Software System Engineering

Dr. M.E. Fayad, Professor
Computer Engineering Department, Room #283I
College of Engineering
San José State University
One Washington Square
San José, CA 95192-0180
http://www.engr.sjsu.edu/~fayad
Lesson 7-2:
Stable Design Patterns
Lesson Objectives

- Explore Stable Design Patterns
- Examine Stable Design Patterns
- Discuss the following Stable Design Patterns:
  - Account
  - Entry
  - Party
  - Trust
The concept of extracting patterns using software stability concepts

- Problem

**FIRST ABSTRACTION:**
- Apply SSM concepts
- Define EBTs, BOs, and IOs
- Create SSM for the problem

**SECOND ABSTRACTION:**
- Model each EBT using SSM concepts
- Build natural domain Stable Design Pattern
Stable Design Patterns’ Properties

- Timeless Notion Pattern
- Working Horse of the system Patterns
- Adaptability Patterns
- True Presentation of The Solution Space Patterns
- Management Workflow Metaphor Patterns
- Domain-Independent Patterns
Stable Design Patterns Advantages

- Contribute to Stable Architectures
- Can be Used to Model a Better Solution Space
- Provide True Solution Modeling
- Specifies Hooks for Extensions
- Enhance Team Dynamic
Stable Design Patterns’ Challenges

- Abstractions
- Adaptation
- Extensions
- Customization
- Integration
AnyAccount Pattern (1)

- 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; however, each pattern has its own structure, with noticeable differences from the others.
AnyAccount Pattern (3)

- What might be surprising is that most of these different models are developed for the similar applications, which are usually monetary applications, and all are claimed to be working just fine in the project they were originally developed for.

- Examples of different patterns that model the account problem can be found in the following Ref:


There are some fundamental questions still to answer:

- Why do we have MANY different patterns that model single problem?
- Can we develop a pattern that capture the atomic account notion, and thus can serve as a base for modeling any kind of accounts?

The objective of this lecture is to provide an answer to these questions by discussing and documenting the atomic pattern *AnyAccount*.

This pattern models the core knowledge of any account, making it easy to reuse this pattern and build on the top of it to model any kind of accounts rather than thinking of the same problem each time from scratch.
AnyAccount Pattern (5)

Pattern Name: *AnyAccount*
- This pattern is required to model the core knowledge of any account without tied the pattern to a specific application or domain; hence the name *AnyAccount* is chosen.

Problem
- 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 any application?
Forces (1)

- 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 appears.

- Even after extracting the common feature of different accounts types, the difficulty still resides in how these common features can be abstracted in such away that makes them still valid for all these wide applications.
Forces (2)

- Different accounts have some features that are not applies 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, student account in a university, for example, is 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

<<Patterns-EBT>>
Ownership

has

1..*

ownership

party

<<Pattern-BO>>
AnyParty

Role_1
Role_2
Role_n

<<IO>>

controls

account

AnyAccount
Participants

Classes
- **Ownership.** Represents the existence of the account. There is no existing for the account without the existence of an Owner. Therefore, ownership is always present, whenever there is an account exists. Describes the ownership rules, and regulations to the account holder(s).
- **AnyAccount.** Represents the account itself.

2. Patterns
- **AnyParty.** Represents the account handler(s). Party can be a person, organization, a group with specific orientation
CRC Cards (1)

<table>
<thead>
<tr>
<th>Ownership (Owing Controller)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibility</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Describes the ownership rules, and regulations to the account holder(s).</td>
</tr>
</tbody>
</table>
## CRC Cards (2)

### AnyParty (Account holder)

<table>
<thead>
<tr>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access the account.</td>
<td></td>
</tr>
<tr>
<td>Ownership <em>AnyAccount</em></td>
<td>access()</td>
</tr>
<tr>
<td></td>
<td>approve()</td>
</tr>
<tr>
<td></td>
<td>activate()</td>
</tr>
<tr>
<td></td>
<td>grant()</td>
</tr>
</tbody>
</table>

Clients

Server
## CRC Cards (3)

<table>
<thead>
<tr>
<th>AnyAccount (Descriptor)</th>
<th>Responsibility</th>
<th>Collaboration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Describe account terms and conditions.</td>
<td>Clients</td>
<td>Server</td>
</tr>
<tr>
<td>- AnyParty</td>
<td>defineAccountR egulations()</td>
<td>open() close()</td>
</tr>
</tbody>
</table>
Applicability with Illustrated Examples (1)

- AnyAccount pattern has developed in such a way that makes 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 any application where any sort of account is required.
Another feature that worth to illustrate in the applicability section, is how this atomic pattern can indeed form the core where other specific patterns can be built on its top.

For instance, by modeling the checking account as a standalone problem using the AnyAccount pattern we can build and document a new pattern called `CheckingAccount` pattern, which is domain-specific pattern, in contrast to the `AnyAccount` pattern, which is domain-neutral pattern.
Applicability with Illustrated Examples (3)

- **Simple Problem (1) 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.
Applicability with Illustrated Examples (4)

<table>
<thead>
<tr>
<th>EBTs</th>
<th>BOs</th>
<th>IOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ownership</td>
<td>AnyParty</td>
<td>Student</td>
</tr>
<tr>
<td>AnyAccount</td>
<td>CopyMachineAccount</td>
<td></td>
</tr>
</tbody>
</table>

EBTs (Ownership) has 1:* ownership of BOs (AnyParty, AnyAccount). IOs (Student, CopyMachineAccount) are related to BOs (AnyParty, AnyAccount).
Applicability with Illustrated Examples (5)

- 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.

- Other Comments on the solution: Where IOs, Application classes.
Problem (2): Modeling a simple 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.

- It is important to note that this model is not complete; it is merely for demonstration purpose.
Applicability with Illustrated Examples (7)

<table>
<thead>
<tr>
<th>EBTs</th>
<th>BOs</th>
<th>IOs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ownership</td>
<td>&lt;&lt;Pattern-BO&gt;&gt; AnyParty</td>
<td>&lt;&lt;IO&gt;&gt; EmailUser</td>
</tr>
<tr>
<td>Recording</td>
<td>&lt;&lt;Pattern-BO&gt;&gt; AnyAccount</td>
<td>&lt;&lt;IO&gt;&gt; EmailAccount</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;Pattern-BO&gt;&gt; AnyEntry</td>
<td>&lt;&lt;IO&gt;&gt; EmailMessage</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;IO&gt;&gt; UnformattedEntry</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;&lt;IO&gt;&gt; FormattedEntry</td>
</tr>
</tbody>
</table>
Discussion Questions

1. T/F
   a. Account Pattern is an analysis pattern.
   b. Account pattern contains the core knowledge of any account without tied the pattern to a specific application or domain.

2. Use AnyAccount Pattern as a foundation to model the following concepts: