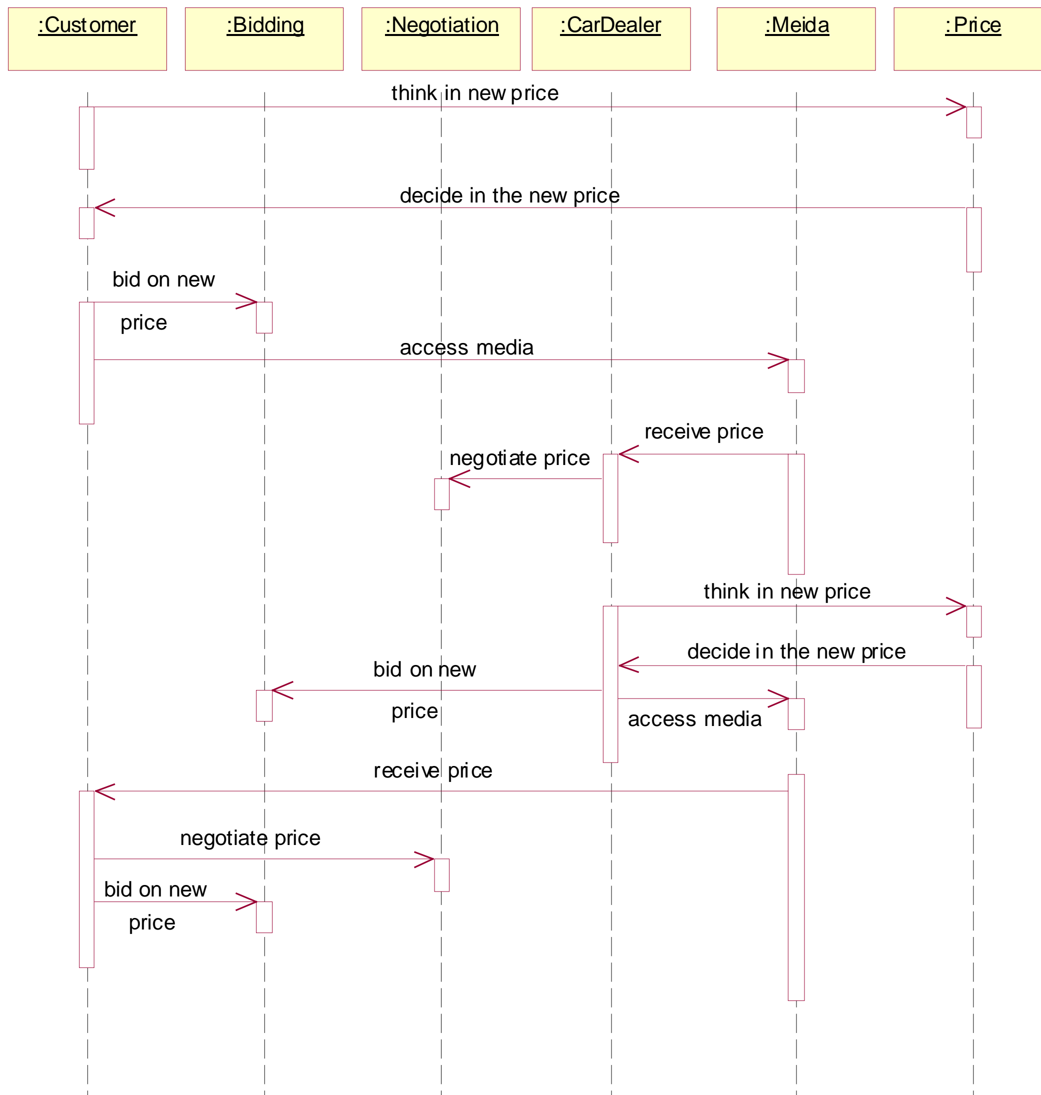


Figure 4.9: Negotiate Car Price sequence diagram.

Use Case 3: ***Prepare Contract***

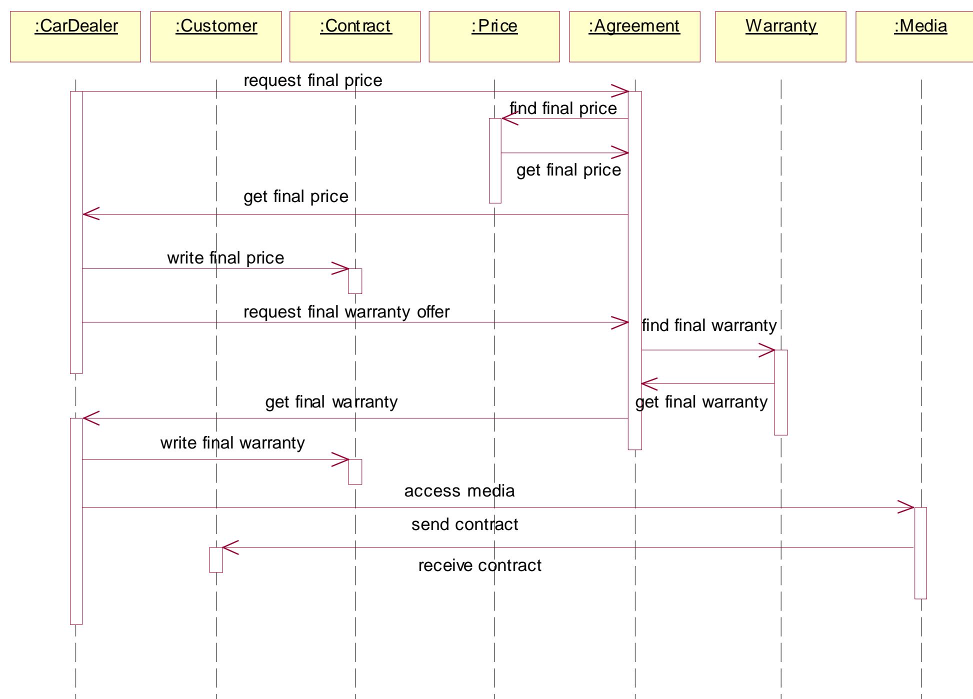


Figure 4.10: Prepare Contract sequence diagram.

Use Case 4: ***Sign Contract***

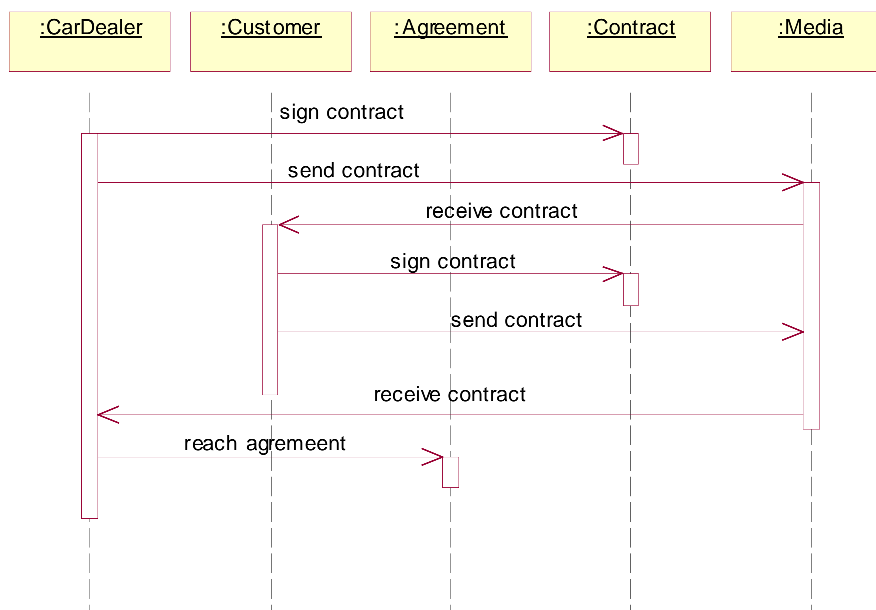


Figure 4.11: Sign Contract sequence diagram.

II. State Transition Diagrams

1. Negotiate Price

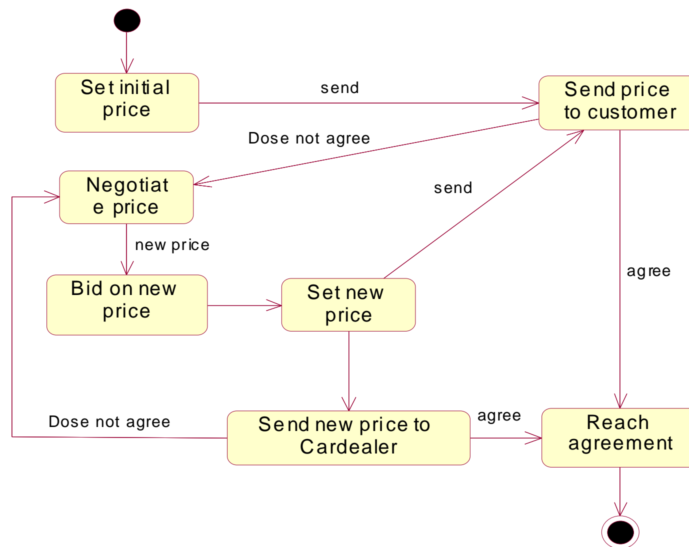


Figure 4.12: Negotiate Price state transition diagram.

2. Sign Contract

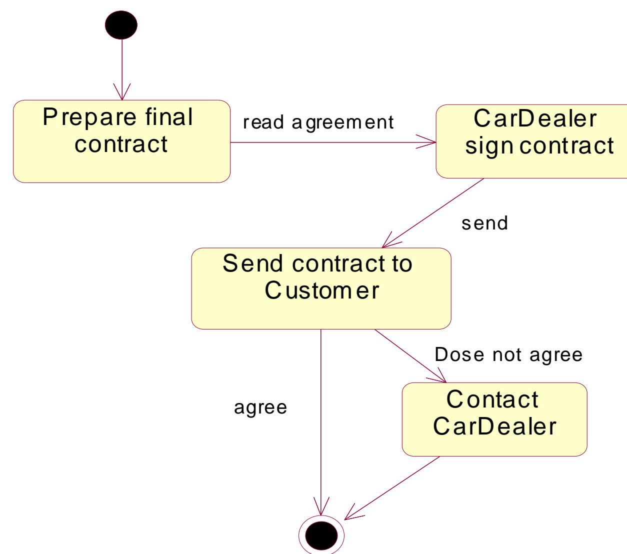


Figure 4.13: Sign Contract state transition diagram.

4.4.7.2 Case Study 2: Content Negotiation using Composite Capability/ Preference Profile (CC/PP)

Problem Description

Today, very heterogeneous devices are required to access the World Wide Web. Yet, each device has its own set of capabilities. Therefore, the server will need to know the capabilities of these devices in order to provide the appropriate contents. As a result, a negotiation between the client that needs to access the web and the server that originates content (the origin server) is needed. This case study models one of the known techniques of performing content negotiation called Composite Capability/Preference Profile (CC/PP) [4]. One possible scenario of content negotiation using CC/PP is given in figure 4.14.

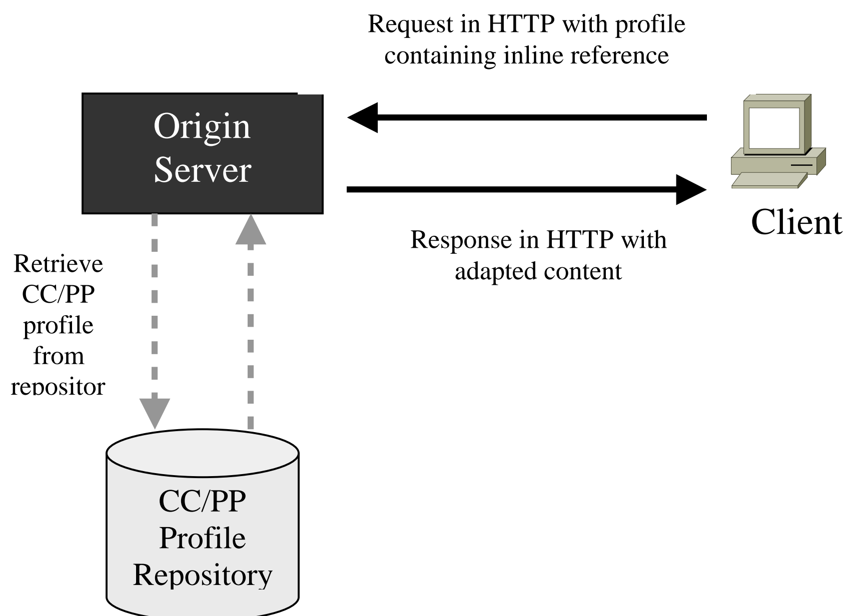


Figure 4.14: Possible scenario of content negotiation using CC/PP.

Problem Class Diagram

Figure 4.15 shows the stability model of the content negotiation using CC/PP. The IOs added onto the top of the core pattern are colored gray.

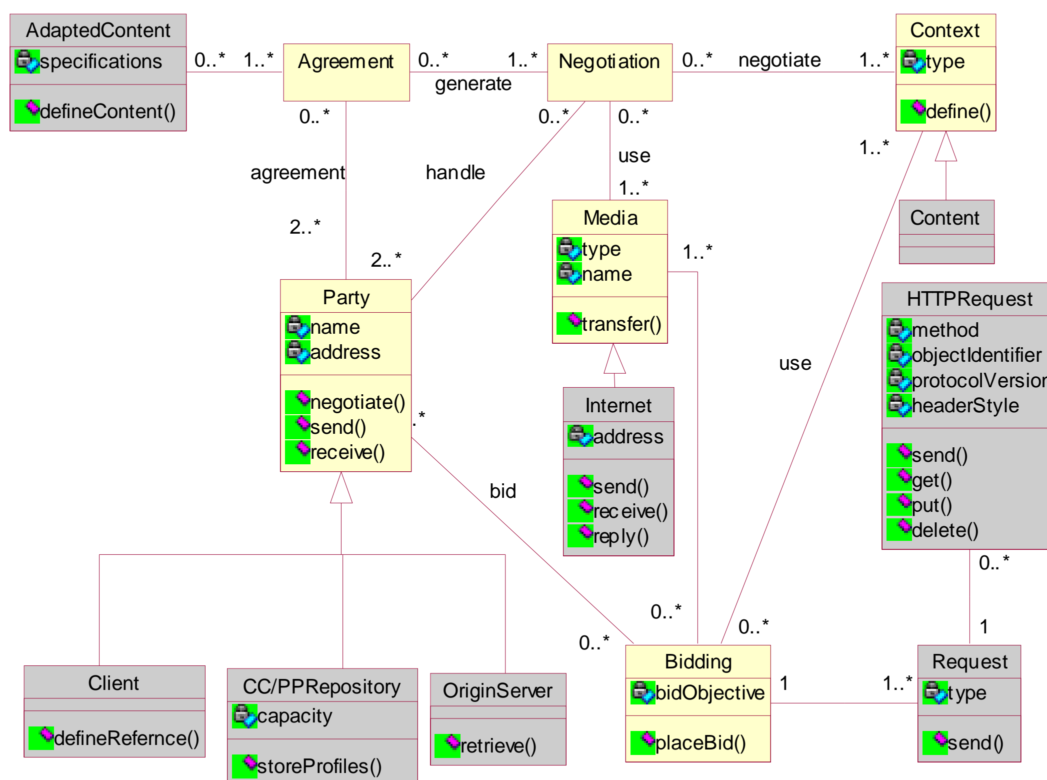
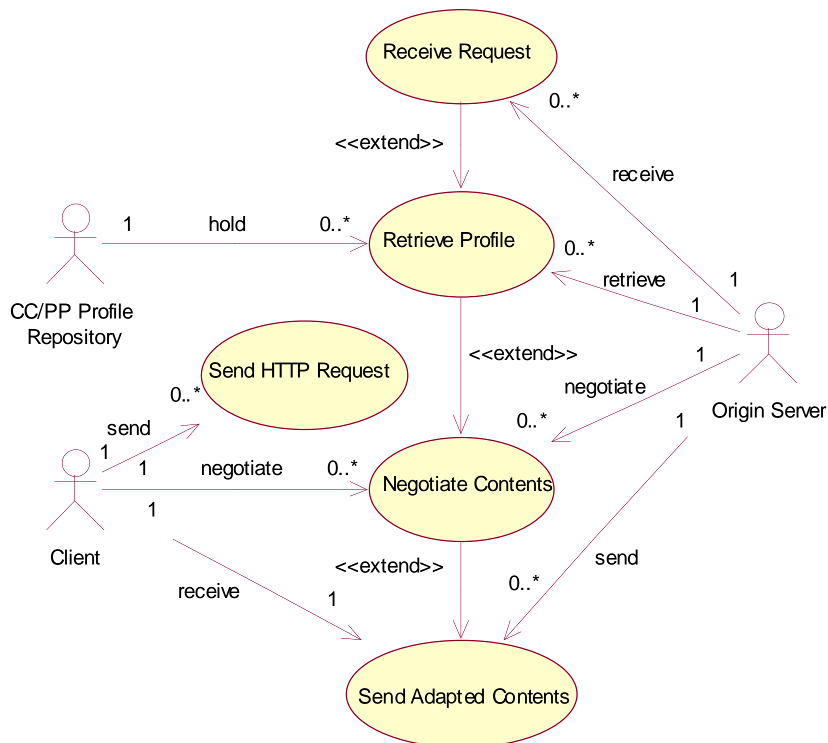


Figure 4.15: The stability model of the content negotiation case study.

Use Case Diagram

Figure 4.16 shows the use case diagram for the case study.



.

Figure 4.16 Use Case diagram for content negotiation case study.

Use Case Description

In this subsection, the full documentation is given for five use cases.

Use Case Id: 1.0

Title: *Send HTTP Request*

Actors	Roles
Client	Sender
OriginServer	Receiver

Class	Supper Class	Attributes	Operations
Client	Party	Name, Address	Negotiate() Send() Receive() DefineReference()
OriginServer	Party	Name, Address	Negotiate() Send() Receive() Retrieve()
Bidding	-	BidObjective	PlaceBid()
Internet	Media	Type, Name, Address	Transfer() Send() Receive() Reply()
Request	-	Type	Send()
HTTPRequest	-	Method, ObjectIdentifier, ProtocolVersion, HeaderStyle	Send() Get() Put() Delete()

EBTs: Bidding.

BOs: Party, Media, Context, Request.

IOs: Client, OriginServer, Internet, HTTPRequest

Description:

1. The Client starts Bidding by preparing a reference to its profile.
2. The Client sends HTTPRequest to the OriginServer through the appropriate Media (Here the Media is the Internet).
3. The OriginServer receives the HTTPRequest.

Use Case Id: 2.0

Title: *Retrieve Profile*

Actors	Roles
OriginServer	Sender, Retriever
CC/PPRepository	Receiver, Sender

Class	Supper Class	Attributes	Operations
OriginServer	Party	Name, Address	Negotiate() Send() Receive() Retrieve()
CC/PPRepository	Party	Name, Address, Capacity	Negotiate() Send() Receive() StoreProfiles
Internet	Media	Type, Name, Address	Transfer() Send() Receive() Reply()
Request	-	Type	Send()
HTTPRequest	-	Method, ObjectIdentifier, ProtocolVersion, HeaderStyle	Send() Get() Put() Delete()

EBTs: <none>

BOs: Party, Media, Request.

IOs: OriginServer, CC/PPRepository, Internet, HTTPRequest

Description:

1. The OriginServer extracts the reference from the HTTPRequest.
2. The OriginServer sends the reference to the CC/PPRepository
3. The CC/PPRepository receives the reference from the OriginServer.
4. The CC/PPRepository searches for the Client profile.
5. The CC/PPRepository sends the Client preference profile to the OriginServer.
6. The OriginServer receives the client preference profile from the CC/PPRepository

Alternatives:

5.a The CC/PPRepository did not find corresponding profile to the provided reference, and sends error message to the OriginServer.

6.a The OriginServer receives an error message from the CC/PPRepository.

Use Case Id: 3.0

Title: *Adapt Contents*

Actors	**Roles**
OriginServer	Sender, Receiver, Negotiator

Class	**Supper Class**	**Attributes**	**Operations**
OriginServer	Party	Name, Address	Negotiate() Send() Receive() Retrieve()
Bidding	-	BidObjective	PlaceBid()
Internet	Media	Type, Name, Address	Transfer() Send() Receive() Reply()
Request	-	Type	Send()
HTTPRequest	-	Method, ObjectIdentifier, ProtocolVersion, HeaderStyle	Send() Get() Put() Delete()
Content	Context	Type	Define()

EBTs: Bidding

BOs: Party, Media, Request, Context.

IOs: OriginServer, Internet, HTTPRequest, Content.

Description:

1. The OriginServer reads the client profile, and extracts its capabilities and preferences.
2. The OriginServer initiate Bidding by requesting the Contents required in the HTTPRequest.
3. The OriginServer get the required Contents.
4. The OriginServer adapt the Contents according to the client profile.

Alternatives:

3.a The OriginServer does not find the required Contents.

Use Case Id: 4.0

Title: *Negotiate Contents*

Actors	Roles
OriginServer	Sender, Receiver, Negotiator
Client	Receiver, Sender, Negotiator
CC/PPRepository	Receiver, Sender

Class	Supper Class	Attributes	Operations
OriginServer	Party	Name, Address	Negotiate() Send() Receive() Retrieve()
Client	Party	Name, Address	Negotiate() Send() Receive() DefineReference()
CC/PPRepository	Party	Name, Address, Capacity	Negotiate() Send() Receive() StoreProfiles
Negotiation	-	-	-
Bidding	-	BidObjective	PlaceBid()
Internet	Media	Type, Name, Address	Transfer() Send() Receive() Reply()

EBTs: Bidding, Negotiation.

BOs: Party, Media

IOs: OriginServer, Client, Internet, CC/PPRepository.

Description:

1. The OriginServer receives error message from the CC/PPRepository indicating that the required reference does not exist.
2. The OriginServer Negotiates the profile with the CC/PPRepository. It initiates a new Bidding by sending a modified reference to the CC/PPRepository. It also saves the changes it made to the reference.
3. The CC/PPRepository receives the new reference and searches for the corresponding profile.
4. The CC/PPRepository sends the Client preference profile to the OriginServer.
5. The OriginServer receives the client preference profile from the CC/PPRepository.
6. The OriginServer Negotiates the new profile. The OriginServer start bidding by sending the new profile to the Client.
7. The Client receives the new profile and checks its identifiers and style.
8. The Client Negotiates the new profile. It initiates a new Bidding by sending a newer version of its acceptable profile.
9. The OriginServer receives the proposed profile from the Client. It sends the CC/PPRepository the new profile to match it with the closest profile available.
10. The CC/PPRepository sends the closest new profile to the OriginServer.

11. The OriginServer sends the new profile to the Client.

Alternatives:

4.a The CC/PPRepository did not find corresponding profile to the provided reference, and sends error message to the OriginServer.

5.a The OriginServer receives an error message from the CC/PPRepository.

6.a The OriginServer sends an error message from the client.

8.a The Clients sends an error message the OriginServer indicating the termination of Negotiation. (The new profile is too far from the acceptable one).

8.b The Client sends acceptance on the new profile to the OriginServer. (Means no further modifications are required).

10.a The CC/PPRepository does not have any closer profile. It sends an error message to the OriginServer.

11.a The Client receives an error message from the OriginServer. This will terminate the Content Negotiation, since no more profiles can be searched.

Use Case Id: 5.0

Title: *Send Adapted Contents*

Actors	Roles
OriginServer	Sender
Client	Receiver

Class	Supper Class	Attributes	Operations
OriginServer	Party	Name, Address	Negotiate() Send() Receive() Retrieve()
Client	Party	Name, Address	Negotiate() Send() Receive() DefineReference()
Internet	Media	Type, Name, Address	Transfer() Send() Receive() Reply()
Agreement	-	-	-
AdaptedContent	-	Method, ObjectIdentifier, ProtocolVersion, HeaderStyle	Send() Get() Put() Delete()

EBTs: <none>

BOs: Party, Media, Context, Agreement.

IOs: OriginServer, Client, Internet, AdaptedContent

Description:

1. The OriginServer adapts the requested contents according to the Client profile.
2. The OriginServer sends the AdaptedContent to the Client.
3. The Client receives the AdaptedContent from the OriginServer.
4. Agreement is reached between the Client and the OriginServer.

Behavior Diagrams

I. Sequence Diagrams

Use Case 1: Send HTTPRequest

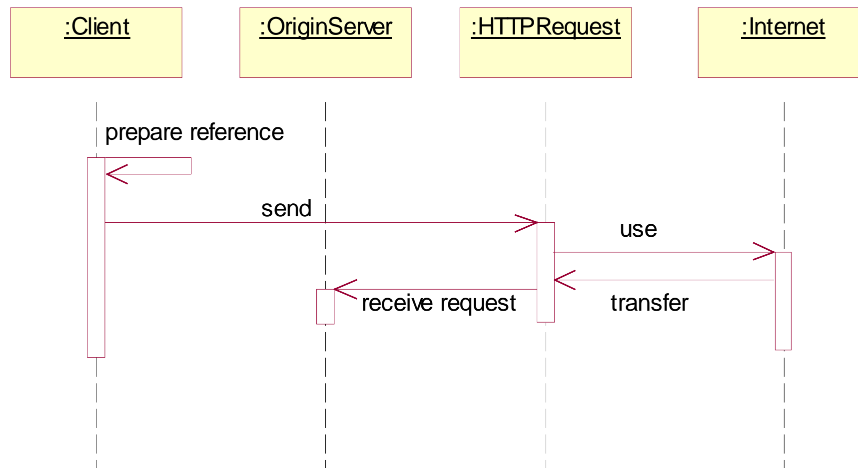


Figure 4.17: Send HTTPRequest sequence diagram.

Use Case 2: Retrieve Profile

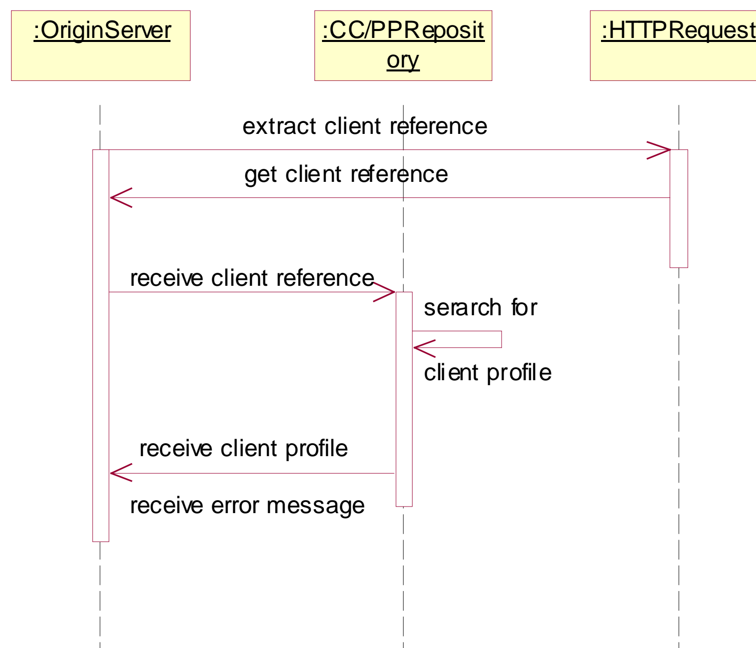


Figure 4.18: Retrieve Profile sequence diagram.

Use Case 3: Adapt Content

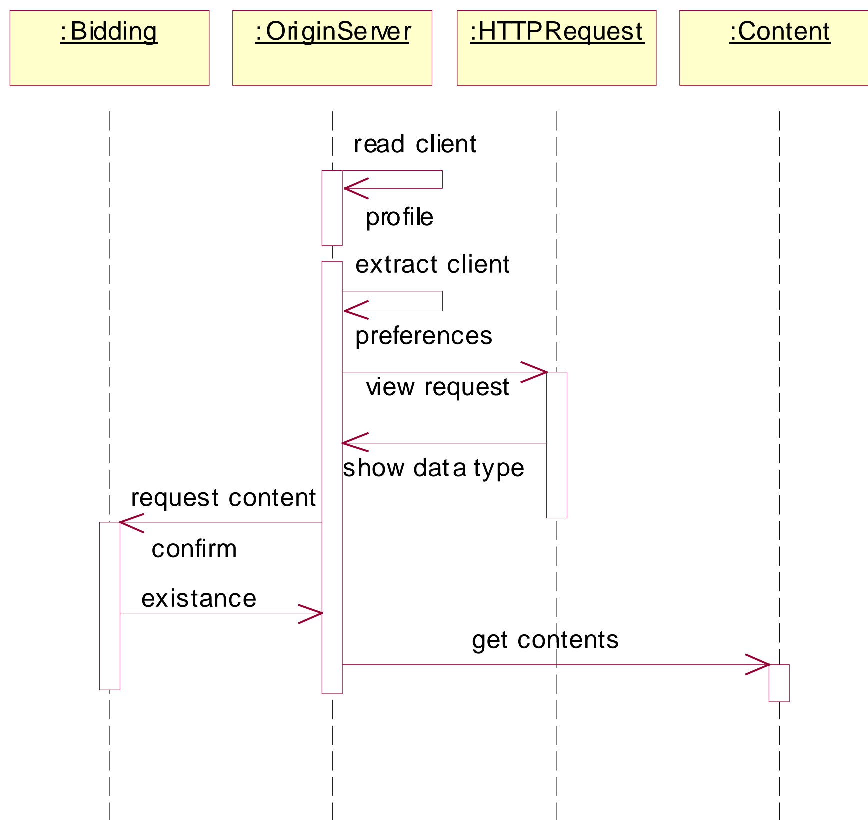


Figure 4.19: Adapt Content sequence diagram.

Use Case 4: Negotiate Content

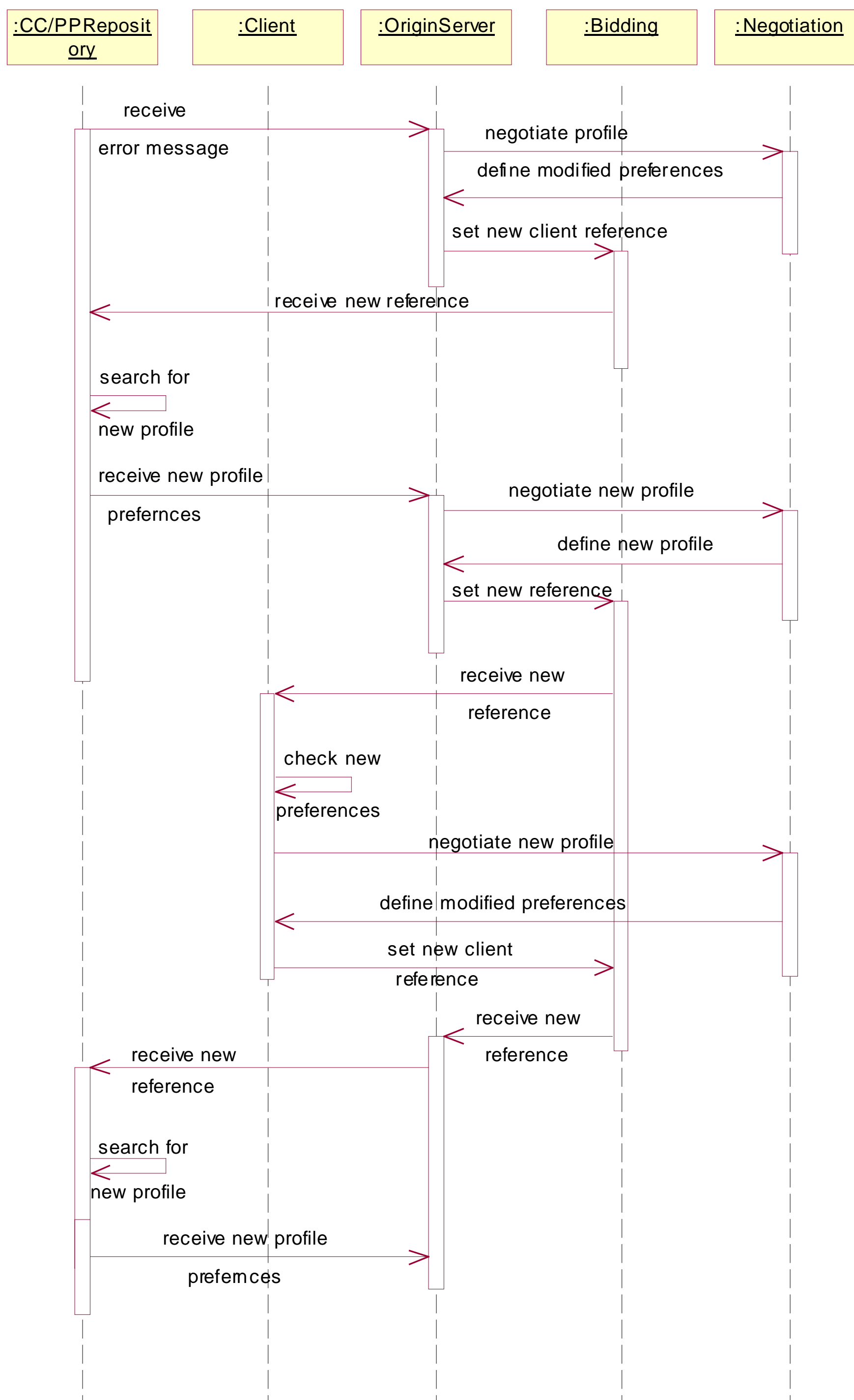


Figure 4.20: Negotiate Content sequence diagram.

II. State Transition Diagrams

1. Retrieve Profile

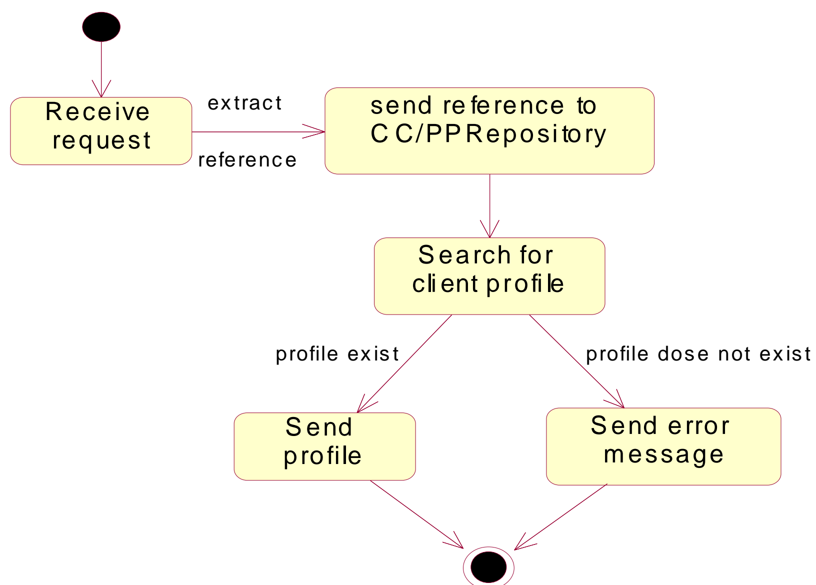


Figure 4.21: Retrieve Profile state transition diagram.

2. Negotiate Content

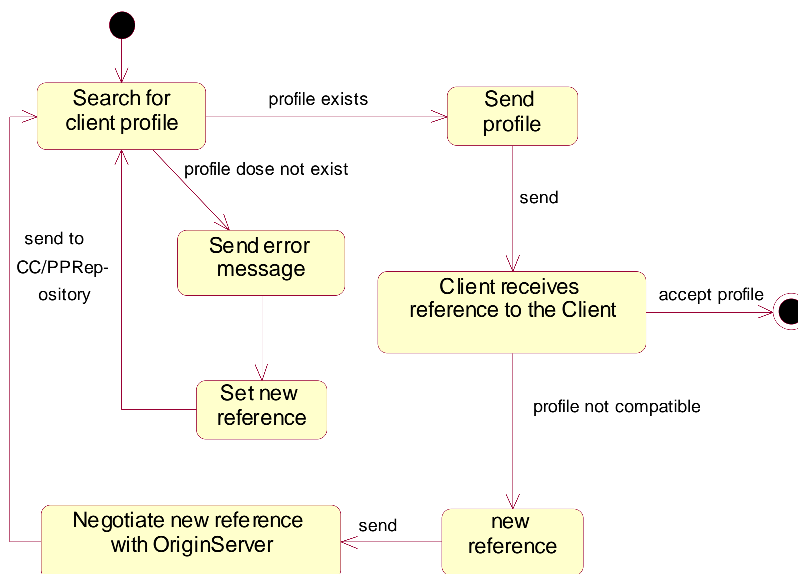


Figure 4.22: Negotiate Content state transition diagram.

4.4.8 Related Patterns

- Related *analysis patterns*: Bidding pattern
- Related *design patterns*:
    1. Party pattern
    2. Agreement pattern

4.4.9 Possible Extensions

The *"AnyNegotiation"* pattern models the basic negotiation process. It gives a stable model that can be reused to model the negotiation process regarding the negotiation context as shown in the two cases studies presented herein.

In more complex contexts, some additional classes might be needed in addition to the basic model of the negotiation system model. The flexibility of the stable concept that is used in the development of the *"AnyNegotiation"* pattern makes it easy to build on to model more complex negotiation systems. For instance, negotiation could involve "negotiation style". This feature models the different possible modes that negotiations can take. In [25], five styles for negotiation within the purchasing context are outlined: Competition style, accommodation style, avoidance style, compromise style, and collaboration style. A single negotiation can use more than one style at a time. You can negotiate the price using any of these styles, while negotiation the maintenance procedure using another.

Figure 4.10 shows how the NegotiationStyle package can be added onto the top of the *"AnyNegotiation"* pattern. The class diagram of the NegotiationStyle package is given in figure 4.11.

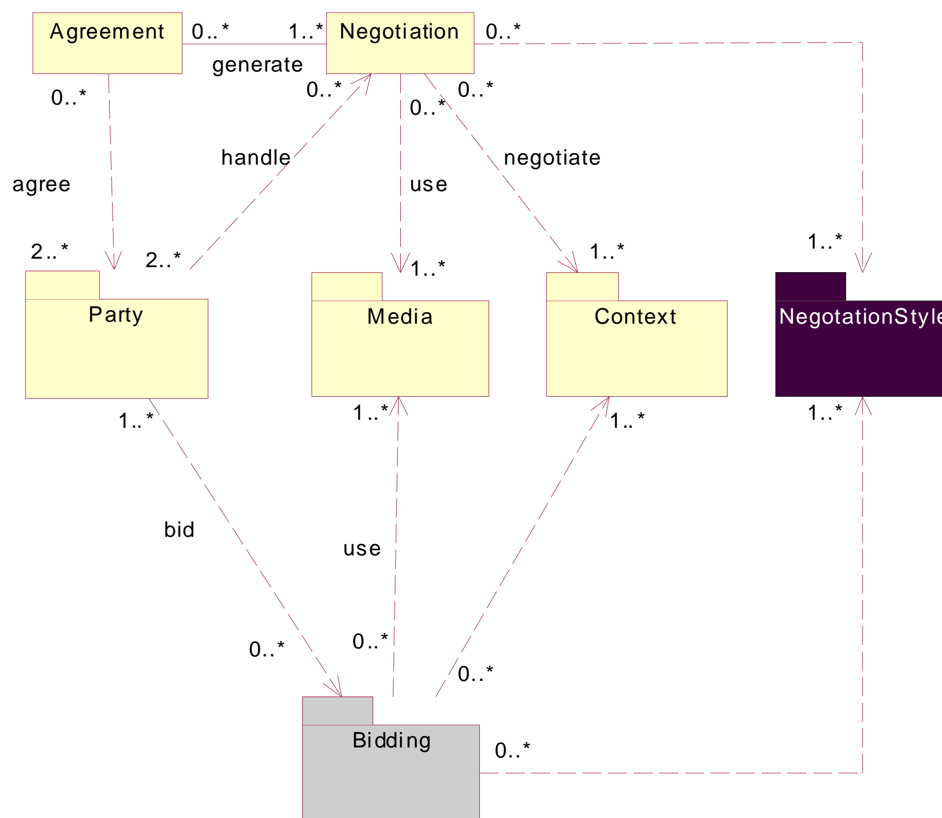


Figure 4.10: Negotiation system with NegotiationStyle package.

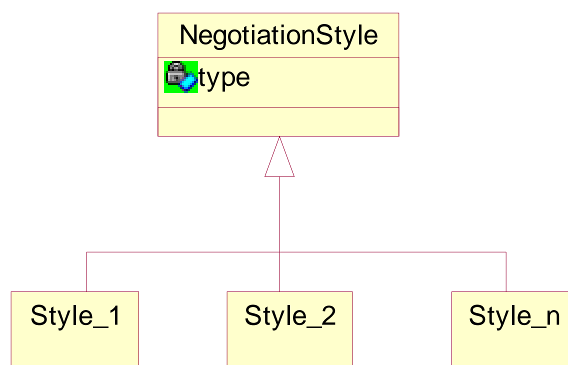


Figure 4.11 NegotiationStyle package class diagram.

Chapter 5

A Pattern Language for Building Stable Analysis Patterns

5.1 Introduction

Analysis patterns are conceptual models that model the core knowledge of the problem. Therefore, it is expected that the pattern that models a specific problem should be easily and successfully reused to model the same problem, regardless of the context in which the problem appears. In reality, this is not always the case. In fact, many of today's analysis patterns model well-known problems that span many domains. Yet, using these patterns to model the same problem in different contexts, if it is possible, is not as easy as it should be. As a result, software developers frequently prefer to start their analysis from scratch.

Building analysis patterns with stability in mind will help in producing effective and reusable patterns. These patterns can be used to model the problem whenever the problem appears, independent of the context of the problem. In this chapter, we introduce a novel pattern language for building stable analysis patterns. The objective of this language is to propose a way of achieving stability while constructing analysis patterns.

5.2 Pattern Language Overview

The proposed pattern language contains eight patterns [14]. These patterns document the steps required for building stable analysis patterns. The eight patterns are categorized into three main levels. Each level has its own main objective. Figure 5.1 shows the three levels of the pattern language and the corresponding patterns in each level. Each pattern has two digits number. The first digit shows the level of the pattern, and the second digit gives the pattern's number.

The first level is the "*Concept Patterns"* level. Patterns in this level provide the main concepts required in order to understand the rest of the pattern language. The Concept Patterns level contains two patterns: the *Efficient Usable Analysis Models (1.1),* which describes the main essential properties required for building efficient and usable analysis models, and the *Software Stability Model (1.2),* which provides the fundamental concepts of the software stability model as a solution for achieving stability.

The second level is the *"Problem Analysis Patterns"* level. Patterns in this level show how to analyze the problem that needs to be modeled. The analysis of the problem can be done through four main steps, each step is documented as a pattern. First, *Identify The Problem (2.3).* Second, *Identify Enduring Business Themes (2.4).* Third, *Identify Business Objects (2.5).* Fourth, put these elements into the *Proper Abstraction Level (2.6).*

The third level is the "Building-Process Patterns" level. After analyzing the problem in the previous level, we need to know how to *Build Stable Analysis Patterns (3.7)*, and how to *Use Stable Analysis Patterns (3.8).* Table 5.1 summarizes the patterns language for quick reference. The relationship among the language patterns is shown in Figure 5.2.

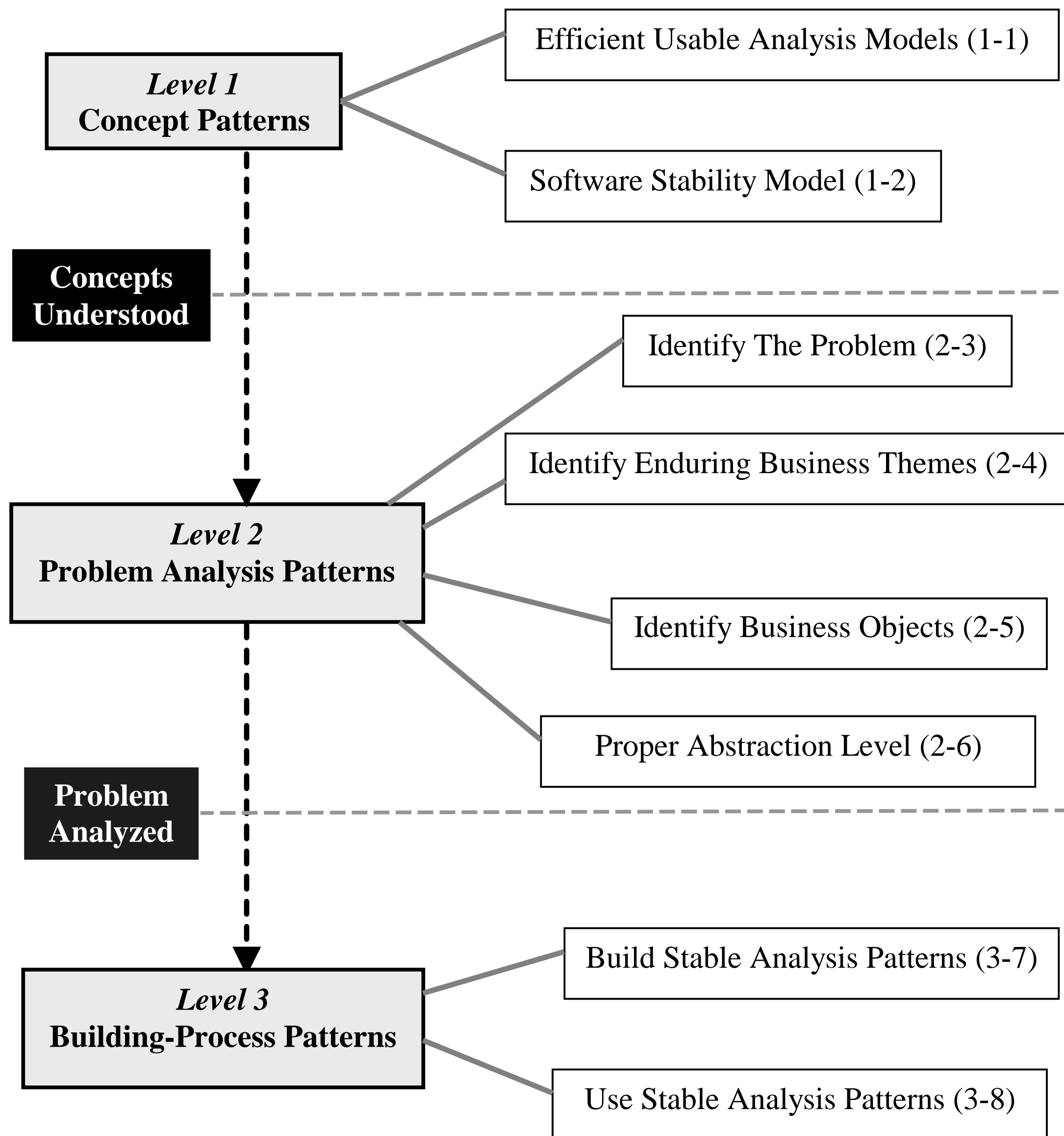


Figure 5.1: Description of the pattern language.

Pattern	**Problem**	**Solution**
Efficient Usable Analysis Models (1-1)	What are the main essential properties that affect the usability and the effectiveness of analysis models?	Page 76
Software Stability Model (1-2)	How to accomplish software stability?	Page 77
Identify The Problem (2-3)	How to focus on a specific problem that the analysis pattern will model?	Page 82
Identify Enduring Business Themes (2-4)	How to identify the enduring business themes of the problem?	Page 86
Identify Business Objects (2-5)	How to identify the business objects of the problem?	Page 91
Proper Abstraction Level (2-6)	How to achieve the proper abstraction level?	Page 98
Build Stable Analysis Patterns (3-7)	How to assemble the problem model components to build the stable analysis pattern? How to define the relations between the identified EBTs and BOs?	Page 104
Use Stable Analysis Patterns (3-8)	How to use the constructed stable analysis pattern to model the problem within a specific context?	Page 108

Table 5.1: Summary of the patterns language.

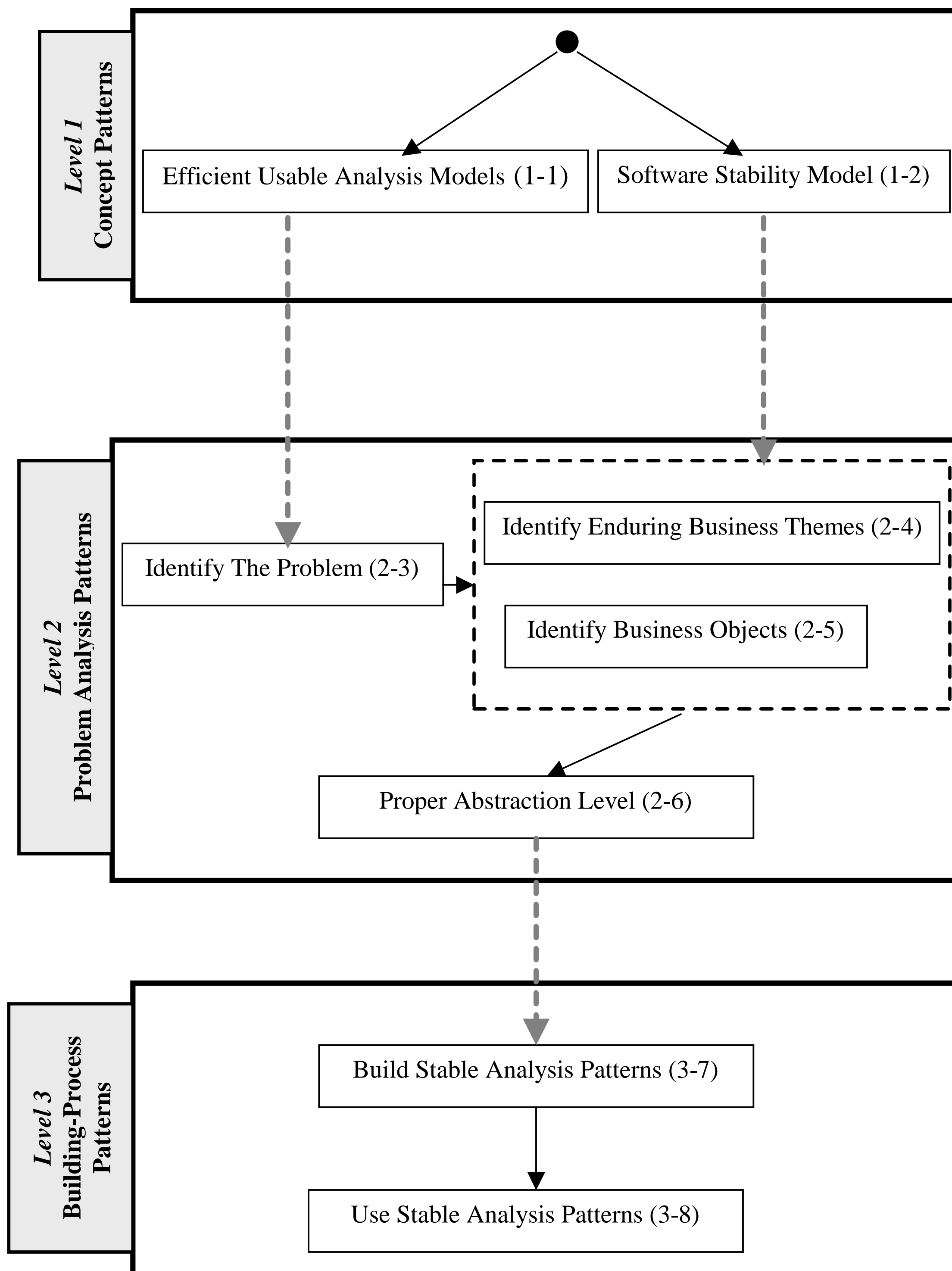


Figure 5.2**:** The pattern language chart

5.3 Building Stable Analysis Pattern Language Description

***Level 1* Concept Patterns:** First, understand the main properties of the Efficient *Usable Analysis Models (1.1)*. Second, understand the basic concepts of *Software Stability Model (1.2).*

5.3.1 Pattern 1.1 – Efficient Usable Analysis Models

Intent

Presents the essential properties of efficient and usable analysis models.

Context

Analysis is the first step to solving any problem. Some essential properties are needed for analysis models in order to be useful and usable. Keeping these properties in mind while building analysis models will tremendously improve the quality of these models.

Problem

What are the essential properties that affect the usability and the effectiveness of analysis models?

Forces

- As models, analysis models must satisfy the six basic model properties introduced in [20]. However, due the nature of analysis models, these six properties are insufficient.
- There are some misconceptions in the understanding of some of the models' essential properties. For, instance, for the model to be "easy to understand" is not the same as to be "simple". Many analysis models are easy to understand;

yet, most of them are not simple. Such confusion tremendously affects the resultant models.

Solution

Analysis patterns must satisfy the six introduced in [20]. That is, to be simple, complete and most probably accurate, testable, stable, to have visual representation, and to be easily understood. In addition to these six properties, reusable artifacts must satisfy two additional properties: first to be general, and second to be easily and actually reused. Thus, a pattern that models a specific problem should be constructed so that it is easily reused whenever the problem occurs, and independent of the context in which the problem appears.

Next Pattern

After learning the essential properties that influence the usability and effectiveness of analysis models, we need now to understand the basic concepts of the *"Software Stability Model"*.

5.3.2 Pattern 1.2 – Software Stability Model

Intent

Describes the structure of the software stability model and its basic concepts (Enduring Business Themes "EBTs", Business Objects "BOs", and Industrial Objects "IOs"). It shows the relationships between these elements and how they work together to build stable models.

Context

Stability is a highly desired feature for any engineering system. In software engineering, having stable analysis models, stable design models, stable software architectures, stable patterns etc., will definitely reduce the cost and improve the quality of software engineering products.

Problem

How to accomplish software stability?

Forces

- It is usually hard to separate analysis, design, and implementation issues while modeling the problem. Usually, analysts analyze the problem (problem domain) with some design issues in mind (solution space). Due to the fact that different solutions can exist for the same problem, mixing the modeling of the problem with its solution issues will affect the reusability of this model. As a result, people who want to approach the same problem with different solutions will need to remodel the problem from scratch. As a result, many analysis models will lack stability.
- Analysis models are required to capture the core knowledge of the problem they model.

Solution

Figure 1 shows the architecture of the Software Stability Model (SSM). In the SSM, the model of the system is viewed as three layers: the Enduring Business Themes (EBTs) layer, the Business Objects (BOs) layer, and the Industrial Objects (IOs) layer [17, 19].

Each class in the system is classified into one of these three layers according to its nature.

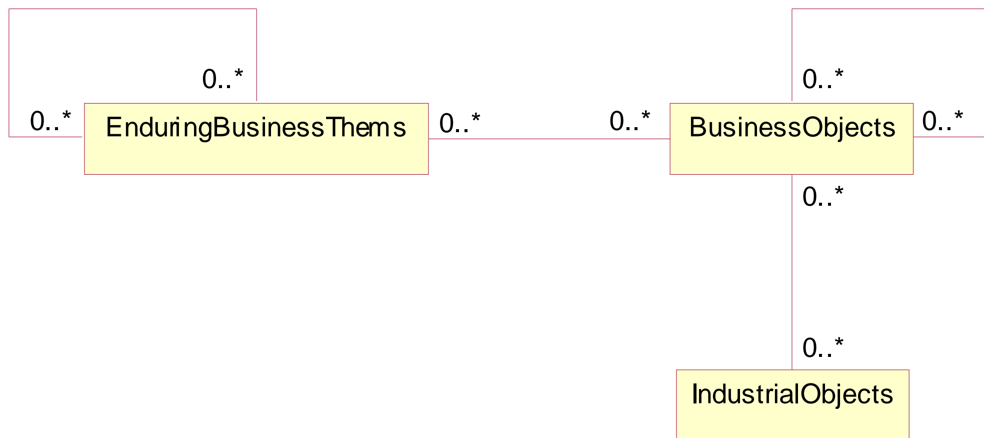


Figure 5.3**:** The relation between SSM elements

Example

This example shows how to apply software stability concepts in the modeling of a simple problem.

Problem: Model the business of a small shipping company. The company provides several different shipping services.

Solution: We will model the problem using software stability concepts. First, we need to identify the EBTs, BOs, and IOs of this problem to build its stable model. Possible EBTs in this problem are:

- "Transportation", which represents the main purpose of this business.
- "Scheduling", which shows how to manage the shipping dates, destinations, etc.

These EBTs are very abstract. However, they describe the reasons for the existence of the system. To make these abstract and intangible objects more tangible, we need to map them into more tangible objects. Identifying the BOs of the problem will do this mapping.

One Possible BO in this problem is “Schedule”. This BO will map the EBT “Scheduling” into more tangible object “Schedule”. The BO is externally stable, however, it can change internally. For instance, schedule is changing from one day to another; however, such changes will not affect the existence of the schedule as an object in the problem model.

Now, we need more concrete objects that can physically map the “BOs” into fully tangible objects. These objects are the IOs of the system. In this problem, we can think about different implementations for the BO “Schedule”. For instance, we can make “Schedule” using a piece of “Paper”, using some sophisticated “Software” program, or using both of them. By modeling the problem using the stability concepts, such changes will never affect the core of our model.

Figure 5.4 shows how the EBT “Scheduling” can be mapped into the more tangible object (BO) “Schedule”, and finally into concrete objects (IO) “Paper” and “Software”.

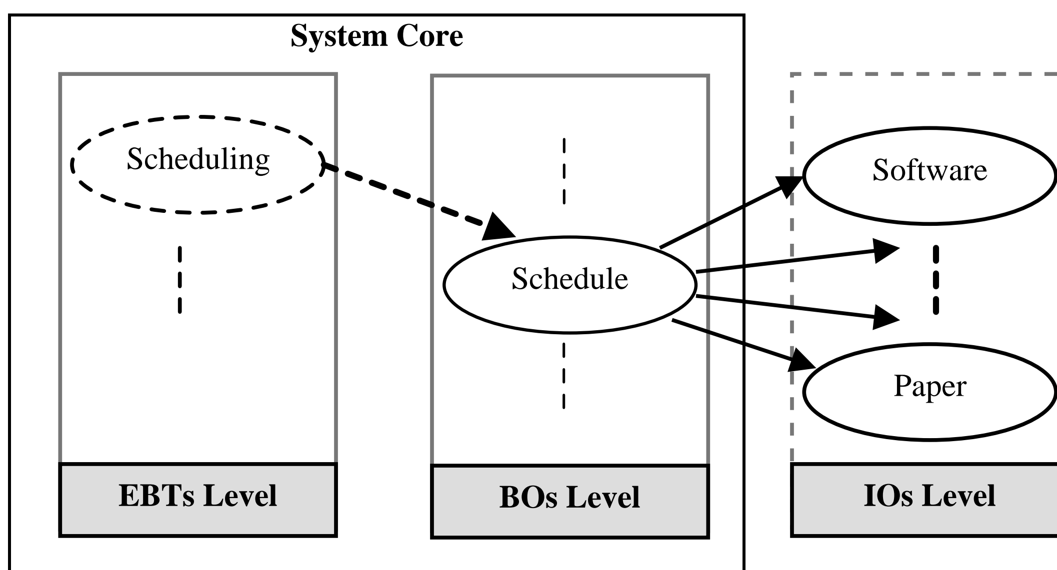


Figure 5.4: Example of the mapping between EBTs, BOs, and IOs.

Next Pattern

After understanding the required concepts, the next step is to analyze the problem. To analyze the problem we need to *"Identify The Problem"*, *"Identify Enduring Business Themes"*, *"Identify Business Objects"*, and to accomplish the *"Proper Abstraction Level"*.

***<u>Level 2</u>* Problem Analysis Patterns:** First, *Identify The Problem (2.3)*. Then *Identify Enduring Business Themes (2.4)*. After that, *Identify Business Objects (2.5)*. Finally, put the identified EBTs and BOs in the Proper Abstraction Level (2.6).

5.3.3 Pattern 2.3 – Identify The Problem

Intent

Shows how to focus on the problem we need to analyze.

Context

Analysis patterns reusability has a direct relationship with the number of problems they model. If a pattern is used to model many problems, the generality of the resulting pattern is sacrificed, since the probability of the occurrence of all the problems together is less than the probability of the occurrence of each problem individually. Focusing on a specific problem is one of the key factors that help improve the reusability of the pattern.

Problem

How to focus on a specific problem that the analysis pattern will model?

Forces

- Many problems appear together frequently in many contexts. As a result, they will be modeled as one problem.
- Sometimes it is hard to separate small problems from a bigger problem.

- In reality, not all of the small problems that we can separate are qualified to form practical stand-alone problems.

Solution

Before we start modeling the problem we need to check whether or not this problem can be further divided into smaller, real problems. The following questions help to do so: “What is the problem that we need to solve?” “Can we divide this problem further into a list of smaller problems?” “Are there any known possible scenarios where these smaller problems can appear?”

If we can find practical scenarios for each of the smaller problems we have separated, then we need to model each of them separately. If the smaller problems have no practical use, they should be modeled together.

Example

We choose a simple example in order to illustrate the idea of problem separation. We consider the “account” problem. In fact, it was not too long ago when the word “account” was merely used to indicate banking and financial accounts. Today, the word “account” alone becomes a vague concept if it is not allied with a word related to a certain context. For instance, besides all of the traditional well-known business and banking accounts, today we have e-mail accounts, on-line shopping accounts, on-line learning accounts, subscription accounts, and many others.

One possible model for the account problem is the *Account* pattern provided by Fowler [23]. Figure 5.5 shows the class diagram of the *Account Pattern*. This pattern models two different problems at the same time. The first problem is the “account” problem and the second problem is the “entry” problem. These are two independent problems. Even

though they appear together in many contexts, there is now a possibility of having entries without an account, or accounts without entries. Figure 5.6 shows some examples of accounts without entries, while Figure 5.7 gives an example of entries without accounts. As a result, the generality of the pattern is limited. These factors contribute negatively to the reusability of the pattern.

The stable model of the "Account" problem will be built step by step, through the remainder of this chapter. From this point forward, we will show how each pattern contributes to the building of the stable analysis pattern that models the "Account" problem.

Figure 5.5: *Account* pattern provided by Martin [23].

(1) Free on-line services account: Today, there are many on-line companies that provide free goods or services. For example, some companies provide learning software packages, or learning documents. In order to access these materials, these providers require you to create an account with the company. This account is simply a passport provided to enable you to access their service; you do not have anything in this account that can be considered to be your property. In fact, the only things that you can do with this account are the limited functions prescribed by the company that issued the account.

(2) Access account to the copy machine: Suppose that you have an account to access the copy machine in your school or work. This account is no more than a passport for you for using the copier. There are no entries in this case. (Note that in this example it is possible to use Martin's pattern by changing the names of the behaviors in his patterns. In fact, all the behaviors in Martin's pattern are not relevant in this case).

Figure 5.6: Examples of accounts without entries

The following table contains information about class schedules, at the University of Nebraska-Lincoln, Spring 2002. In this table, each piece of information forms an entry to the table. Here we do not need accounts to keep this information in.

Call #	Course Title	Course #	Cr Hrs	Sec.	Time	Day	Room
2850/2867	Computer Architecture	430/830	003	001	0230-0320p	M W F	Freg 112
2855/2873	Software Engineering	461/861	003	001	0930-1045p	T R	Freg 111

Figure 5.7: Example of entries without accounts

Next Pattern

After we have identified the specific problem to model, now we need to *"Identify Enduring Business Themes"* of this problem.

5.3.4 Pattern 2.4 – Identify Enduring Business Themes

Intent

Shows how to identify the Enduring Business Themes of the problem.

Context

When using software stability concepts in the modeling of the problem, first, identify the core elements of the problem. These are the enduring concepts of the problem.

Problem

How to identify the enduring business themes of the problem?

Forces

- EBTs should capture the core knowledge of the problem, however, some EBTs capture the core knowledge of the problem within a specific context. Such EBTs should be discarded from the model.
- Unfortunately, experience with the domain is not always an accurate generator for the relevant EBTs.
- Even though many of the selected EBTs might appear strongly related to the problem at first glance, many of them in fact have nothing to do with the problem being modeled.
- EBTs of the problem should be as small as possible. Extracting the EBTs that have a real relation to the problem is usually hard.
- Some of the EBTs might lack one or more of the EBTs essential properties. In this case, we should re-identify them as BOs or IOs.

Solution

One approach that helps to extract the appropriate EBTs of the problem is to follow these three steps:

Step 1 Create Initial EBTs List In order to create the initial list of the EBTs of the problem, answer the question: "What is the "problem" for?" In other words: "What are the reasons for the existence of the "problem"?".

The output of this step is the list of the initial EBTs of the problem. These EBTs are still tentative and some of them are not as strongly related to the problem as they might appear.

Step 2 Filter the EBTs List This step is to eliminate the redundant and irrelevant EBTs from the initial list. This step is important due to the fact that people usually construct the initial EBTs list with specific context in mind, even if they do not intend to do so. The output of this step is a modified EBTs list, which is usually smaller than the initial list.

Step 3 Check the Main EBTs Properties This step is to examine the EBTs obtained in previous steps against the main essential properties of the EBTs. The typical procedure is to answer the following questions for each EBT in the list "The desired answer is written in **bold** beside each question":

- Can we replace this EBT with another one? **No.**
- Is this EBT stable internally and externally? In other words, does this EBT reflect the core knowledge of the problem we are trying to model? **Yes.**
- Does this EBT belong to a specific domain? **No.**

- Can we directly represent this EBT physically? **No.**

  It is important to note that the EBTs should not have direct physical representations (IO); otherwise they should be considered BOs. (Refer to the software stability model architecture shown in Figure 3). For example: "Agreement" is a concept and one can see it as an EBT. However, "Agreement" also has a direct physical representation (for instance "Contract"). Therefore, "Agreement" is not an EBT, it is a BO.

Any EBT that does not satisfy one of these properties should be eliminated from the list. Figure 5.8 summarizes the three steps needed to identify the EBTs of the problem.

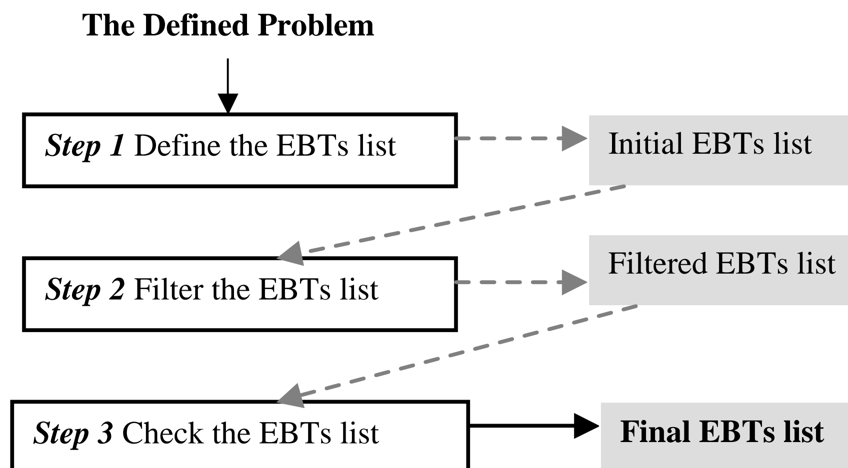


Figure 5.8: The steps for identifying the EBTs of the problem.

Example

This example shows the identification steps for the EBTs of the “account” problem:

Step 1 Create Initial EBTs List

We need to answer the question: “What is the “Account” for?”

The initial EBTs list might contain the following EBTs:

- Storage
- Ownership
- Tractability
- Recording

Step 2 Filter the EBTs List

In this step we need to extract the most appropriate EBTs for the “account” problem. To do so, we need to examine each EBT in the list and see whether or not it reflects the enduring concepts of the problem we model.

Since our focus is on the modeling of the “account” problem alone, we realized that most of the EBTs we have defined are related to accounts that have entries. For instance, “Storage”, “Tractability”, and “Recording” are all concepts that are dependent upon the existence of entries within the account. If the account has no entries, “Storage”, “Tractability” and “Recording” are unneeded. Therefore, we need to eliminate the EBTs “Storage”, “Tractability”, and “Recording” from the list. Now, we have only one EBT, “Ownership”, instead of four.

Step 3 Check the Main EBTs Properties

- Can we replace the "Ownership" with other EBT? No.
- Is "Ownership" stable internally and externally? Yes.
- Does "Ownership" belong to a specific application or domain? No.
- Can we have direct physical representation for "Ownership"? No.

Discussion

1. There are different approaches that can be used to extract the EBTs of the problem, however, our experience shows that the shown approach is effective.
2. Debate and discussion are two important factors that definitely enhance the extracted EBTs.
3. For small problems it is desired to narrow the number of EBTs to one or two. This usually makes the pattern more focused and more effective.
4. The final EBTs list obtained here is not necessary the final EBTs that will appear in the pattern structure. These EBTs are usually enhanced and adapted in later steps.

Next Pattern

After we identify the EBTs of the problem, now we need to *"Identify Business Objects"* of the problem.

5.3.5 Pattern 2.5 – Identify Business Objects

Intent

Shows how to identify the Business Objects of the problem.

Context

Having used software stability concepts in the modeling of the problem to identify the EBTs, next the business objects are identified.

Problem

How to identify the business objects of the problem?

Forces

- In some cases, it is not obvious whether the object is an EBT or BO. For instance, “Agreement” can be considered as an EBT since it presents a concept. However, it is a BO.
- After the EBTs of the problem have been identified, the conceptualization becomes more involved since the BOs of the problem must be based on the defined EBTs. This makes the extracting of the BOs difficult.
- Usually there is no one to one mapping between the EBTs of the problem and its BOs. It is possible for EBTs to have no direct mapping to the BOs and for the BOs to has no direct mapping to the EBTs. Moreover, one EBT can be mapped into several BOs.
- Besides the main BOs that we can identify for the problem, it is possible to have some “hidden” BOs. These hidden BOs have no direct relationship with the

defined EBTs. Instead, they are related to the main BOs and to the other hidden BOs in the problem.

Solution

One approach that helps to extract the appropriate BOs of the problem is to follow the following four steps:

Step 1 Identify the main BO of the problem In this step we identify the main set of BOs that are directly related to each of the EBTs we have in the problem. There could be one or more BOs corresponding to each EBT in the problem. However, some of the EBTs may have no corresponding BOs.

The main set of BOs of the problem can by identified by answering one or more of the following questions for each EBT we have:

[Note: some questions do not apply for some of the EBTs. This depends on the nature of each EBT]

- How can we approach the goal that this EBT presents?

  [For example: To achieve the goal of the EBT "Scheduling" or "Organization", we can use, the BO "Schedule". Another example: for the EBT "Negotiation", we need the BOs: "NegotiationContext", and "NegotiationMedia" to perform the "Negotiation"].

- What are the results of doing/using this EBT?

  [For example: for the EBT "Negotiation", the eventual result is to reach an "Agreement" so this is one possible BO that maps this EBT].

- Who should do/use this EBT?

  [For example: The BO "Party" uses/ does "Negotiation". This "Party" can be a person, a company, or an organization. Therefore, "Party" is one possible BO that maps the EBT "Negotiation"].

Step 2 Filter the main BOs List This step is to purify the main BOs identified in the previous step. The objective of this step is to eliminate the redundant and irrelevant BOs from the initial list. One way to achieve this goal is to debate the listed BOs with a group.

Step 3 Identify the hidden BOs of the problem This step is to identify the hidden BOs of the problem. The name hidden comes from the fact that these BOs have no direct relationships with any of the EBTs of the problem. Thus, we cannot extract them in the first two steps we have performed.

For example, suppose we need to model a simple transportation system that offers transportation services for different types of materials (Gas, water, etc.). One possible EBT is "Transportation". One possible BO that maps this EBT is "Transport". A possible IO that can physically represent this BO is "Trucks". In this problem, one possible hidden BO is the BO "Materials". We do not have a direct EBT that the BO "Materials" can be mapped to, however, there is a clear relationship between the two BOs "Transport" and "Materials".

Before thinking about the hidden BOs in the problem, visualize a provisional scenario for each EBT and its corresponding BOs. Then answer the question "What is still missing in the problem?" Usually the answer to this question is the list of the hidden BOs of the

problem. Some problems do not have any hidden BOs, especially in the case of the small-scale problems.

Step 4 Check the characteristics of the BOs: This step is to make sure that the identified BOs satisfy the main BOs characteristics.

The main BO characteristics are summarize below:

- Business Objects are partially tangible.
- Business Objects are externally stable, and they should remain stable throughout the life of the problem.
- Business Objects are adaptable; thus, they might change internally.
- Business Objects can have direct physical representation (IOs).

Figure 5.9 summarizes the three steps needed to identify the BOs of the problem.

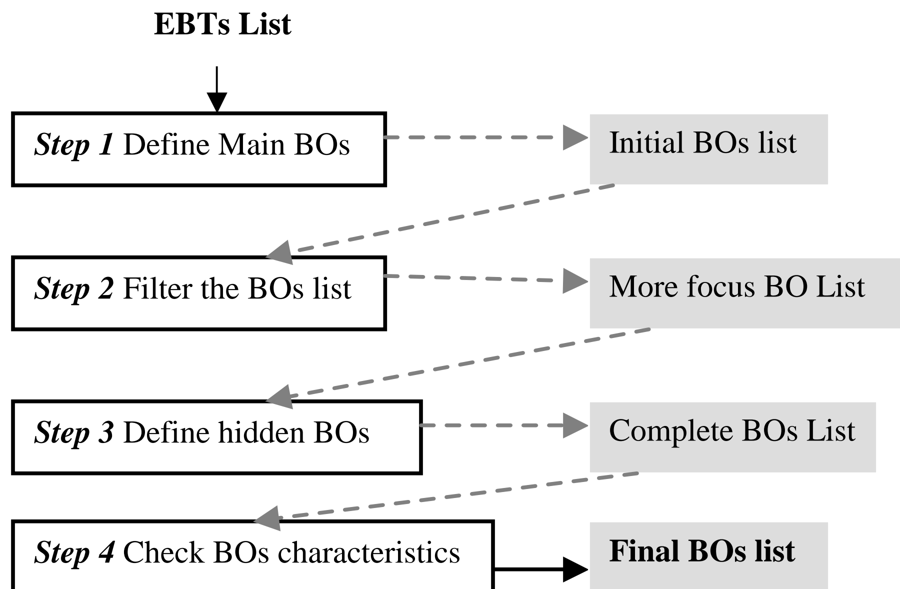


Figure 5.9: The steps for identifying the BOs of the problem.

Example

In this example we need to identify the possible BOs for the "account" problem. In order to identify these BOs, we will apply the four steps we have proposed. The input of the first step is the EBTs list that contains one EBT, "Ownership".

Step 1 Identify the main BO of the problem:

- How can we approach the goal of "Ownership"?

  In the account problem, to achieve "Ownership" we need to have something to own. The "Account" itself is what makes the meaning of "Ownership".

- What are the results of doing/using "Ownership"?

  By having "Ownership" we have "Privacy", but this is not a BO, it is a redundant EBT, so we exclude it. "Agreement" is a possible BO for the problem, when you own an account, you have to agree with its policy.

- Who should do/use "Ownership"?

  For the "Owner" to be able to the use the "Account", he or she should follow the responsibilities defined by the "Ownership" policy.

<u>BOs main list:</u> "Owner", "Account", and "Agreement".

Step 2 Filter the main BOs List

Now, the following BOs remain: "Owner", "Account", and "Agreement". While modeling this problem, we have debated the accuracy of using the BO "Agreement" in our model. "Agreement" is a general term that appears in many contexts. For instance, in the "negotiation" problem, we will need the BO "Agreement". Therefore, having the BO "Agreement" as part of the "Account" pattern will counter the simplicity property of the

pattern. "Agreement" is a stand-alone problem that occurs in many contexts, and hence it is more appropriate to model the "Agreement" problem independent of the context of this problem. Therefore, we have excluded the BO "Agreement" from the main list.

The filtered BOs list contains: "Account" and "Owner".

Step 3 Identify the hidden BOs of the problem

After identifying and filtering the main BOs list, now answer the question "Does the account problem need anything else to be complete?"

We have an "Account", its "Owner", and the concept of "Ownership" that regulates the responsibilities and benefits of the "Owner". This is all that is needed to model any basic account.

Step 4 Check the characteristics of the BOs

- Business Object should be partially tangible. "Owner" and "Account" are partially tangible.
- Business Objects are externally stable, and they should be so throughout the life of the problem. "Owner" and "Account" are always stable. In other words, we cannot have any account without having these two objects in its model
- Business Objects are adaptable, thus, they might change internally. "Account" and "Owner" might change internally. For instance, you can add or remove some feature from your banking "Account" (adding the overdraft protection service for example); however, this is an internal change inside the account. Externally, there is nothing that has changed. Also, for the BO

"Owner", the "Owner" may become ill, for example, however, he is still the owner of the account.

- Business Objects can have direct physical representations (IOs). There are different possible physical representations for the BO "Account" depending on its context. The "Account" could be, physically, a code as in the case of the copy machine. The BO "Owner" is physically the person who owns this account and uses it.

Discussion

1. There are different approaches that can be used to extract the EBTs of the problem, however, our experience shows that the shown approach is effective.
2. For small problems it is desirable to narrow the number of BOs. Usually, for small size problems we have 2 to 4 focused BOs.
3. As for the EBTs, the final BOs list we have defined using this pattern can be further enhanced and modified during the next stages.

Next Pattern

After we have identified the EBTs and the BOs of the problem, now we need to accomplish the *"Proper Abstraction Level"*.

5.3.6 Pattern 2.6 – Proper Abstraction Level

Intent

Helps to accomplish the proper abstraction level for the analysis pattern.

Context

If the pattern lacks the appropriate abstraction level, the reusability of this pattern becomes critical. The ultimate goal of having the proper level of abstraction is to make the pattern as useful as possible in covering all of the possible situations that might appear in the problems the pattern models.

Identifying the EBTs and BOs of the problem was the first step for achieving the proper abstraction level. During the identification of the EBTs and BOs, we have excluded the EBTs and the BOs that make the model adhere to a specific domain. However, this level of abstraction is not sufficient for our needs. Therefore, more work is needed in order to enhance the abstraction level of our model.

Problem

How to achieve the proper abstraction level?

Forces

- Usually the names of the EBTs and the BOs resulting from the previous stages are not accurate.
- Even though we can find an appropriate name for each EBT and BO in the problem, it is usually hard to define the attributes and operations for each class that can fit all the contexts that the problem might appear in.

Solution

After identifying the EBTs and the BOs of the problem, ensure that these elements have the proper abstraction level. Our approach to achieving the proper abstraction level is summarized in following points:

- We prefer not to assign specific attributes or operations for any of the EBTs or the BOs of the problem, even though some may argue that by not doing so we might impose more difficulty in the use of the pattern. We believe that assigning specific attributes and operations can cause confusion rather than help in understanding the model.
- We need to inspect both the names and the structure of each EBT and BO we have identified in the problem. In this step, it is most likely that some BO names will need to be changed. One way of conducting such an inspection is to examine the EBTs and the BOs of the problem against different situations that usually appear in the context of the problem we are modeling. Also, thinking about exceptional situations and examining whether or not the identified EBTs and BOs will handle them will also help to further abstract the pattern and make it more general. This will be illustrated with an example, shortly.
- If there are some situations that the identified EBTs and BOs cannot handle by changing their names or modify their structure, the problem that we are trying to solve is either too big or too small. Therefore, we will need to redefine the problem.

Example

Consider the "account" problem again. So far, we have identified the following EBTs and BOs:

EBTs: "Ownership"

BOs: "Owner" and "AnyAccount".

At first glance, the name of the BO "Owner" seems very appropriate. However, thinking deeper about some of the possible situations that might occur in any account context, a problematic situation arises. What if there are many users sharing the same account. For instance, in credit card accounts, one individual could be the owner of the account, however, that individual can allow other users to use the credit card account with specific privileges. In this case, using the BO "Owner" limits the pattern to the specific accounts where there is only one possible user who can use the account, the owner of the account.

Now, it is obvious that the BO "Owner" lacks the appropriate level of abstraction that helps to handle the different situations of the problem. Therefore, we need to redefine the BO "Owner" in such a way as to make it more general.

As a solution to the problem, we changed the name of the BO "Owner" to the name "Holder", which is more general. Then, we change the inheritance structure to capture the different roles of the different account holders. Figure 5.10 shows the detailed steps taken while contemplating the problem.

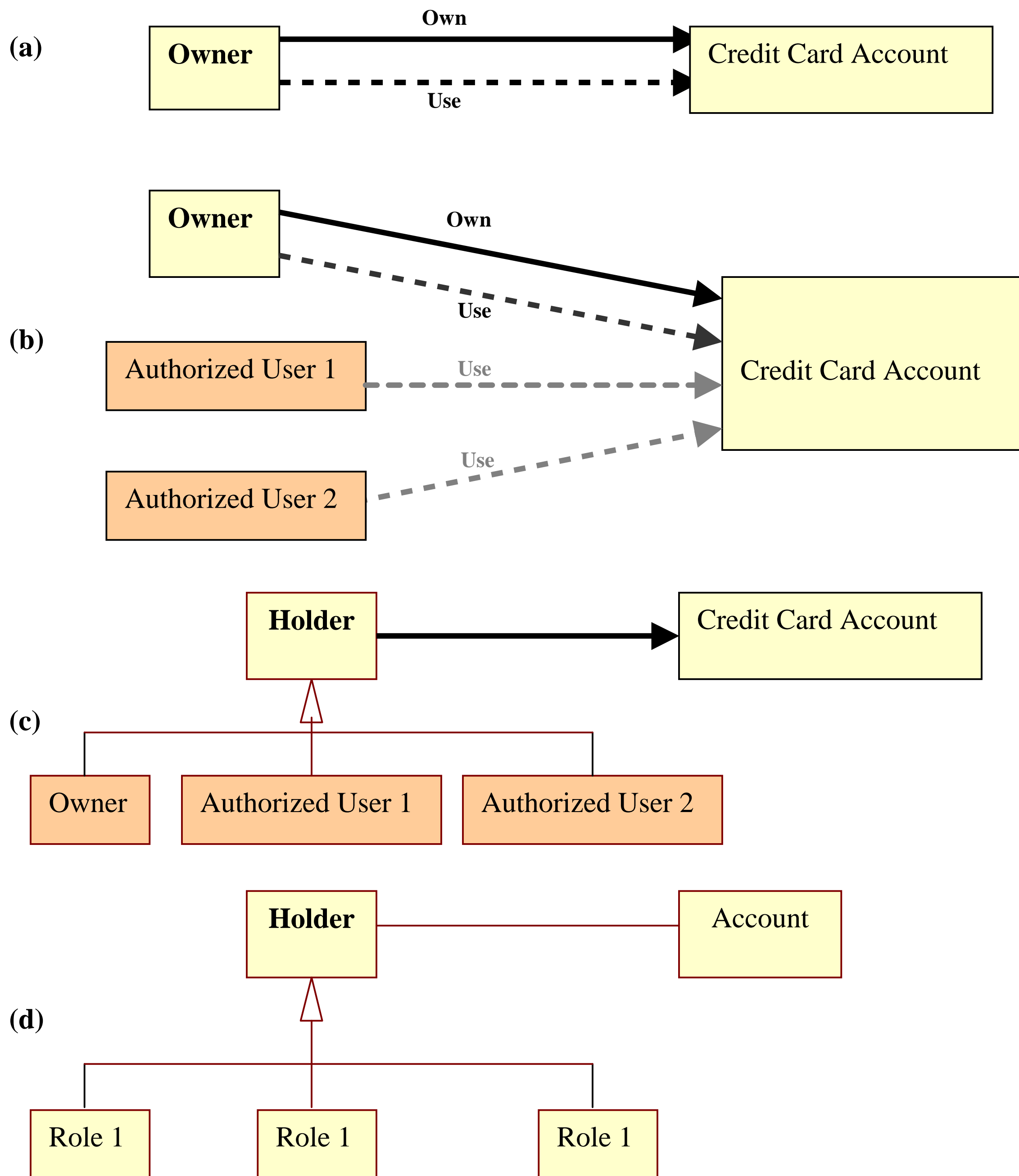


Figure 5.10: Proper abstraction level. **(a)** Shows the classic relationship between the "Owner" and his "Account". **(b)** Shows the possible situation that cannot be handled with the current representation. **(c)** Shows possible solution to handle the problem. **(d)** Gives the final abstraction level that handles the problem.

Discussion

The level of abstraction determines how broadly the pattern can be used. However, too much abstraction can be a negative. There is a trade off between flexibility and reusability of the model on one hand, and complexity of understanding and reuse of the model on the other. The level of abstraction plays a major role in this equation. A low level of abstraction will reduce the reusability of the model; adapting it will no longer be an easy task. However, the resulting model will be relatively simple when compared to the same model designed with a higher level of abstraction. Conversely, too much abstraction will lead to models that are too complex. Therefore, the optimum solution is a conceptual model that is simple enough to be easily used and understood, but with a level of abstraction high enough to make it general and hence more reusable. Several different approaches currently exist to handle the abstraction problem. Some of these views are illustrated below:

- Some people who extract analysis patterns from several projects they have experienced, believe that it is unsafe to further abstract patterns generated within certain projects in order to make them reusable in other contexts. The *Account* pattern shown in Figure 6 shows an example of this approach. Since no one can be an expert in all fields, domain expert analysts often extract domain specific patterns, even if the problem they model occurs in many other contexts. For example, following this approach we might end up having a list of patterns that models the account problem within different contexts, for instance, account patterns that models banking accounts, account patterns that models web applications accounts, and so on. It will be more efficient if we have one pattern

called *"AnyAccount"* that can capture the core structure of the different account types, hence, we can use this pattern whenever we need to model the account problem regardless of the context of this account.

- Analogy is another view for the abstraction problem. According to this approach, patterns that model complete systems in one context are reused by making an analogy between the pattern and the new application. Thus, by analogy, they change the names of the pattern's classes to be relevant to the new application. Sometimes, you might have to add/remove a few classes to adapt the pattern to the new application. This approach to the task of accomplishing abstraction levels results in the building of templates instead of patterns

Next Pattern

After we have identified the problem, its Enduring Business Themes, its Business Objects, and the appropriate abstraction level, the next step is to *"Build Stable Analysis Patterns"*, and to *"Use Stable Analysis Patterns"*.

***Level 3* Building-Process Patterns:** In this level, first, Build Stable Analysis Patterns (3-7)**,** and then Use Stable Analysis Patterns (3-8).

5.3.7 Pattern 3.7 – Build Stable Analysis Patterns

Intent

After defining the problem, its EBTs, BOs, and their proper abstraction level, now we need to glue things together to build the stable analysis pattern that models the problem.

Context

After focusing on a specific problem and defining all the stability model elements of this problem, we now need to understand how to identify the relationship between these elements.

Problem

How to assemble the problem model components to build the stable analysis pattern? How to define the relations between the identified EBTs and BOs?

Forces

- Abstraction level of the pattern elements imposes difficulty in defining the different relationships between these elements.
- We have some EBTs that have no corresponding BOs.
- We need to identify the relations between the main and the hidden BOs of the problem.

Solution

One way of identifying the different relationships between the EBTs and the BOs of the problem is to put the pattern into a context so we can visualize these relationships. However, in order to insure the generality of the pattern, we need to visualize the pattern in different contexts. By doing so we can increase the accuracy of defining the relationships and the multiplicities between the elements of the problem.

One possible way for identifying the relationships between the different EBTs and BOs of the problem is as follows:

First join each EBT in the EBTs list with its corresponding BOs in the main BOs list. Second, define the relationships between the EBTs of the problem. Finally, define the relationships between the main BOs of the problem, and between the main BOs and the "Hidden" BOs of the problem. Figure 5.11 shows the relationship between the EBTs and the BOs of the problem.

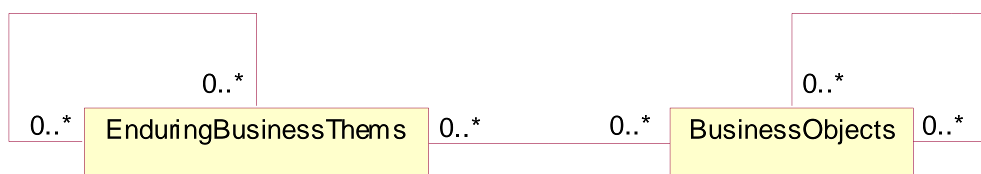


Figure 5.11: The relationship between the EBTs and the BOs of the problem

Examples

Example 1. Building "AnyAccount" Pattern

This example shows how to join the EBTs and the BOs of the "account" problem to build the stable analysis pattern *"AnyAccount"*. Below is the summary of the information we have, so far, regarding the "account" problem:

Element	Description
Problem Definition	Modeling any account for any context
EBTs List	"Ownership".
Main BOs List	"Holder", and "AnyAccount".
Hidden BOs List	None

Define the relationships between the EBTs and the corresponding main BOs:

What is the relationship between the EBT "Ownership" and the BO "Holder"?

What is the relationship between the EBT "Ownership" and the BO "AnyAccount"?

"Ownership" presents the policy of owning the account, and it can regulate from 'zero' to 'many' accounts. For instance in banking accounts, we have a predefined policy that regulates all checking accounts, and another that regulates all saving accounts, and so on. Since parts of this policy regulate the responsibilities and the benefits of the account "Holder", we have an association between the EBT "Ownership" and the BO "Holder".

Define the relationships between the EBTs of the problem:

There are no other EBTs in this problem, so we move to the last step:

Define the relationships between the BOs of the problem:

What is the relationship between the BO "Holder" and the BO "AnyAccount"? The relationship is clear, the "Holder" uses the "AnyAccount", and hence, an association between the two classes should exist. Since the "Holder" can have as many

"AnyAccounts" as he wants the multiplicity is from 'zero' to 'many'. Also, the "AnyAccount" can belong to one or more "Holder", keeping in mind the fact that the "Holder" of the "AnyAccount" could be the owner only, or could be the owner and any other authorized holders, as in the case of credit cards accounts. Figure 5.12 gives the class diagram of the *"AnyAccount"* pattern.

Example 2. Building "AnyEntry" Pattern

By following the same steps as for the *"AnyAccount"* pattern, we have constructed another pattern called *"AnyEntry"* pattern. This pattern captures the core element of any entry independent of the context of the problem. Figure 5.13 gives the structure of the "AnyEntry" pattern.

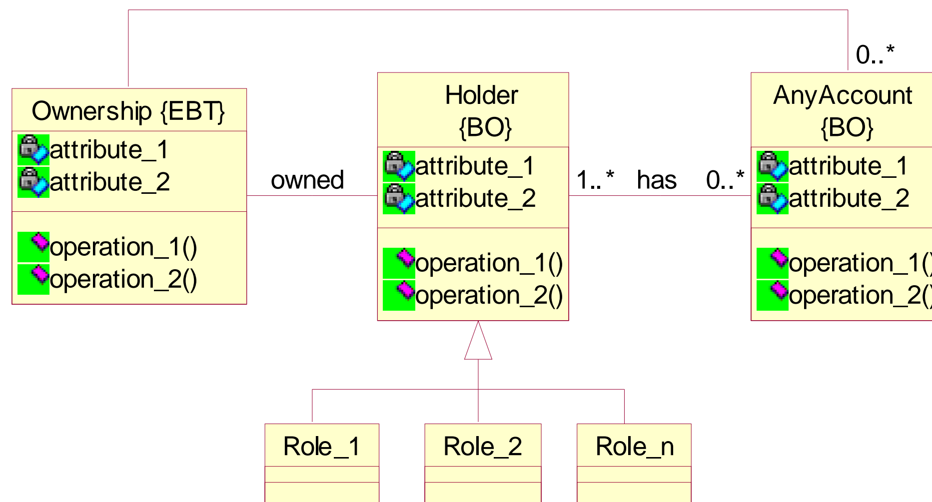


Figure 5.12: *AnyAccount* pattern class diagram

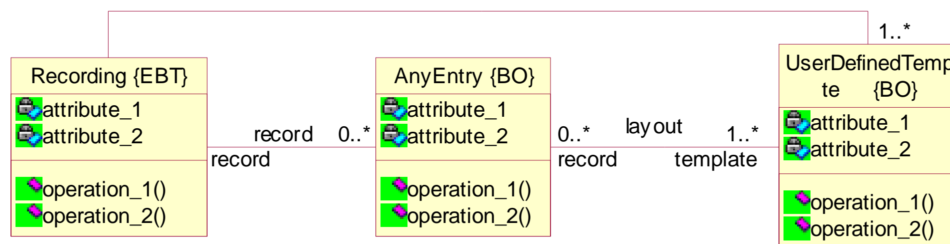


Figure 15.13: *AnyEntry* pattern class diagram

Next Pattern

After we have built the stable analysis pattern, now we need to see how to *Use Stable Analysis Patterns*.

5.3.8 Pattern 3.8 – Use Stable Analysis Patterns

Intent

After the "Build Stable Analysis Patterns" patterns are built, they should to be applied in context.

Context

After the construction of the stable analysis pattern and refinement to the proper abstraction level, this pattern needs to be used to model the problem in context.

Problem

How to use the constructed stable analysis pattern to model the problem within a specific context?

Forces

- Analysis patterns usually have high levels of abstraction, and hence, they need to be properly instantiated in order to be used within a specific context.
- In many situations, it is required to integrate multiple stable analysis patterns to model bigger problems.

Solution

In order to utilize these stable analysis patterns:

1- Choose the appropriate attributes and operations for each EBT and BO of the problem, based on the context.

2- Identify the IOs of the system based on the identified BOs. Both the main and the hidden BOs can have corresponding IOs that will physically represent them. However, in some cases there is no one to one mapping between the BOs and the IOs of the problem.

3- Identify the EBTs, BOs, and IOs for the rest of the problem or the system being modeled.

4- In the case of using more than one pattern together, we need to define the level at which the different patterns are to be connected. (Connection shall take place at the EBT level, the BO level, and/or the IOs level, depending on the problem nature.).

Comment

The names that we have chosen for each EBT and BO in the pattern will remain the same. However, in some situations we might need to modify some of the BOs names for clarity of purpose only. The names of the EBTs never change.

Example

In this section, two examples are provided to show how to use the patterns that have been developed (the "*Any Account*", and "*AnyEntry*" patterns) in specific contexts.

Example 1. Modeling Copy Machine Account

This simple problem shows how to use the *"AnyAccount"* pattern in the modeling of a simple copy machine account in one of the universities. Each student in the university has an account that he can use to access a central copy machine.

Figure 5.14 gives the object diagram of the *Copy Machine Account.* Possible IOs that map the BOs of the problem are identified. For the BO "Student", the *"AnyAccount"* pattern without the inheritance part is used, since for each account there should be only one holder. For the BO "Account", one possible physical representation is the IO "Code". Each student has a "Code" in order to use the copy machine. If there should be any other physical representations for the BO "Account", all that would need to be done is to remove the current IO "Code" and insert the new IO into the model without affecting the core. In this problem, no extra EBTs, BOs, or IOs are needed.

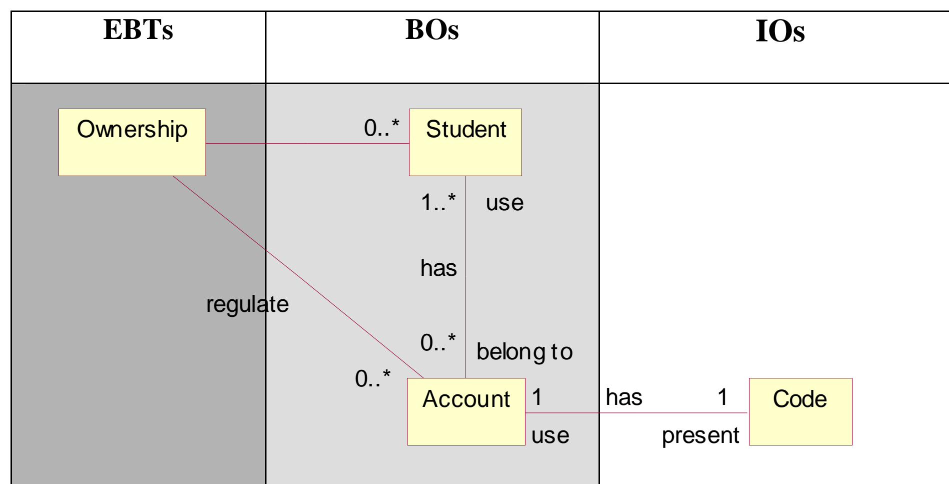


Figure 5.14: *Copy Machine Account* object diagram

Example 2. Modeling Hotmail Account

This example shows how to integrate more than one pattern to model larger problems. The aim of the problem is to utilize the two constructed patterns: the *"AnyAccount"*, and the "AnyEntry" patterns, in the modeling of a simple Hotmail Account. For simplicity, only the object diagram of the problem model is shown. Also, it is important to note that this model is not complete; it is merely for demonstration purpose. Figure 5.15 gives the object diagram of the *Hotmail Account.*

For each BO one possible physical representation "IO" is displayed. For instance, "Hotmail" is one possible physical presentation for the "Host", however, for generality, this IO can change anytime in order to represent any other hosts, without affecting the core of the model. Notice that in this example, all of the connections between the two patterns are made in the IOs level only.

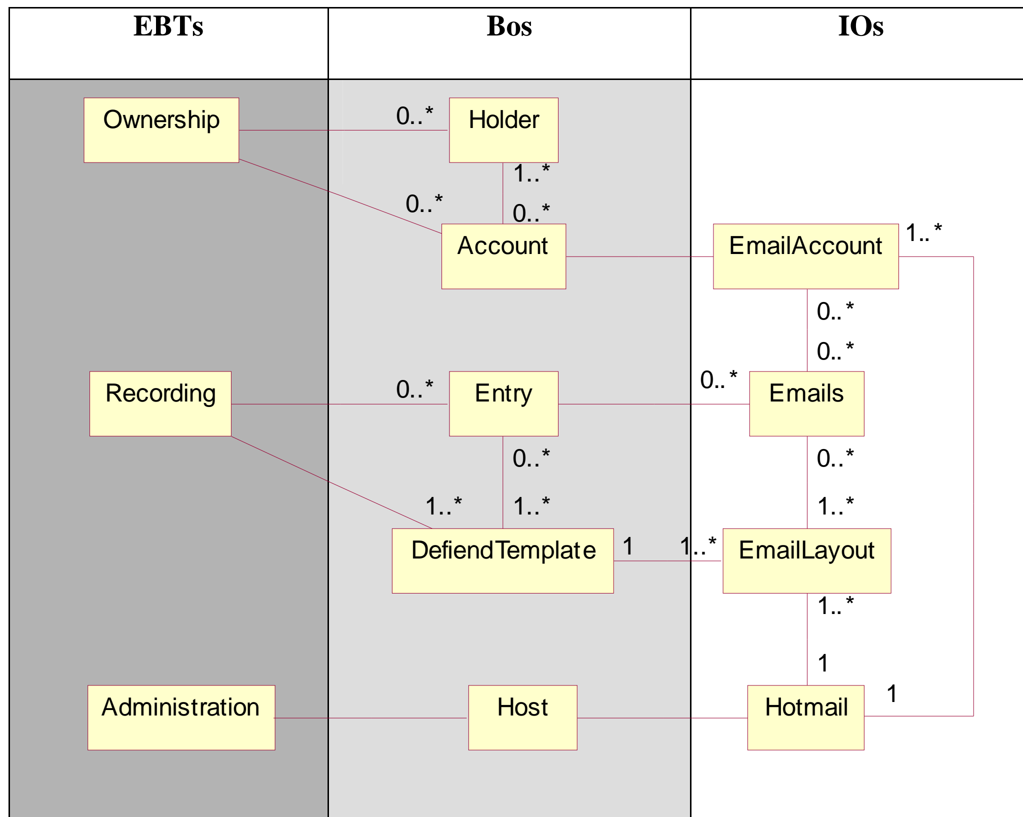


Figure 5.15. *Hotmail Account* object diagram

Chapter 6

Evaluation of Stable Analysis Patterns

6.1 Introduction

As models, analysis patterns must satisfy the six essential properties introduced in [20]. Also, as patterns, they must satisfy two additional properties: first to be general, and second to be easily and actually reused. Lacking any of these properties will vastly affect the effectiveness of the pattern, diminishing its reusability.

In this chapter, the evaluation criterion, and the eight essential properties of analysis patterns are introduced. These eight essential properties are then used to evaluate the three main groups of analysis patterns: the experience group, the analogy group, and the stable analysis patterns group, the group proposed in this thesis [12, 13].

6.2 Evaluation Criterion

In order to evaluate the effectiveness of each of the three groups of analysis patterns, we introduce eight essential properties that cover the main qualities of any analysis model [12, 13, 14]. Satisfying these eight properties does not guarantee an efficient, reusable

model. However, in practice, lacking any of these properties will conspicuously affect the reusability and the effectiveness of the model.

1. ***Simple:*** a pattern is not intended to represent a model for a complete system. Rather, it models a specific problem that commonly appears within larger systems. Systems, by their nature, combine many problems. Thus, they are modeled using a collection of analysis patterns. In fact, each analysis pattern should focus on one specific problem, otherwise, many problems arise. Without decomposing a system into components, models become unreasonably complex, the generality of the patterns are adversely affected and the model becomes highly non-intuitive. If a pattern is used to model an overly broad portion of a system, the generality of the resulting patterns is sacrificed – the maxim holds: the probability of the occurrence of all the problems together is less than the probability of the occurrence of each problem individually. For instance, modeling the "payment" problem with "buying a car" is not effective since the "payment" problem may appear in unlimited problems. Pattern completeness is also sacrificed when we model a system at an improper level of resolution. The analyst's focus is not on a specific problem and it is likely that important features of the system and its subcomponents will be overlooked.
2. ***Complete and most likely accurate:*** closely related to the concept of simplicity, this property guarantees that all the required information is present. In order to be considered complete, the model should not omit any component. The model must be able to express the essential concepts of its properties. For example, trying to model the whole rental system of any property will force us to miss some of the parts of this system. Renting a car will involve something related to insurance, which is not the

case when renting a book from a library. As a result, a pattern that models the whole rental system, besides lacking the simplicity property, will not be complete or accurate.

3. ***Testable:*** for the model to be testable, it must be specific and unambiguous. Thus, all the classes and the relationships of the model should be qualified and validated.
4. ***Stable:*** stability influences the reusability of a model. Stable models are easily adapted and modified without the risk of obsolescence.
5. ***Graphical or visual:*** conceptual models are difficult to visualize. Therefore, having a graphical representation for the model aids understanding.
6. ***Easy to understand:*** Conceptual models are complex as they represent a high level of abstraction. Therefore, it is required for analysis patterns to be well described such that they aid in communicating an understanding of the system. Otherwise use of the pattern is neither attractive nor effective.
7. ***General:*** This property is essential to ensure model reusability. Pattern models lacking generality become useless, since analysts will tend to build new models rather than spend time and effort adapting an unruly pattern to fit into an application. Generality means that a pattern that models a specific problem is easily used to model the same problem independent of context. Pattern generality may be divided into two categories: Patterns that solve problems that frequently appear in different contexts (domain-less patterns), and patterns that solve problems that frequently appear within specific contexts (domain-specific patterns). In the latter sense, the pattern is still considered to be general even if it is only applicable in a certain domain, but in this

case, we should make sure that the problem that this pattern models does not occur in other contexts.

8. ***Easy to use and reuse:*** analysis patterns should be presented in a clear way that makes them easily reused. It is important to remember that patterns are consumed in larger models. Patterns that are easy to use and designed for reuse stand a greater chance of actually being reused.

Each pattern will be examined against these eight essential properties. For each property, the pattern will be assigned one of the three measures: *Always, Sometimes, and Never,* based on how well the pattern satisfies the aspect of this property. *Always* means that when the approach of the evaluated group is used, the resultant pattern will always satisfies this property. *Sometimes* means that when the approach of the evaluated group is used, the resultant pattern will sometimes satisfy this property, although this is not guaranteed. *Never* means that when the approach of the evaluated group is used, the resultant pattern cannot satisfy this property.

6.3 Experience Group Evaluation

The *Account Pattern* provided by Fowler [22] is representative of this group. Figure 1 shows the class diagram of the *Account Pattern*. The purpose of this pattern is to provide a model for the "account" problem, thus, we can, for instance, use this pattern to model a banking account. In fact, it was not that long ago when the word account was used strictly to indicate banking and financial accounts. Today, the word account, when used alone, becomes a vague concept if it is not allied with a word that relates it to a certain context.

For instance, in addition to all of the traditional, well-known business and banking accounts, today we have e-mail accounts, on-line shopping accounts, on-line learning accounts, subscription accounts, and many others. As a result, using words such as balance and withdrawal while modeling the concept of an account makes the use of this pattern to model accounts in different contexts time and effort-consuming, if not, impossible. For instance, suppose we want to model an e-mail account using Fowler's pattern. Perhaps the most obvious changes are all the classes' behaviors, which are completely irrelevant to the email application.

Figure 6.1: *Account* pattern provided by Fowler [22].

From the point of view of simplicity, Fowler's *Account Pattern* is not considered simple, in the sense that it models two different problems at the same time. The first problem is the "account" problem and the second problem is the "entry" problem. In fact, these are two independent problems. Even though they appear together in many contexts, there is now the possibility of having entries without an account, or having an account without entries. As a result, the generality of the pattern is limited. These factors contribute negatively to the reusability of the pattern.

Fowler's pattern is not complete in the sense that it lacks some of the "basic" concepts that appear frequently in banking accounts. For instance, suppose that we need to use this pattern to model a banking account. In banking accounts it is possible that two or more persons may be holders of the same account. Perhaps, there is a primary holder that

has the full authorization to manage and control the account, while each of the other holders have specific privileges for using the account. Such situation cannot be handled while using Fowler's account. Thus, Fowler's pattern is not applicable to some of the usual financial and banking account situations. Since significant effort is required to adapt this pattern to other circumstances, the stability of the system is limited and the pattern structure is not stable over time.

This pattern is graphical in the sense that it has a graphical model that describes it (the class diagram). Having such a graphical representation will make the pattern visually testable. Table 6.1 summarizes the evaluation of this group.

Properties	Evaluation Summary
Simple	*Never.* It models two different problems.
Complete and most likely accurate	*Never.* It does not cover all the circumstances that might occur in the application.
Stable	*Never.* This pattern cannot model all types of today's accounts. Thus, we will always need to do major changes to reuse this pattern for different applications.
Testable	*Sometimes.* Since the pattern can be visualized we can, at least, visually test it.
Easy to understand	*Sometimes.* Despite the accuracy of the pattern, it is easy to understand its structure. However, this is not always the case.
Graphical or visual	*Always.* The pattern has a graphical presentation, which is the class diagram.
General	*Never.* We cannot use it to model the account in other contexts other than monetary applications. Also, it does not cover the cases of having accounts without entries and vise versa. Moreover, the pattern does not cover some of the situations such as having more than one holder for the same account.
Easy to use and reuse	*Never.* Using the patterns of this group in different applications than they were originally built for, if at all possible, is not an easy task.

Table 6.1: Experience group evaluation summary

6.4 Analogy Group Evaluation

Figure 6.2 provides the class diagram of the *Resource Rental Pattern* [27], a pattern representative of this group. The objective of the pattern is to provide a model that can be reused to model the problem of renting any resource. Figure 6.3 provides an example of

the *Resource Rental Pattern* applied in the context of a library service [27]. Many examples for applying this pattern to different applications are suggested in [27].

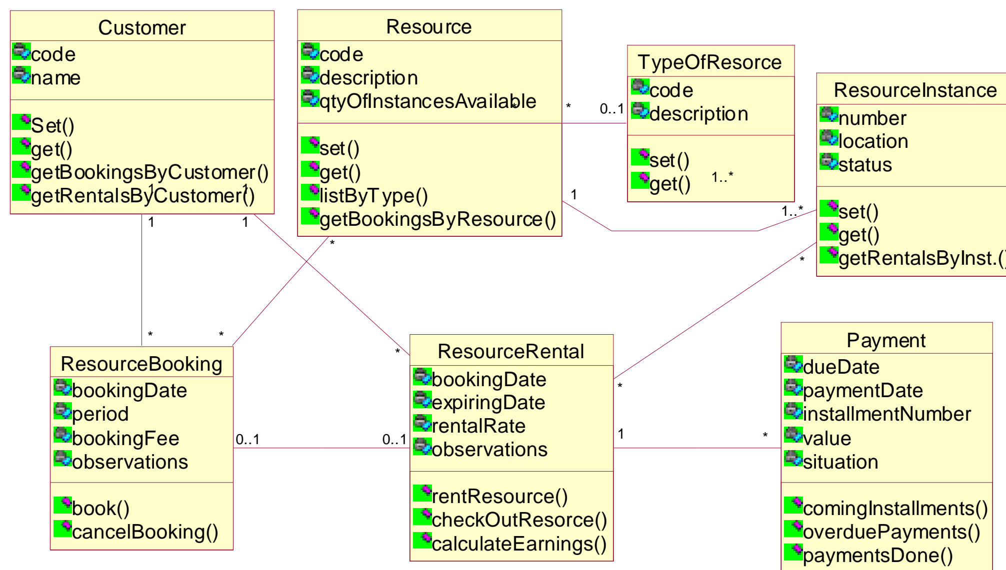


Figure 6.2: Resource Rental Pattern.

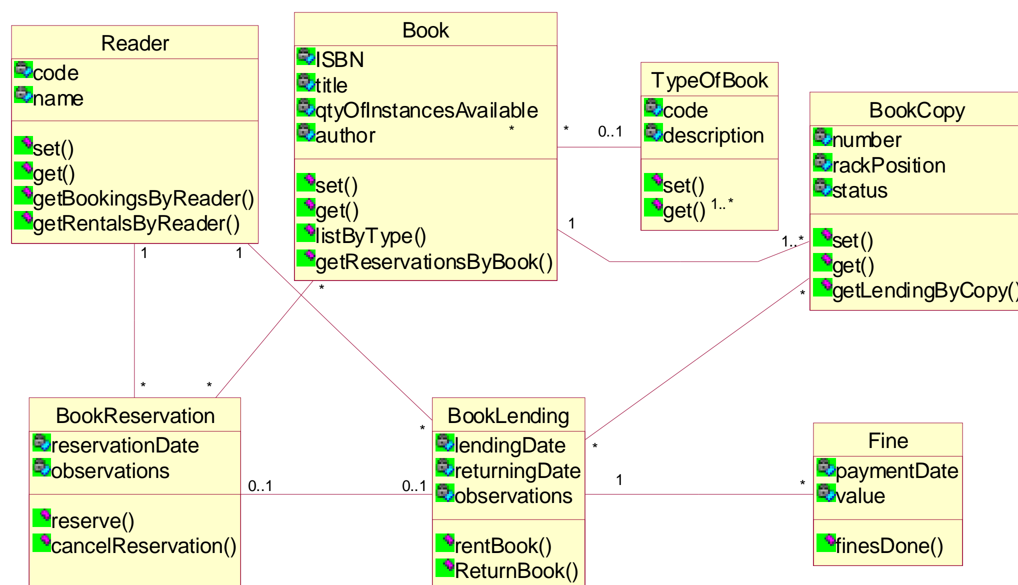


Figure 6.3: Instantiation of Resource Rental Pattern for a library service [24].

This pattern models a complete resource rental system. It models a collection of problems, whereas each of these problems could be modeled individually. For example, the “payment” process is a stand-alone problem, which could appear in many other contexts. Therefore, having a pattern that models the “payment” problem alone would be more effective, since such a pattern could be reused in many other applications.

The resource rental pattern lacks simplicity. In addition, it is not general, because it is not applicable for all resource rentals. One of the most basic steps in the automobile rental process is the question of insurance. There is nothing in this model that can be used to address insurance programs.

Another issue that is not addressed in this pattern, yet it is essential in many renting systems, is the verification process. In many cases, renting a resource might require a membership (as in the case of the universities library) or other identification (such as the driver license in the case of renting a car). There is a link between the completeness of the model and its stability. Reuse is challenging when applying incomplete patterns, because many new classes are needed to complete the model.

As an example, suppose we need to model a car rental system using this pattern. Suppose that we have reached the conclusion that we need to add the verification process classes and the insurance process classes. Substantial analysis is needed to complete this new model, hence the analyst may be inclined to build a new model from scratch. Therefore, the pattern probably will not be reused.

As in the first group, the existence of a class diagram to represent the model makes the model graphical. As a result the pattern is readily tested. Table 6.2 summarizes the evaluation of this group

Properties	Evaluation Summary
Simple	*Never.* This approach aims to model an entire system at a time. Consequently, the resultant models will model more than one problem at a time.
Complete and most likely accurate	*Never.* It is not sufficiently general to address the requirements of different renting applications.
Stable	*Never.* Using this pattern to model different applications will require major changes. For instance, adding the verification process to the model will entail a lot of changes.
Testable	*Sometimes.* Since the pattern can be visualized, we can, at least, visually test it.
Easy to understand	*Sometimes.* It does not cover all of the circumstances that might occur in an application.
Graphical or visual	*Always.* The pattern has a graphical presentation, which is the class diagram.
General	*Never.* Since this approach is based on modeling the entire system, its generality will diminish. It is usually hard to capture all of the aspects of a specific application and all of its possible variations within a single model. The shown pattern cannot be used to model the rental of some resources, such as car rental, since there is nothing in the model that covers insurance issues, which are an essential part of any car rental process.
Easy to use and reuse	*Never.* As mentioned before, major changes would be required to use this pattern in different applications.

Table 6.2: Analogy group evaluation summary

6.5 Stable Analysis Pattern Group Evaluation

This group reflects our proposed solution for building reusable analysis models and avoiding most of the common problems we found in the other groups. In order to make the evaluation of this group of patterns more interesting, the pattern example that has been chosen is the stability version of the *Account* pattern introduced by Fowler.

From the stability point of view, the model that focuses on the account problem has nothing to do with the entry problem. Therefore, it is necessary to develop separate patterns for each individual problem. In this manner, the simplicity of our models is guaranteed, since each pattern will focus on a specific problem.

Stability goes further by providing classes that do not exist in Fowler's model. Figure 6.4 shows the proposed analysis pattern "*AnyAccount*" that provides the stable model for the "account" problem. The new classes that appear in the stability model help handle those circumstances that Fowler's model fails to cover, thus, the model becomes more accurate and complete. For instance, the use of the EBT "Ownership" and the BO "Holder" in the modeling of the account aids in contexts where there is a difference between the account owner, and those who are authorized to use the account under certain circumstances.

"Ownership" is an enduring concept, which should never change, independent of context. On the other hand, "Holder" is externally stable and never changes with time, but the holders of the account could change internally (holder may get ill, for example). As we can see in the pattern class diagram, the inherited objects from the "Holder" object model the different roles inherent to the different levels of usability of the account. This

pattern's structure is stable over time and general enough to handle different applications that involve accounts and different situations within the same application, as well.

Considering "entry" as a stand-alone problem forces us to build a pattern that models an entry regardless of context. Using stability concepts, we were able to design a stable pattern the models any entry for any application. This pattern is called "*AnyEntry*", and its class diagram is given in Figure 6.5.

By combining the pattern that models the account problem (the "*AnyAccount*" pattern) with that which models the entry problem (the "*AnyEntry*" pattern) we can demonstrate the ease of reusing stability models to construct comprehensive models. Figure 6.6 shows the class diagram for this third pattern. The "*AccountWithEntry*" pattern can be used to model any account that has entries associated with it, as in the case of banking accounts and email accounts. Table 6.2 summarizes the evaluation of this group.

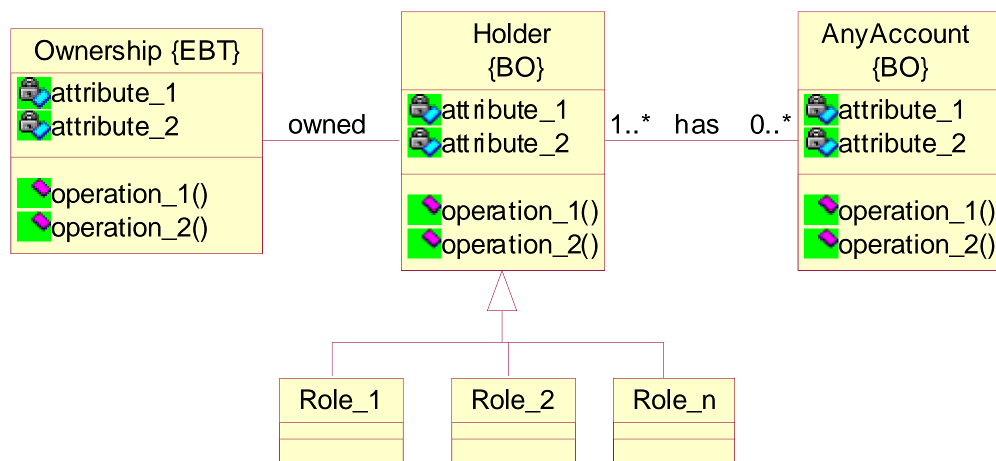


Figure 6.4: *AnyAccount* pattern class diagram

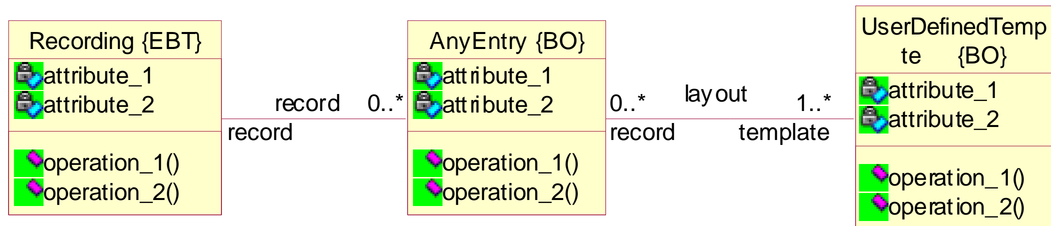


Figure 6.5: *AnyEntry* pattern class diagram

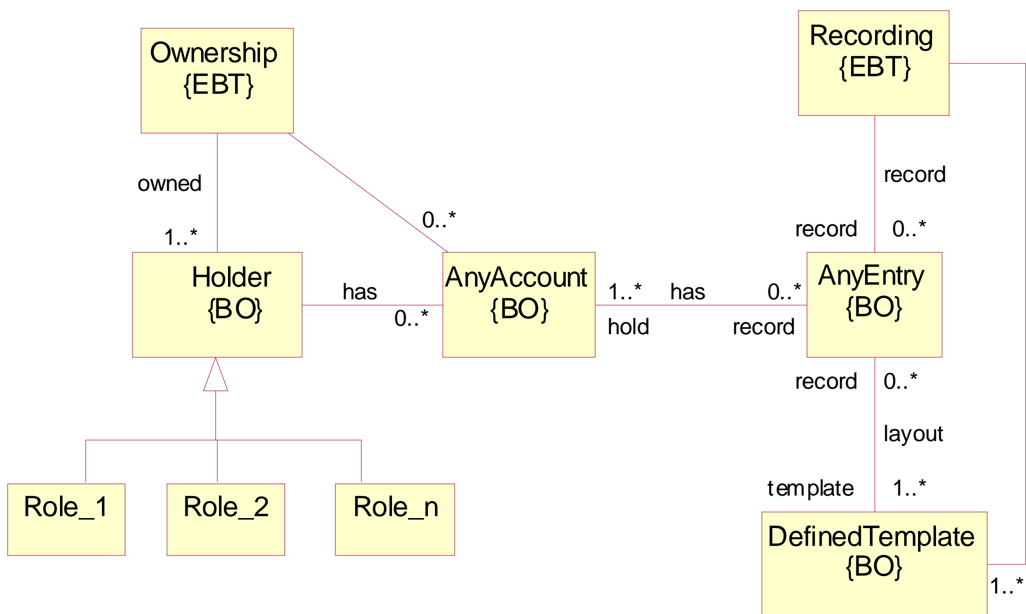


Figure 6.6 *AccountWithEntry* pattern

Properties	Evaluation Summary
Simple	*Always.* The way this approach analysis the problem forces the analysis to focus on one problem.
Complete and most likely accurate	*Sometimes.* Satisfying this property will greatly depends on how much effort will be put in the modeling. With stability, iteration is necessarily in order to reach good levels of accuracy.
Stable	*Always.* The patterns in this group are built with stability in mind. The use of EBTs and BOs ensure stability in the model.
Testable	*Sometimes.* Since the pattern can be visualized, we can, at least, visually test it.
Easy to understand	*Sometimes.* Once familiarity with the stability concepts is achieved, this evaluation will become *Always.* Accomplishing the concepts of stability is easy, yet it requires some time in the beginning.
Graphical or visual	*Always.* The pattern has a graphical presentation, which is the class diagram.
General	*Always.* Because of the stability concept, our models focus on a specific problem, trying to illuminate the core knowledge beneath the surface of the problem. Since the core knowledge of a problem is constant, regardless of the context that this problem might appear in, the model of the problem is general and can apply to the problem whenever it occurs.
Easy to use and reuse	*Always.* Using the pattern by itself or the integration of a few patterns are both easy to accomplish. This property is demonstrated by introducing the third pattern shown in figure 6.

Table 6.3: Stable analysis pattern group evaluation summary

6.6 Conclusion

The evaluation of some analysis patterns shows that these patterns lack many essential properties. As a result, their reusability is diminished. On the other hand, *Stable analysis patterns* demonstrate effectiveness by satisfying all the proposed properties.

Chapter 7

Conclusions and Future Work

- In this thesis, we have investigated the main problems that diminish the effectiveness and reusability of today's analysis patterns. The lack of stability, the insufficient level of abstraction, and the inadequate description of analysis patterns prevent software developer from effectively utilizing analysis patterns.
- Based on the SSM concepts, we have proposed the new concept of *stable analysis patterns* as a solution to the stability and abstraction level problems.
- To enhance the communication of analysis patterns among developers, we have proposed a new documentation template. The proposed template has been shown to capture all of the necessary aspects to ensure deep understanding and effective reuse of the patterns it describes.
- In order to demonstrate the use of the proposed approach, we have proposed five stable analysis patterns. Table 7.1 provides a quick reference for these five domain-less patterns.
- In order to evaluate the proposed approach, we developed eight essential properties and examined the proposed stable analysis patterns against these properties. Stable

analysis patterns demonstrated effectiveness by satisfied all eight of the essential properties.

Pattern Name	Page
AnyNegotiation	30
Party	31
AnyAccount	124
AnyEntry	125
AccountWithEntry	125

Table 7.1: Quick reference to the proposed five stable analysis patterns.

- By documenting our experience while building several stable analysis patterns, we have proposed a pattern language to help software developers construct their own stable analysis patterns. The high abstraction level of the pattern language makes it usable for any domain.
- In the future, the documentation template will be enhanced by adding the following fields:
  1. Formal specification, which gives the formal description of the pattern. A Formalized description of the patterns using the Z formal language will be written. This formalization will improve the verification and the application of analysis patterns.
  2. Design issues, which discuss the important issues required for linking the analysis phase to the design phase.

In addition, metrics for measuring pattern reusability will be developed. These metrics will be used to evaluate and to compare today's analysis patterns and the proposed stable analysis patterns.

In summary, we have presented a new concept for building analysis patterns, *Stable Analysis Patterns,* a novel template for documenting and communicating analysis patterns, and eight essential properties for evaluating the effectiveness of analysis patterns. As a result of our experience in building several stable analysis patterns, we have developed a pattern language that shows the "why" and the "how" of each step in building stable analysis patterns. In conclusion, our results show that the stable analysis pattern approach can play a major role in enhancing the use of analysis patterns in software development.

References

1. A. Geyer-Schulz and M. Hahsler, "Software Engineering with Analysis Patterns", Technical Reports 01/2001, *Institut fur Informationsverarbetung und –wirtschft, Wirschaftsuniversitat Wine, Augasse 2-6,* 1090 Wine, November 2001.

2. A. Mahdy, M.E. Fayad, H. Hamza, and P. Tugnawat, " Stable and ReusableModel-Based Architectures" *12th Workshop on Model-based Software Reuse, 16th ECOOP,* June 2002, Malaga, Spain.

3. C. Alexnader, S. Ishikawa, M. Silverstein, M. Jacobson, M. Fiksdahl-King, and S. Andel, *"A pattern Language"*, New Work: Oxford University, 1977.

4. Composite Capability/Preference Profiles (CC/PP): A user side framework for content negotiation. W3C Note 21 July 2000. http://www.w3.org/TR/NOTE-CC.

5. D.C. Schmidt, M.E. Fayad, and R. Johnson, "Software Patterns", *Communications of the ACM, Vol. 39, No. 10*, October 1996, pp. 37-39.

6. D.C. Hay, *"Data Model Patterns: Conventions of Thought"*, New York: Dorset House Publishing. 1995.

7. D.L. Parnas, " Software Aging", *Proceedings of the 16th International Conference on Software Engineering*, May 1994, pp 279-287.

8. D. Riehle and H. Züllichoven, "Understanding and Using Patterns in Software Development", *Theory and practice of object systems*, 2:1, 1996.

9. E. B. Fernandez and X. Yuan, "An analysis pattern for reservation and use of reusable entities", *Pattern Languages of Programs Conference, (PLoP99)*. http://st-www.cs.uiuc.edu/~plop/plop99.

10. E. Gamma et al., *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, 1995.

11. F. Buschmann et al., *"Pattern-Oriented Software Architecture, A System of Patterns"*, John Wiley & Sons Ltd, Chichester, 1996.

12. H. Hamza, "Building Stable Analysis Patterns Using Software Stability", *4th European GCSE Young Researchers Workshop 2002 (GCSE/NoDE YRW 2002),* October 2002, Erfurt, Germany.

13. H. Hamza, and M.E. Fayad "Model-base Software Reuse Using Stable Analysis Patterns", *ECOOP 2002, Workshop on Model-based Software Reuse,* June 2002, Malaga, Spain.

14. H. Hamza and M.E. Fayad, "A Pattern Language for Building Stable Analysis Patterns", *9th Conference on Pattern Language of Programs (PLoP 02)*, Illinois, USA, September 2002.

15. J. Coplien, D. Hoffman, D. Weiss, " Commonality and Variability in Software Engineering", *IEEE Software, Vol. 15, No. 6*, November 1998, pp 37-45.

16. M. Cline and M. Girou, " Enduring Business Themes", *Communications of the ACM, Vol. 43, No. 5*, May 2000, pp. 101-106.

17. M.E. Fayad, " Accomplishing Software Stability", *Communications of the ACM, Vol. 45, No. 1*, January 2001, pp 95-98.

18. M.E. Fayad, " How to Deal with Software Stability", *Communications of the ACM, Vol. 45, No. 4*, April 2002, pp 109-112.

19. M.E. Fayad, and A. Altman, "Introduction to Software Stability", *Communications of the ACM, Vol. 44, No. 9*, September 2001.

20. M.E. Fayad and M. Laitinen, "Transition to Object-Oriented Software Developments", New York: Wiley & Sons, August 1998.

21. M.E. Fayad and S. W., " Merging Multiple Conventional Models into One Stable Model", *Communications of the ACM, Vol. 45, No. 9*, September 2002.

22. M.E. Fayad, S. W., and M. Nabavi "Stable Model-Based Software Reuse", *ECOOP 2002, Workshop on Model-based Software Reuse*, June 2002, Malaga, Spain.

23. M. Fowler, *"Analysis Patterns: Reusable Object Models*", Addison-Wesley, 1997.

24. P. Coad, *"Object Models: Strategies, Patterns, and Applications"*, New Jersey: Yourdon Press, 1997.

25. P. Tugnawat, H. Hamza, and M.E. Fayad, "Software Stability in Reverse Buying System", *The ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01), DesignFest*, Tempa Bay, FL, October 2001.

26. Queensland Government, Negotiation in Purchasing. http://www.qgm.qld.gov.au/bpguide/neoti/neg.html

27. R.C. Martin, " Stability", *C++ Report*, Feb. 1997.

28. R.T. Vaccare Braga et al., “A Confederation of Patterns for Business resource Management”, *Proceedings of Pattern Language of Programs' 98 (PLOP'98),* September 1998.

29. Th. Greth, R. Schachtschabel, and R. Schönefeld, “Using Patterns in Design and documentation of Software”, *Proceedings of WOON '96- The white OO nights.* St. Petersburg, Russia, 20-21 June 1996.