A Pattern Language for Building Stable Analysis Patterns

Haitham Hamza and Mohamed E. Fayad
Computer Science and Engineering Dept.
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
hhamza@cse.unl.edu

Abstract

Software analysis patterns are believed to play a major role in reducing the cost and condensing the time of software product lifecycles. However, analysis patterns have not realized their full potential. One of the common problems with today's analysis patterns is the lack of stability. In many cases, analysis pattern that model specific problems fail to model the same problem when it appears in different context, forcing software developers to analyze the problem from scratch. As a result, the reusability of the pattern will diminish. This paper presents a pattern language for building stable analysis patterns. The objective of this language is to propose a way of achieving stability while constructing analysis patterns.

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.

Software stability concepts introduced in [1,2] have demonstrated great promise in the area of software reuse and lifecycle improvement. Software stability models apply the concepts of "Enduring Business Themes" (EBTs) and "Business Objects" (BOs). By applying stability model concepts to the notion of analysis patterns we propose the concept of Stable Analysis Patterns. 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. The pattern language presented in this paper shows to main steps required for building stable analysis patterns.

The remainder of this paper is organized as follows: Section 2 provides an overview for the proposed pattern language, and shows the interaction between the different patterns. The detailed description of each pattern is given is Section 3. Concluding remarks are presented in Section 4.

2. Pattern Language Overview

The proposed pattern language contains eight patterns. 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 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 1 summarizes the patterns language for quick reference. The relationship among the language patterns is shown in Figure 2.

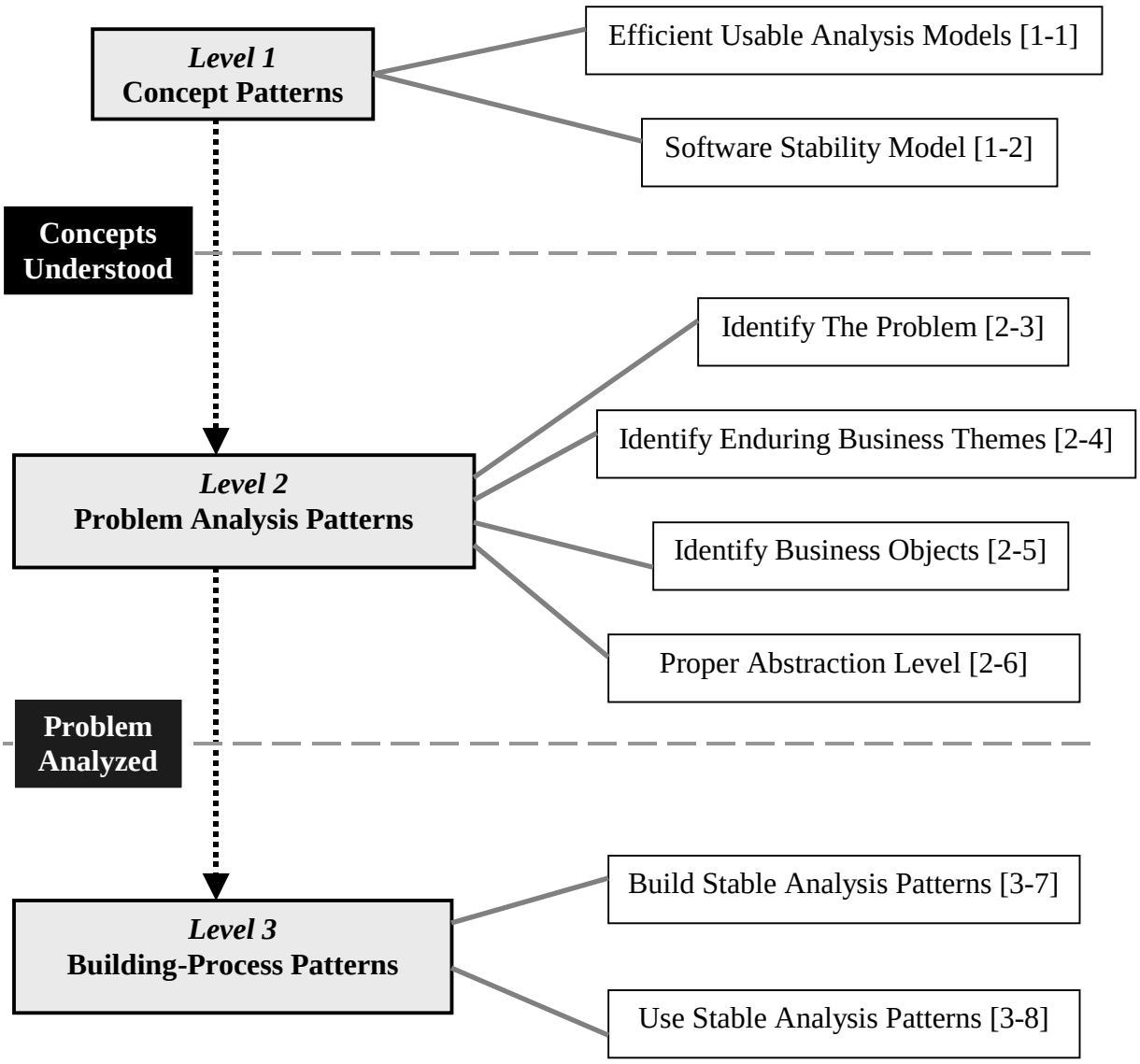


Figure 1. Description of the pattern language

Table 1. Summary of the patterns 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 5
Software Stability Model [1-2]	How to accomplish software stability?	Page 8
Identify The Problem [2-3]	How to focus on a specific problem that the analysis pattern will model?	Page 11
Identify Enduring Business Themes [2-4]	How to identify the enduring business themes of the problem?	Page 14
Identify Business Objects [2-5]	How to identify the business objects of the problem?	Page 18
Proper Abstraction Level [2-6]	How to achieve the proper abstraction level?	Page 22
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 27
Use Stable Analysis Patterns [3-8]	How to use the contracted stable analysis pattern to model the problem within a specific context?	Page 30

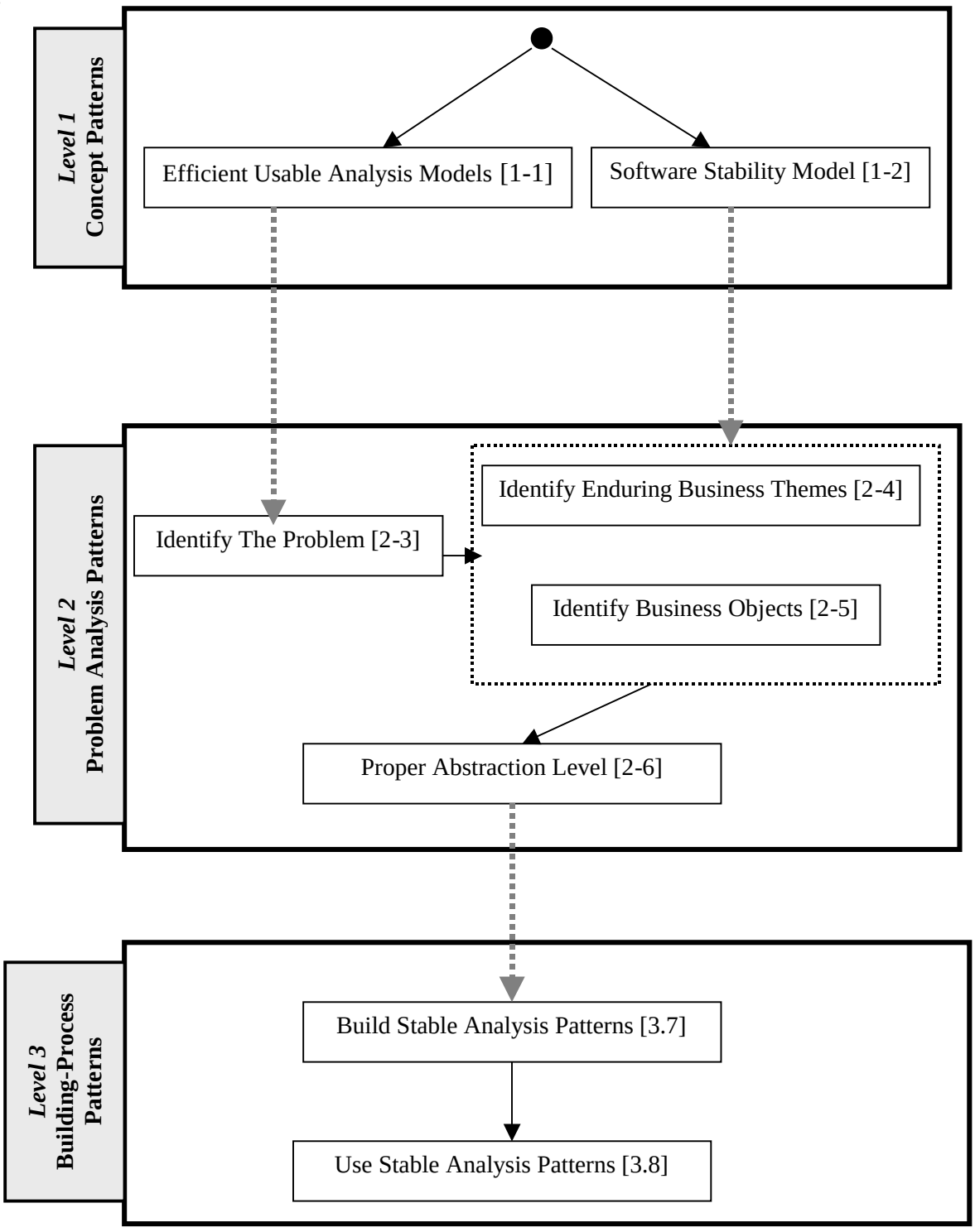


Figure 2. The pattern language chart

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].**

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 [10]. That is, to be simple, complete and most probably accurate, testable, stable, to have visual representation, and to be easily understood. 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
We introduce eight essential properties that cover the main qualities of any analysis model. 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.

