THE ANYACCOUNT PATTERN

Mohamed E. Fayad[1] and Haitham Hamza[2]

[1]*Computer Engineering Dept., College of Engineering, San José State University*
One Washington Square, San José, CA 95192-0180
m.fayad@sjsu.edu

[2]*Computer Science and Engineering Dept., University of Nebraska-Lincoln*
Lincoln, NE 68588, USA
hhamza@cse.unl.edu

Abstract

*The **AnyAccount** pattern models the core knowledge of any account, making it easy to reuse this pattern and build upon it to model different kinds of accounts rather than thinking of the same problem each time from scratch. This pattern can be utilized to model any kind of account in any application and it can be reused as part of a new model.*

1. Introduction

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 kinds of accounts.

In the last decade, there were many patterns that have been developed to model the account problems. However, even though they are all aimed to model the same problem "the account problem", and they are all developed based on the long experience of their developers; each pattern has its own structure, which is noticeably different from the others. What might be surprising is that most of these different models are developed for similar applications, which are usually monetary applications, and all are claimed to be working just fine in the project they where originally developed for. Examples of different patterns that model the account problem can be found in [1][2][4]. In the end of this paper we will discuss and evaluate some of these patterns.

There are some fundamental questions still to answer: Why do we have MANY different patterns that model a single problem? Can we develop a pattern that captures the atomic account notion, and thus can serve as a base for modeling any kind of accounts? The objective of this paper is to provide an answer to these questions by discussing and documenting the atomic pattern ***AnyAccount***. This pattern models the core knowledge of

an account, making it easy to reuse this pattern and build upon it to model different kinds of accounts rather than re-working the same problem each time from scratch. AnyAccount pattern is a stable pattern [5,6 ,8] that is build based on the software stability concepts [9].

2. Software Stability and Stable Analysis Patterns: Brief Background

The pattern proposed in this paper is based on the concept of Stable analysis patterns introduced in [5,6,8]. The idea behind stable analysis patterns is to analyze the problem under consideration in terms of software stability concepts [9]. Software stability stratifies the classes of the system into three layers: the Enduring Business Themes (EBTs) layer (contains classes that present the enduring and basic knowledge of the underlying industry or business, and hence, they are extremely stable), the Business Objects (BOs) layer (contains classes that map the EBTs of the system into more concrete objects. BOs are tangible and externally stable, but they are internally adaptable), and the Industrial Objects (IOs) layer (contains classes that map the BOs of the system into physical objects.).

Figure 1 depicts the three layers of the software stability model. The detailed characteristics of EBTs, BOs, and IOs and useful heuristics and examples of identifying these concepts in real applications can be found in [7,8,10]

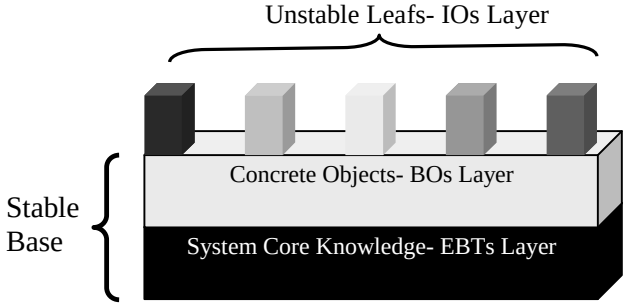

Figure1. Software stability concepts layout

3. AnyAccount Pattern Documentation

Problem

There are main four aspects that formulate the account problem:
1. Wide Recurrence: AnyAccount is required in many systems that belong to many different domains (Banking, web applications, etc).
2. Limited Scope: Existing account patterns are limited to monetary accounts. Consequently, it is fairly hard to adapt these patterns to handle accounts in other domains.

3. Generality: To over come the problem 2 above, account pattern should be general enough to form a base for developing any account in any application.
4. Extensibility: a pattern that represents a base for modeling any account should have an appropriate level of flexibility so that the developer can extend the pattern to the desired account.

Given the above three issues, the problem is how to build an account model that can capture the core knowledge of the account problem and can be reused to model the account problem in different application?

Forces

- The account problem spans a fairly wide range of applications and domains, which makes the task of capturing the core concept of the account problem more challenging than it might appear.
- Even after extracting the common features of different accounts types, the difficulty still resides in how these common features can be abstracted in such a way that makes them still valid for all these wide applications.
- Different accounts have some features that do not apply to other accounts types. The challenge arises when such uncommon features are associated with classes that should exist in the atomic pattern (For instance: in credit card accounts, it is acceptable to have many authorized holders who share the same credit card account. While, a student account in a university, for example, solely belongs to him and cannot be shared. On the other hand, the account holder is an essential part in any account independent of the account application, whenever there is an account there should be a holder/holders for this account. In this case, how can we manage to model the holder in such a way that is appropriate for such situations?"

