|  |
| --- |
| School of Electrical and Information Engineering  University of the Witwatersrand  [Wits EIE Logo](http://www.eie.wits.ac.za/) |
| ELEN7045 - SD Methodologies, Analysis and Design |
| The Account Presentation System |
| Submitted by:  **Group 3**  Silas Mahlangu, 699358  Livious Ndebele, 701300  Sibusiso Zwane, 517473  Boitumelo Mahlong,  Bakwanyana Thobela, 855470  Ronald Menya, |
|  |
| 6/26/2014 |

|  |
| --- |
|  |

Abstract

Contents

[1. Introduction 5](#_Toc391824132)

[2. Background 5](#_Toc391824133)

[3. Development Methodology 5](#_Toc391824134)

[3.1. Team Dynamics 6](#_Toc391824135)

[4. Analysis 6](#_Toc391824136)

[4.1. Problem Context Understanding 6](#_Toc391824137)

[4.2. Specification by Example (SBE) 7](#_Toc391824138)

[4.2.1. Reasons for choosing SBE 7](#_Toc391824139)

[4.2.2. How SBE is applied 7](#_Toc391824140)

[5. Design 8](#_Toc391824141)

[5.1. Test Driven Development (TDD) 8](#_Toc391824142)

[5.1.1. Why TDD 8](#_Toc391824143)

[5.1.2. How TDD is applied 8](#_Toc391824144)

[5.2. Domain Driven Design 9](#_Toc391824145)

[5.1.1 Ubiquitous Language 10](#_Toc391824146)

[5.1.2 Domain Model 11](#_Toc391824147)

[5.1.2.1 Entities and Value Objects 11](#_Toc391824148)

[5.1.2.2 Repositories 13](#_Toc391824149)

[5.1.2.3 Factories 13](#_Toc391824150)

[5.1.2.4 Domain Services 13](#_Toc391824151)

[5.1.3 Layered Architecture 13](#_Toc391824152)

[6. Implementation 14](#_Toc391824153)

[6.1. Tools 14](#_Toc391824154)

[6.2. Design Patterns 14](#_Toc391824155)

[6.3. SOLID Principle 15](#_Toc391824156)

[6.3.1. Single Responsibility Principle (SRP) 15](#_Toc391824157)

[6.3.2. Open Closed Principle (OCP) 15](#_Toc391824158)

[6.3.3. Liskov Substitution Principle (LSP) 15](#_Toc391824159)

[6.3.4. Interface Segregation Principle (ISP) 15](#_Toc391824160)

[6.3.5. Dependency Inversion Principle (DIP) 15](#_Toc391824161)

[6.4. Dependency Injection 16](#_Toc391824162)

[6.5. Data Integrity and Error Handling 16](#_Toc391824163)

[7. Discussion 16](#_Toc391824164)

[8. Conclusion 17](#_Toc391824165)

[References 18](#_Toc391824166)

[A. APS Specification by Example scenarios and key examples 19](#_Toc391824167)

[B. APS unit tests 19](#_Toc391824168)

[C. APS DDD Layered Architecture 19](#_Toc391824169)

[D. Spring Framework dependency injection configuration 19](#_Toc391824170)

# Introduction

This report discusses the solution to the Account Presentation System (APS) problem [2]. The report covers the following sections: Section 2 provides the Background of the APS problem; section 3 covers the Development Methodology used. Section 4 explains the Problem Analysis techniques and section 5 discuss the Design of the solution. Implementation is covered in section 6 and section 7 covers the Discussion of the outcome from the analysis, design and the implementation. The report’s Conclusion is covered in section 8.

The discussion is backed by the material that has been covered in the ELEN 7045 course [1] e.g. requirement communication techniques, Test-Driven Development (TDD), Specification By Example, Analysis and Design Patterns, Domain-Driven Design (DDD) and UML.

# Background

Computer systems have become a vital channel for delivering paperless solutions to the consumer. The Web in particular has enabled consumers to view information anytime and anywhere.

Account statements viewing is of no exception. Gone are the days when a customer only relied on postal services like the Post Office and other courier companies to deliver statements. Billing companies have realized that the need to deliver statements online gives them a competitive edge.

Moreover, amalgamation of statements from various **Billing** companies a customer subscribes to is an added advantage in that it provides a centralized point at which a customer can view their statements at once. The goal of the APS problem is to provide a single point of access for all of a customer’s account statements. The APS core intent is to periodically scrape customers’ account statements from their respective **Electronic-Billing** websites and ensure that the customers can view these statements at single focal point.

# Development Methodology

The development methodology adopted in the analysis, design and implementation of the APS solution is Agile [3]. The reasons for adopting Agile are specified below

* The short time frame given to deliver this system
* Requirements vagueness justifies an iterative and incremental development.
* The requirement could be broken down into small deliverable subsystems.
* No need to wait for detailed specification

No specific type of Agile methodology has been chosen because most require a certain set of rules to be obeyed that aren’t feasible due to the following reasons:

* Team members’ geographical distance
* Team members’ day jobs (member availability is impacted)

However the “Specification by Example” [5] technique is adopted as the Agile approach to understanding the APS requirement.