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".***

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
Software stability concepts introduced in [1,2] have demonstrated great promise in the area of software reuse and lifecycle improvement. 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. For instance, classes that present the enduring and basic concepts of the system are classified as EBTs. Since EBTs represent the enduring concepts of the system, they are extremely stable, and thus they form the nucleus of the SSM.

The classes that are tangible and externally stable, but are internally adaptable, are classified as BOs. For instance, human beings are BOs. They are externally stable, however, they can change internally (humans can get married, or become ill). The EBTs and the BOs form the SSM core.

The IOs layer contains the unstable classes, and thus they might be replaced, added, or even removed from the system without affecting the core of the system.

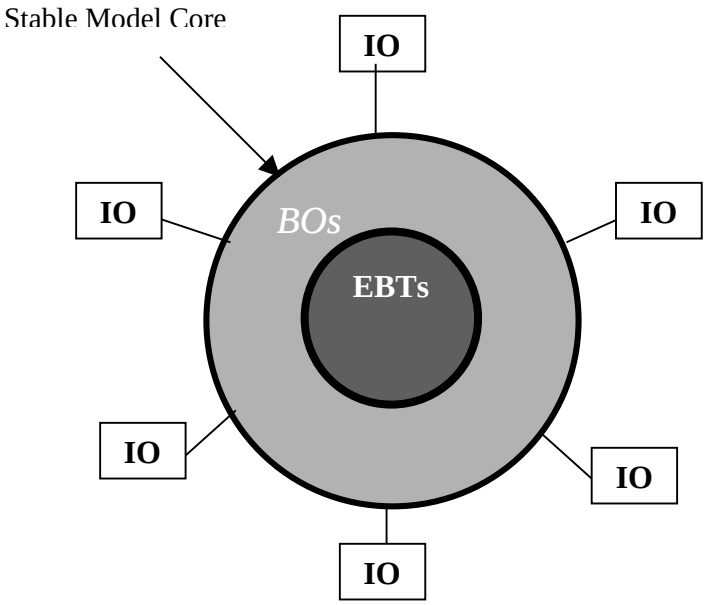


Figure 3. Software Stability Model Architecture.

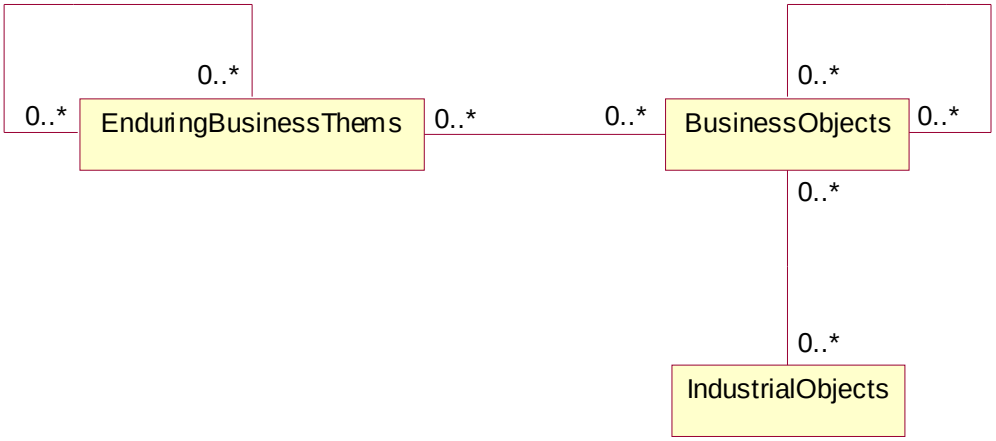


Figure 4. The relation between SSM elements

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

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"**.

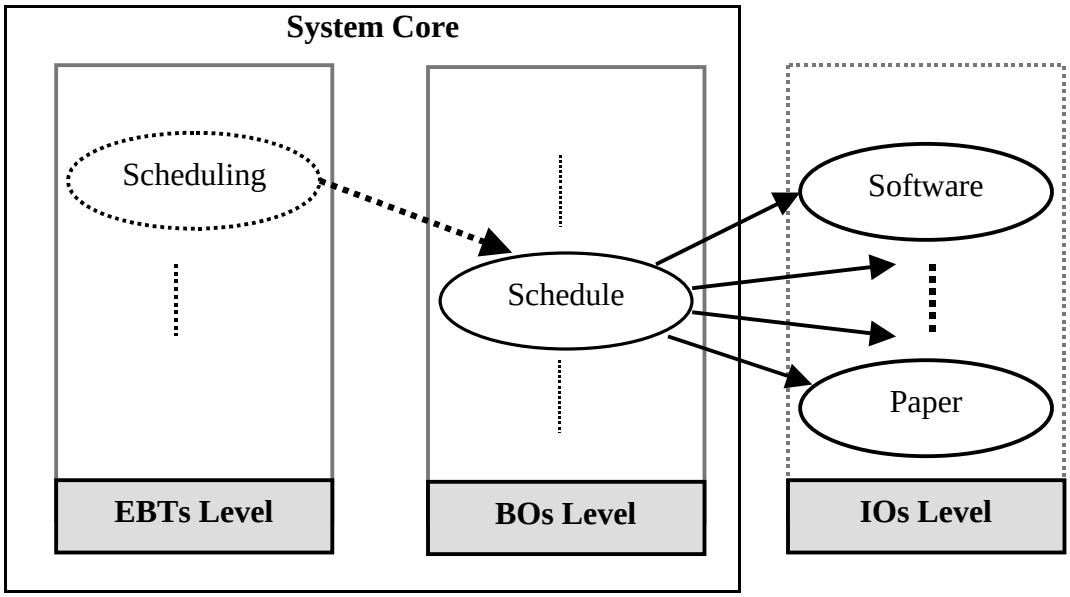


Figure 5. Example of the mapping between EBTs, BOs, and IOs.

Level 2 **Problem Analysis Patterns:** First, **Identify The Problem [3]**. Then **Identify Enduring Business Themes [4]**. After that, **Identify Business Objects [5]**. Finally, put the identified EBTs and BOs in the **Proper Abstraction Level [6].**

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?"

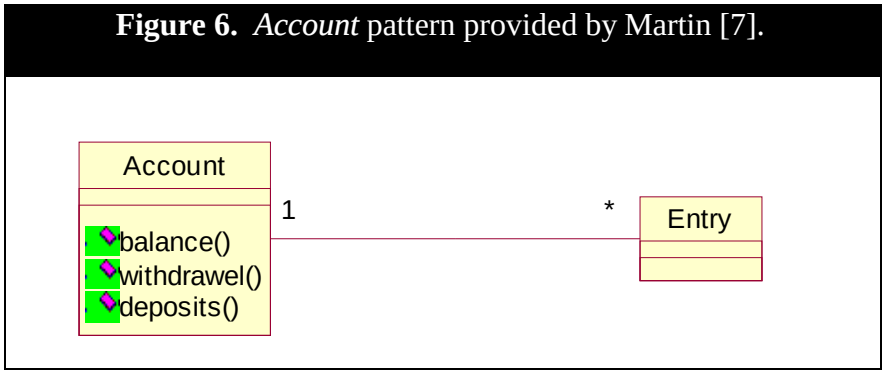
If we can find practical scenarios for each of the smaller problems we have separated, then we need to model each of them separately. If the smaller problems have no practical use, they should be modeled together.

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 [7]. Figure 6 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 7 shows some examples of accounts without entries, while Figure 8 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 the paper. 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 6. *Account* pattern provided by Martin [7].



(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 7 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 8 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.

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 9 summarizes the three steps needed to identify the EBTs of the problem.

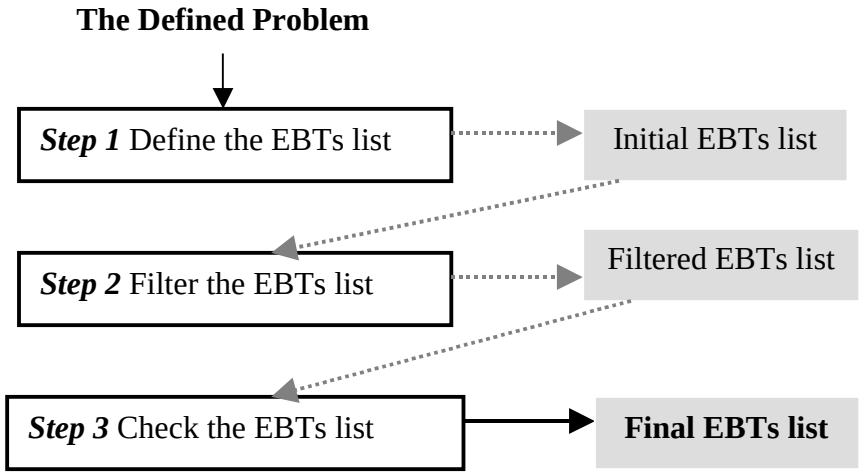


Figure 9. The steps for identifying the EBTs of the problem.

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.

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).

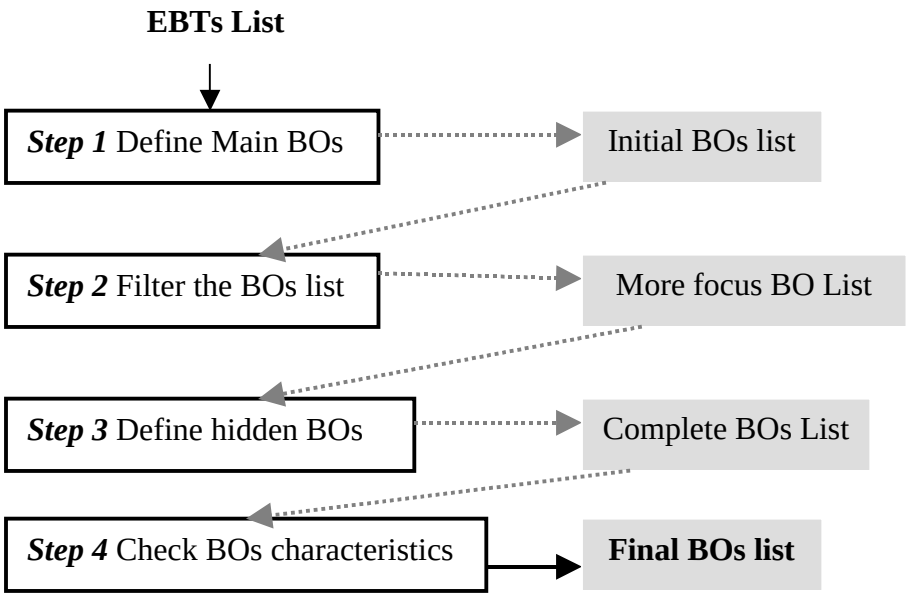


Figure 10. The steps for identifying the BOs of the problem.

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.
  BOs main list: "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”***.

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 11 shows the detailed steps taken while contemplating the problem.

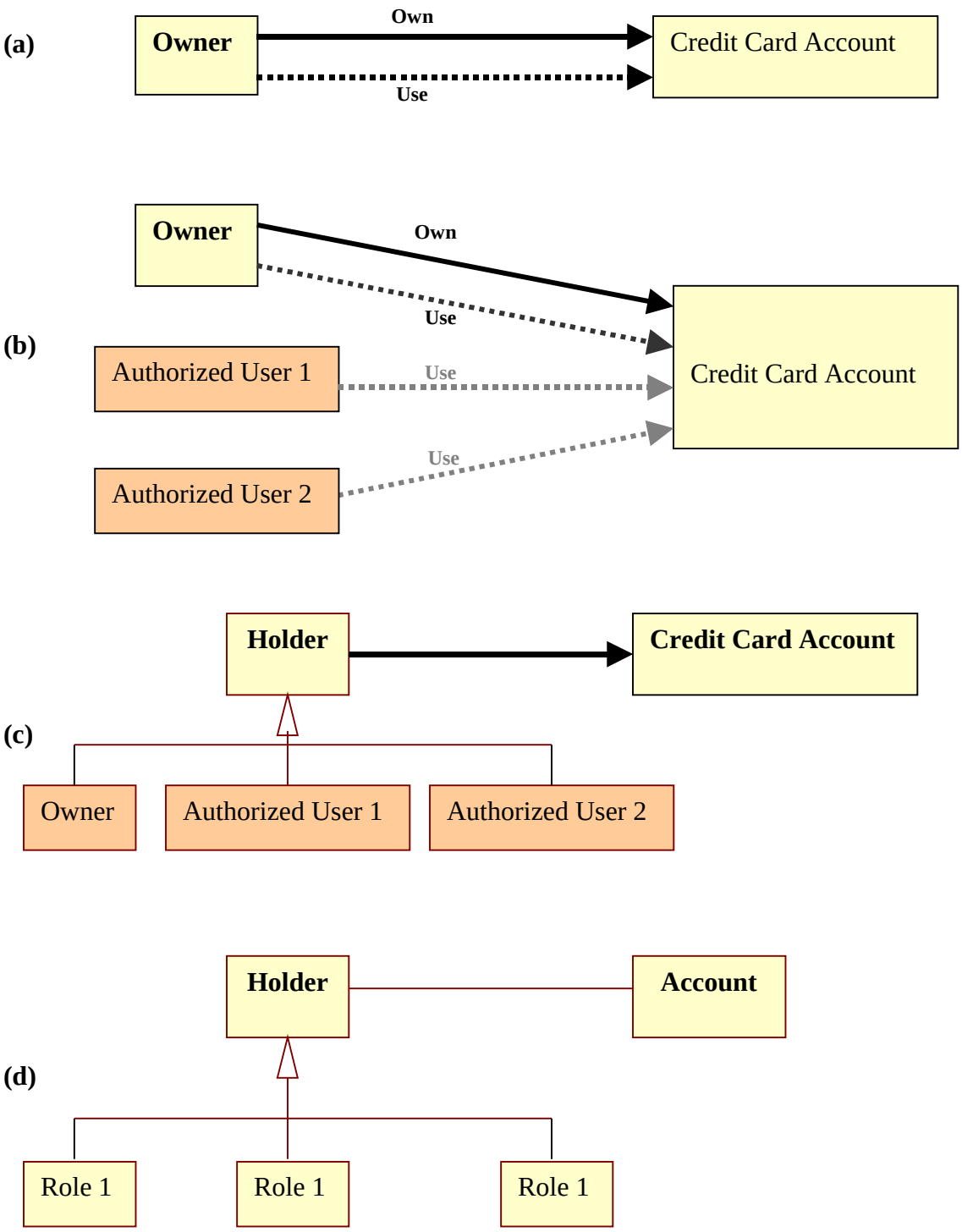


Figure 11. **(a)** Shows the classic relationship between the "Owner" and his "Account". **(b)** Shows the possible situation that cannot be handled with the current representation. **(c)** Shows possible solution to handle the problem. **(d)** Gives the final abstraction level that handles the problem.