Pattern structure and Participants

Figure 2 shows the object diagram of the *AnyAccount* pattern.

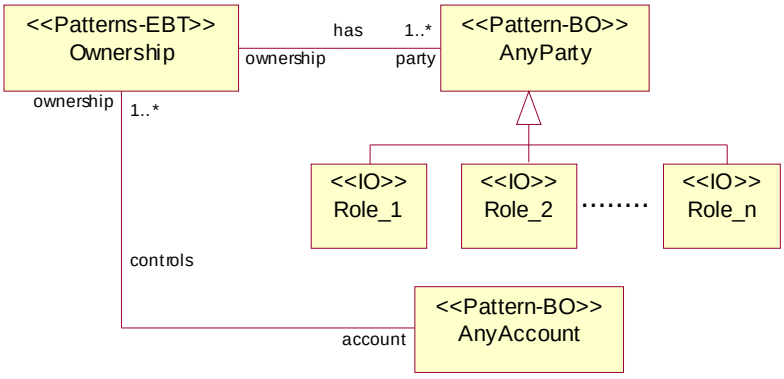

Figure 2. *AnyAccount* pattern object diagram

Participants

The participants of the ***AnyAccount*** pattern are:

1. Classes

- *Ownership.* Represents the existence of the account. An account does not exist without an Owner. Therefore, ownership is always present whenever an account exists. It describes the ownership rules and regulations to the account holder(s).
- *AnyAccount.* Represents the account itself.

2. Patterns

AnyParty. Represents the account handler(s). A party can be a person, organization, a group with specific orientation, or a mix of orientations

CRC- Cards

Ownership (Owning Controller)		
Responsibility	**Collaboration**	
	Clients	**Server**
Describes the ownership rules, and regulations to the account holder(s).	***AnyParty***	defineRules() specify() contrast() evaluate()

AnyParty (Account holder)		
Responsibility	**Collaboration**	
Access the account.	**Clients**	**Server**
	Ownership ***AnyAccount***	access() approve() activate() grant()

AnyAccount (Descriptor)		
Responsibility	**Collaboration**	
Describes account terms and conditions.	**Clients**	**Server**
	- ***AnyParty***	defineAccount() regulate(). open() close()

Applicability with illustrated Examples

The AnyAccount pattern has been developed in such away that its model captures the very basic structure of any account independent of specific applications in mind. Consequently, this atomic pattern is expected to play a role in different applications where any sort of account is required.

We choose to model a different range of applications to demonstrate the reusability of the pattern. Thus, starting from very simple application where the ***AnyAccount*** pattern can be used solely to model a problem and up to more complex examples where other patterns or objects are needed to model the problem.

Another feature that is worth illustrating in the applicability section is how this atomic pattern can indeed form the core where other specific patterns can be built upon it. For instance, by modeling the checking account as a standalone problem using the ***AnyAccount*** pattern we can build and document a new pattern called the ***CheckingAccount*** pattern, which is a domain-specific pattern, in contrast to the ***AnyAccount*** pattern, which is a domain-neutral pattern. Another thing worth realizing is that the generated domain-specific pattern can be used to model ANY checking account, making this pattern a general pattern within a specific application. This will make this pattern valuable and worth documenting and representing.

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 3 shows 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 inheritance is used, since each account should have 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 other physical representations for the BO "Account" all that would be needed 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.

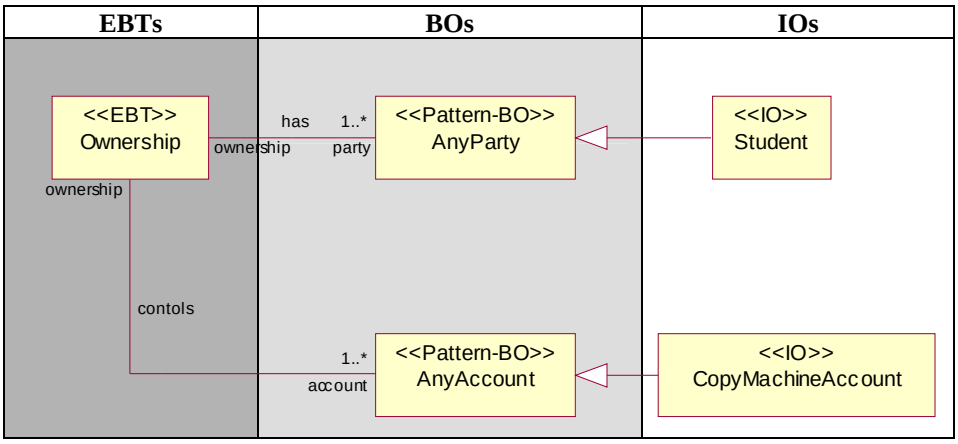

Figure 3. *Copy Machine Account* object diagram

Example 2: Modeling Hotmail Account

This example shows how to integrate more than one pattern to model a larger problem. The aim of the problem is to utilize the two constructed patterns: the *"AnyAccount"*, and the "*AnyEntry*" (shown in Appendix) patterns in the modeling of a simple email Account. For simplicity, only the object diagram of the problem model is shown in Figure 4.

It is worth to notice how two stable patterns are connected together in one model. As shown in Figure 4, the connectivity between the two used patterns is realized in the EBT and BO layers but not the IOs layer.

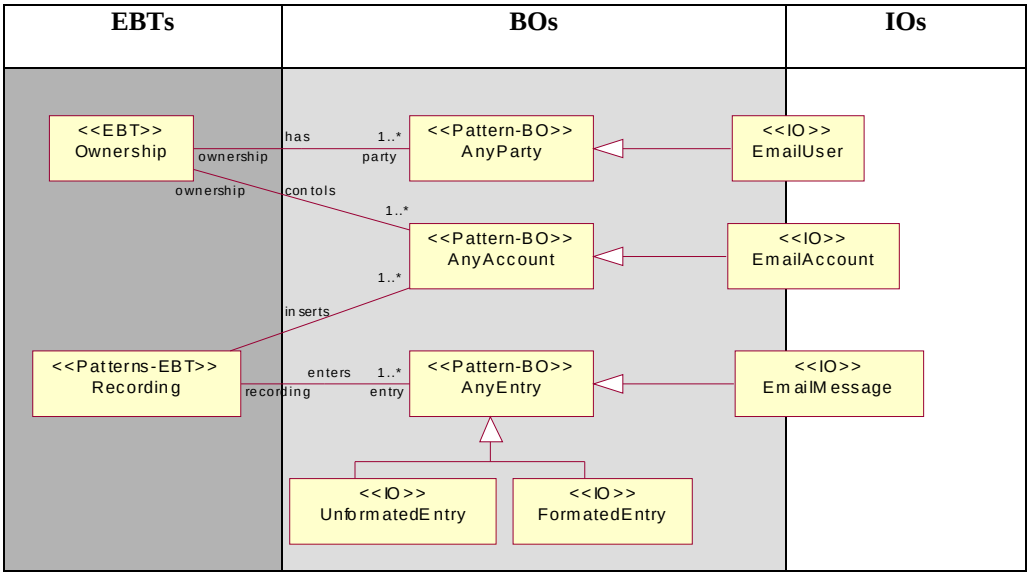

Figure 4. *Hotmail Account* object diagram

Related Patterns

Several patterns that address the "Account" problem have been proposed; yet they are all fairly different [1],[2]. Figure 5 shows the class diagram of the *Account pattern* provided by Fowler [1] 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.. Apparently, the pattern was developed to model monetary accounts, and hence, it is not obvious how far it could be successful if this pattern is to used to develop other account kinds.

Figure 5. *Account* pattern provided by Fowler [22].

Fowler's *Account* pattern 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 discussed earlier in the paper. As a result, the generality of the pattern is limited.

Moreover, even though the goal of this pattern is to deal with monetary, the pattern does not capture all the essential aspects that might appear frequently in banking accounts, for instance. To clarify this point we consider a simple example. 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 has specific privileges for using the account. Such situation cannot be handled by using this account model.

.

In Figure 6 depicts another pattern that has been developed to deal with the account problem. While the structure of this pattern is quite different than the one shown in figure 5, both present monetary accounts, and hence can not be applied to model any account kinds.

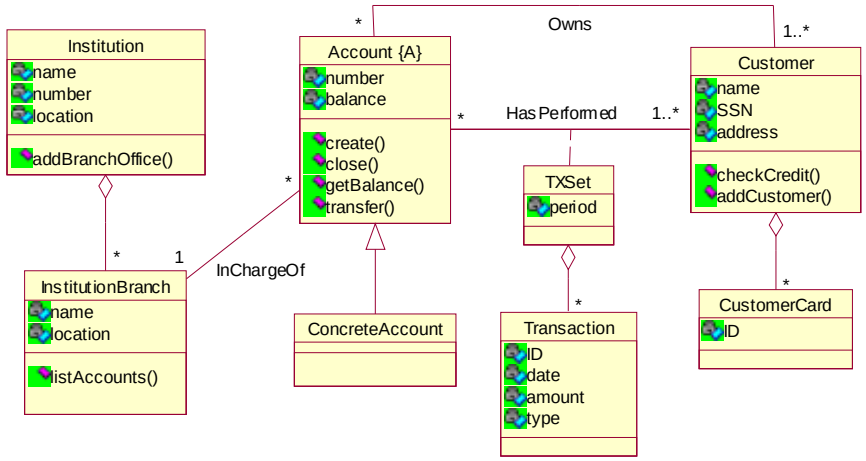

Figure 6. Another pattern for the Account Problem [2]