# Team Dynamics

Joint Application Design (JAD) [WIKI] and development sessions were held twice a week. Members also communicated challenges, insights and progress using instant messaging and email.

The analysis and design was a joint effort. However, implementation was divided into respective subject areas and each member concentrated on a specific context of the problem.

# Analysis

# Problem Context Understanding

To gain a high level understanding of the problem the APS is meant to solve, the call for understanding the problem world the APS is intended to control is imperative. According to Michael Jackson [4], clearly understanding the problem world (The physical domains the machine is target for) is crucial. Figure 1 is a high level context diagram that illustrates the relationship between the machine (APS) and the physical domains that have been identified.

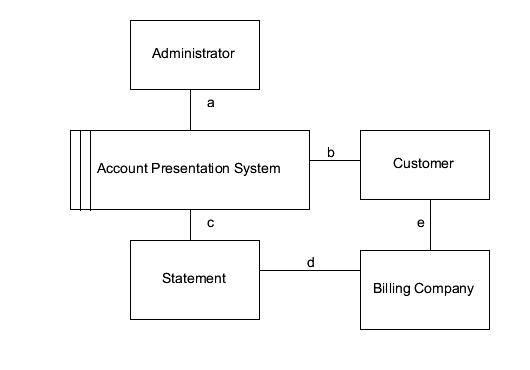


Figure 1: Relationship between the APS and the problem world

The relationships or shared phenomena [4] between the domains in figure 1 are influential in the outcome of the APS specification. Below is high level description of the shared phenomena.

1. An administrator schedules statement scraping, resolve errors.
2. A customer registers with the APS, configures statement viewing, and views statements.
3. The APS scrapes/retrieves statement data and generates a statement.
4. A statement data is from a billing company.
5. A customer has an account with a billing company.

# Specification by Example (SBE)

*“It is a collaborative approach to defining requirements and business oriented functional tests for software products based on capturing and illustrating requirements using realistic examples instead of abstract statements”* ***[ wiki]***

# Reasons for choosing SBE

SBE is adopted for the following factors:

* **Collaborative** – SBE provides guidelines on how to effectively analyze the vague system requirements and derive scope from the business goal [13].
* **Testing** – The key examples become inputs to Unit Testing **[cite Gerard Meszaros]**
* **Living Documentation** – The documentation captures the system’s iterations and extensions. The resulting documentation reflects a common understanding each stakeholder’s interest in the system.
* **Managing Change** – specification changes are facilitated by relatively low overhead in refactoring **[cite Martin Fowler]**.
* **Ambiguity** – avoid using only natural language as means for specification because natural language is ambiguous and context dependent.

# How SBE is applied

From the business goal stated in the project brief [1] and the investigation done thus far with regards to the APS domain, website scraping, and the context diagram; the business goal is presented below as follows:

**Business Goal**

Provide customers with the single point of access to view their

aggregated statements.

**Scope**

* **Given**: APS customer has successfully logged in by the APS system.
* **And** : have an active e-billing account with the billing company
* **And :** statement Data is available
* **When :** APS customer wants to view aggregated account information.
* **Then:** accounts information is displayed

Table1 Illustrates key examples of the APS high level specification

Table 1: Key examples of the APS high level specification

|  |  |  |  |
| --- | --- | --- | --- |
| **Successful Login** | **Active e-billing** | **Data available** | **View Data** |
| Yes | Yes | Yes | Yes |
| Yes | No | No | No |
| Yes | Yes | No | No |
| No | Yes | Yes | No |

This high level scope is further decomposed into sub scopes that will collaborate to fulfill the business goal. Refer to Appendix A for the sub scopes

# Design

# Test Driven Development (TDD)

# Why TDD

The choice of TDD is based on the following factors:

* It necessitates for “fail early [Martin Folwer]”. This allows for quick and flexible reaction to change.
* It maps well on to SBE. The key examples serve as input to TDD. Automate test are used to verify the specification.
* It’s iterative development. “It progressively refines the implementation of features in response to feedback until they are good enough” [Steve Freeman].
* It facilitates communication. By applying the “Red, Green, and Refactor” [Kent Beck], feedback is provided at the end of each iteration to verify that the requirement is fulfilled.
* It’s tool for living documentation. The changes from each refactoring are applied to the specification.

# How TDD is applied

The scenarios derived in section 4.2.2 are used as artifacts to specify the

unit tests. Figure 2 illustrates the TDD process steps used for the iterative

development of the APS solution.

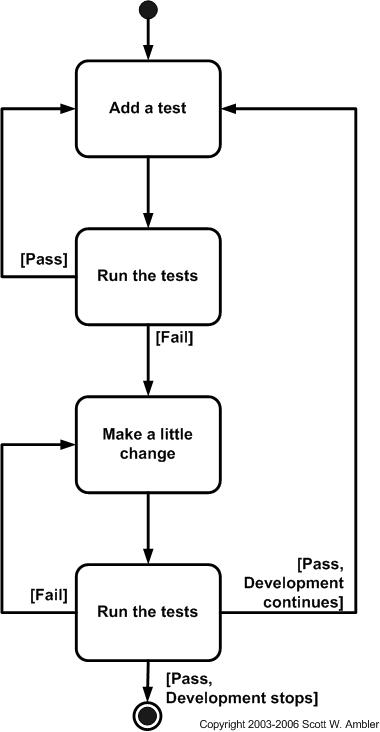


Figure 2: TDD steps followed. Adopted from Source: <http://www.agiledata.org/essays/tdd.html>

For a comprehensive listing of the test cases, please refer to Appendix B.