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. Figure 12 shows an example of this group' pattern [11]. The objective of the shown pattern is to provide a model that can be reused to model the problem of renting any resource. Figure 13 shows an example of using this pattern in the context of a library service [11].

Figure 12. Resource Rental Pattern [11].

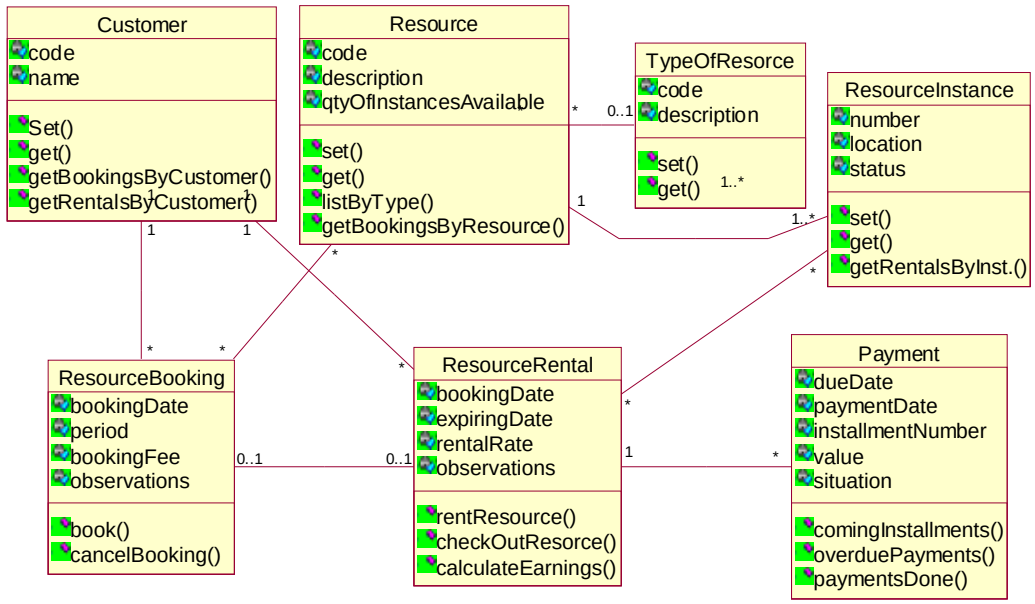
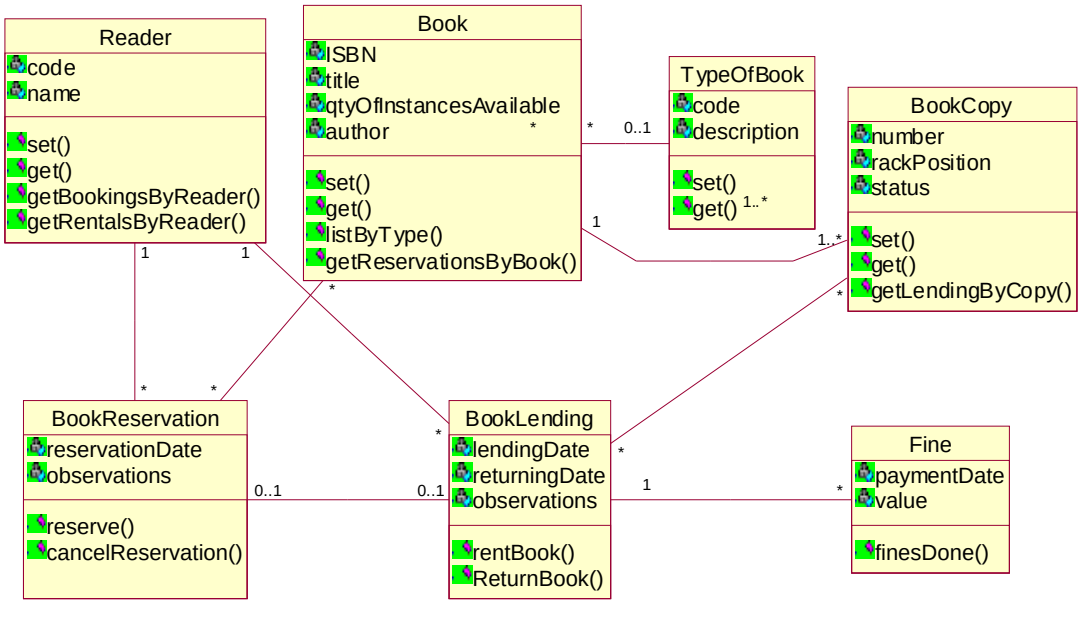


Figure 13. Instantiation of Resource Rental Pattern for a library service [11].



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 [7],** and then **Use Stable Analysis Patterns [8].**

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 14 shows the relationship between the EBTs and the BOs of the problem.

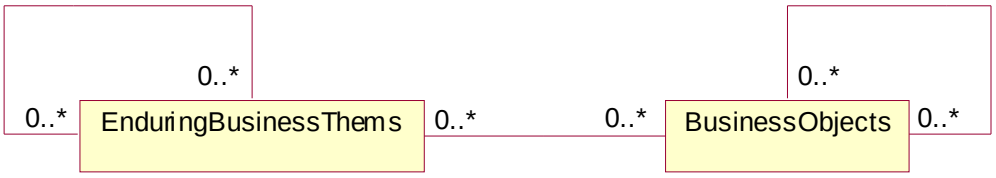


Figure 14. The relationship between the EBTs and the BOs of the problem.

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 15 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 16 gives the structure of the "AnyEntry" pattern.

Figure 15. *AnyAccount* pattern class diagram

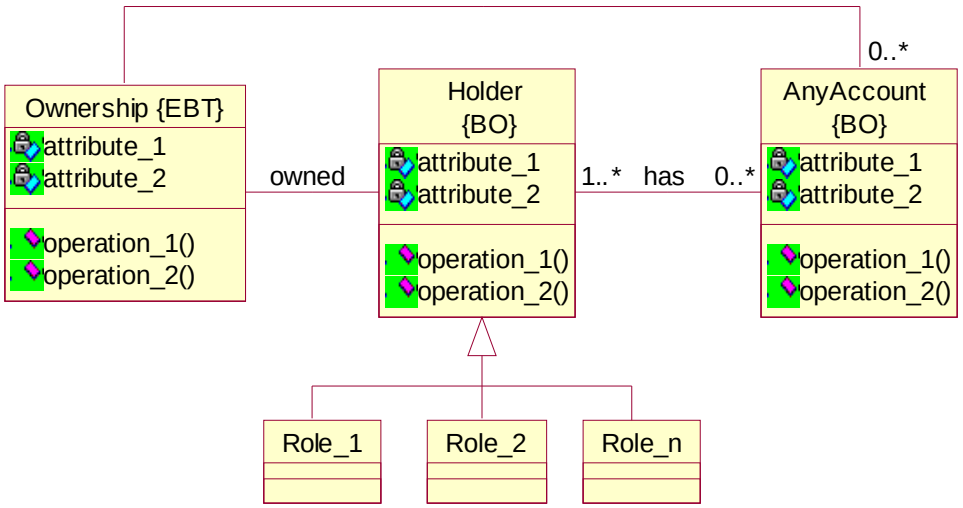
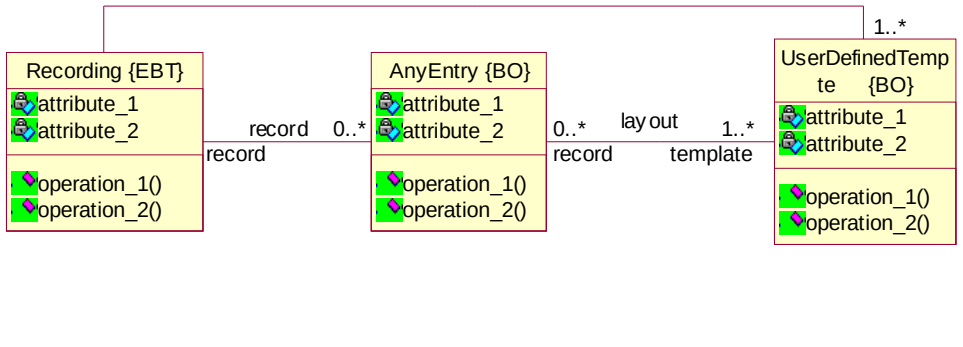


Figure 16. *AnyEntry* pattern class diagram



Next Pattern

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

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 contracted 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 17 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.

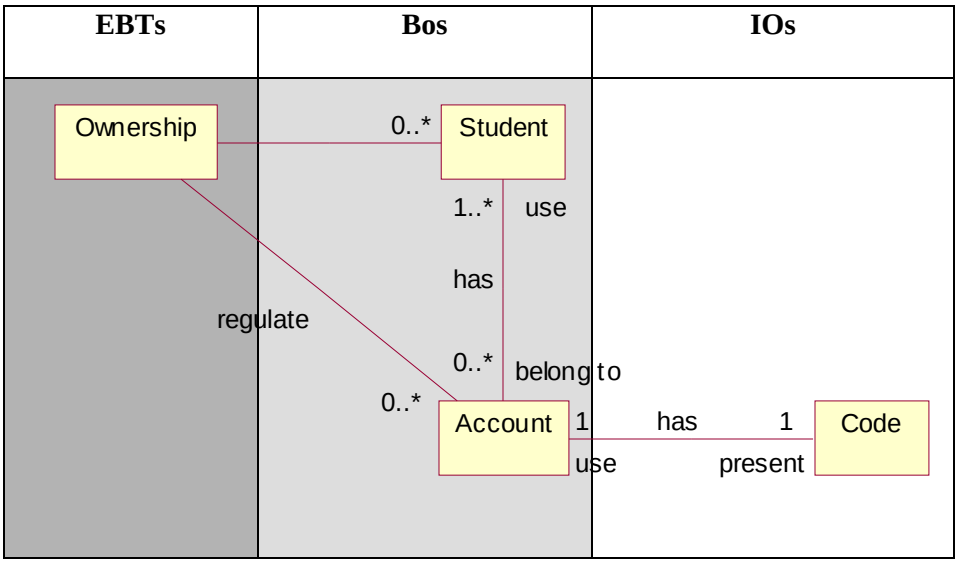


Figure 17. *Copy Machine Account* object diagram

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 18 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.

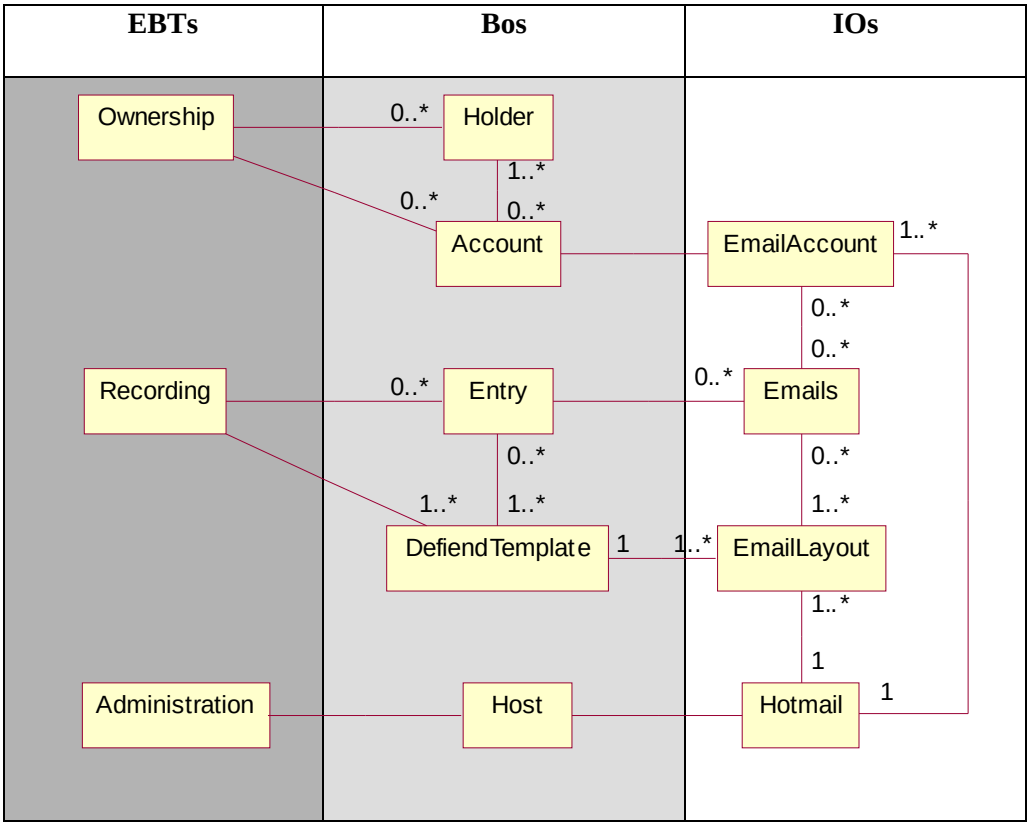


Figure 18. *Hotmail Account* object diagram

Concluding Remarks

The proposed pattern language provides an approach for achieving stability while building analysis patterns. Thus, we can increase the effectiveness, and improve the reusability of analysis patterns. Future work includes demonstrating the effectiveness of this pattern language by applying it to many problems in different contexts.

Acknowledgement

We would like to thank our Shepherd, Bruce Whitenack for his useful comments and for his great suggestions that have significantly improved this paper.

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] 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.

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

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

[5] H. Hamza, and Mohamed E. Fayad "Model-base Software Reuse Using Stable Analysis Patterns", ECOOP 2002, Workshop on Model-based Software Reuse, June 2002, Malaga, Spain.

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

[7] M. Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997.

[8] M.E. Fayad and H. Hamza. "Introduction to Stable Analysis Patterns", Communications of the ACM**,** Thinking Objectively, Vol. 45, No. 9, September 2002.

[9] M.E. Fayad and H. Hamza. "Comparative Study of Analysis Patterns", Communications of the ACM**,** Thinking Objectively, Vol. 45, No. 11, November 2002.

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

[11] R. T. Vaccare Braga et al., "A Confederation of Patterns for Business resource Management" Proceedings of Pattern Language of Programs' 98 (PLOP'98), 1998.