Discussions and Conclusions

It is worth to point out that the AnyAccount pattern is concerned with the enduring themes of “Ownership”, "Identity" and "Security". However, the only enduring concept appears in the pattern structure (figure 2) is “Ownership”. The reason for that is due to the fact that it necessary to limit the size of the developed pattern, otherwise, the resultant pattern might become too complex. This complexity will reflect on the pattern understanding and hence its usability. A crucial question in developing stable patterns is: which EBT should we include? In fact, in this paper we do not show the details of developing the AnyAccount pattern. However, the details of such steps can be found in [7]. In developing any stable pattern, one basic step is to identify the EBTs in the problem that the pattern addresses. The output of this step is usually a "list" of few EBTs (A reasonable list would contain three to four EBTs). This list is then refined to choose the final EBT. If this step has not been conducted, it would become fairly hard to manage the size and the complexity of developed pattern.

In the AnyAccount pattern, Security is an important EBT and it is apparently an integral part of account; however, Security by itself is a stand alone concept that appears in

unlimited number of applications, and hence it forms a pattern by itself. This pattern can then be applied as a base in any application that involves the security notion.

In this paper we presented the AnyAccount design pattern. AnyAccount pattern encapsulates the core knowledge that qualifies the pattern to serve as a base for developing any kind of accounts in any application domains. In addition, according to Software Stability concepts [9], AnyAccount is considered to be a business object (BO) that is stable and adaptable without touching its internal structure.

Acknowledgement

We thank David Hamu, our shepherd, for his faithful help and valuable discussions that have improved this paper.

References

[1] M. Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997.
[2] Fernandez, E., and Liu, Y., "The Account Analysis Pattern", Proceeding of the 7th European Conference on Pattern Languages of Programs, Euro*PloPP02*
[3] D. Hay, "Data model patterns", Conventions of thought, Dorset House Publ., 1996
[4] IBM Corp., Patterns for e-business, http://www-106.ibm.com/developerworks/patterns/
[5] H. Hamza "A Foundation For Building Stable Analysis Patterns." Master thesis. University of Nebraska-Lincoln, 2002
[6] H. Hamza. "Building Stable Analysis Patterns Using Software Stability". 4th European GCSE Young Researchers Workshop 2002 (GCSE/NoDE YRW 2002), October 2002, Erfurt, Germany.
[7] 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.
[8] H. Hamza and M.E. Fayad. "Model-based Software Reuse Using Stable Analysis Patterns" ECOOP 2002, Workshop on Model-based Software Reuse, June 2002, Malaga, Spain.
[9] M.E. Fayad, and A. Altman. "Introduction to Software Stability." Communications of the ACM, Vol. 44, No. 9, September 2001.
[10] M.E Fayad. "Accomplishing Software Stability." Communications of the ACM, Vol. 45, No. 1, January 2002,

Appendix A: *AnyEntry* Pattern

The basic objective of an entry in any application is to keep records for something; therefore, ***Recording*** is an enduring business theme that will never change. Whenever we have an "entry" in any application, the object ***Recording*** will be there. The ***AnyEntry*** can be either formatted following defined structure or unformatted (i.e. free-formatted).

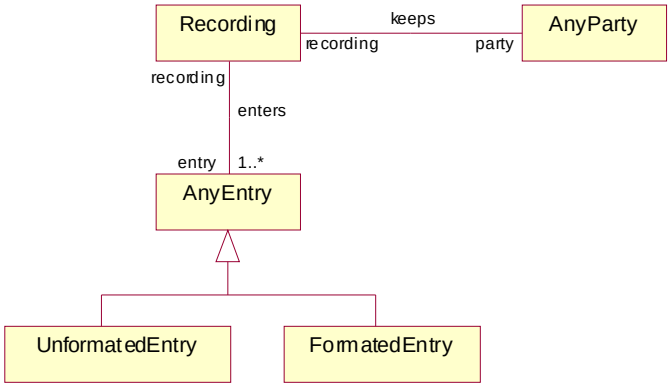

Figure A. *AnyEntry* pattern object model