# Domain Driven Design

Provide Definition

The choice of the DDD technique is based on the following:

* To gain a clear understanding of the business domain
* Defines a ubiquitous (common) language shared by all stakeholders.
* Helps in clearly defining system boundaries using bounded contexts [Eric Evans].
* Avoids the Anemic domain model anti-pattern [5] by promoting the existence business logic in the domain layer.
* Fits in with well with SBE and TDD. According Gojko [13], a specification must be written in a common language that everybody understands. The common language is what DDD refers to as the Ubiquitous language

# Ubiquitous Language

Table 2 shows the terms the form the APS Ubiquitous language.

Table 2: The APS Ubiquitous Language

|  |  |
| --- | --- |
| **Term** | **Description** |
| Customer | Person who register to APS to view statement |
| Billing Account | eBilling Account |
| Administrator |  |
| Billing Account Statement |  |
| Scheduler |  |
| Scraper |  |
| Credentials |  |
| Billing Company |  |
| Scrape Errors |  |
| Notification |  |
| Account Status |  |
| Maintenance Window |  |
| Company Type |  |
| Peak period |  |
| Billing Cycle |  |
| Lead Time |  |
| Scrape Log |  |

# Domain Model

# Entities and Value Objects

Figure 4 Show the APS domain model Entities and Value Object.

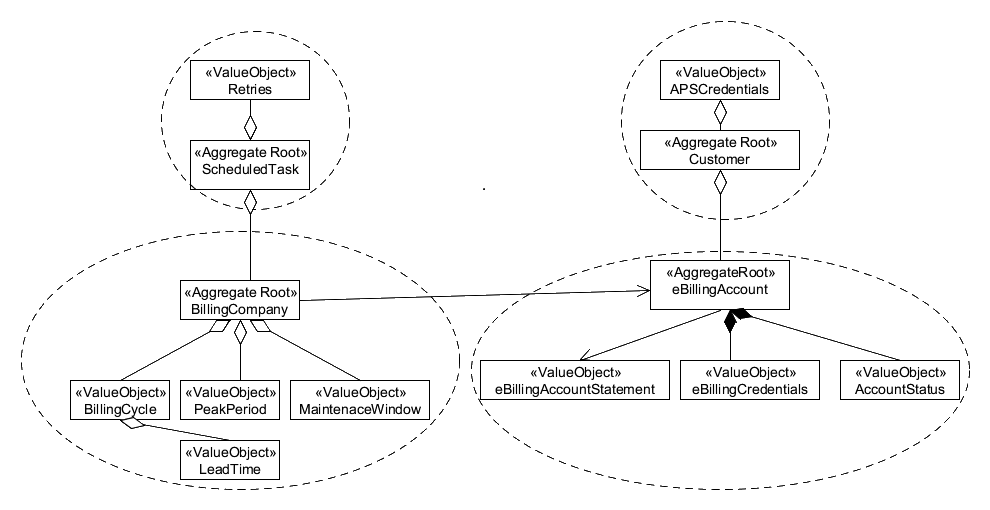


Figure 4: APS Domain entities and value objects

The objects that are modeled as entities are as such because they identifiable and mutable.

The objects that are modeled as value objects are as such because they are immutable.

# Repositories

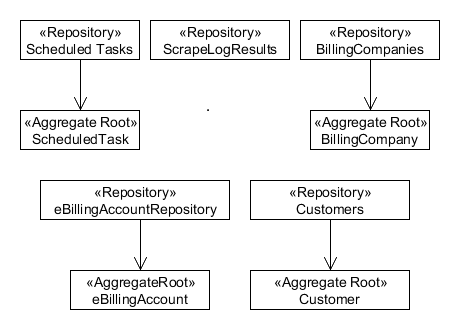


Figure 5: APS Domain Repositories

# Factories

# Domain Services

# Layered Architecture

A layered architecture promotes the separation of concerns from a systems engineering point view. The APS layered architecture is comprised of the following:

Table 3: APS Domain Architectural Layers

|  |  |
| --- | --- |
| **Layer** | **Objects** |
| User Interface | APS Login, Registration, statement configuration and viewing, admin |
| Application |  |
| Domain |  |
| Infrastructure |  |

The

# Implementation

# Tools

This section outlines the tools and frameworks used to implements the APS solution.

* Java 2 Standard Edition (J2SE) [Oracle] as the programming language
* Eclipse Integrated Development Environment (IDE) [Eclipse.org] as the code editor.
* Spring framework [Spring.org] for dependency injection and scheduling
* JMock [JMock guys] & Mockito [Mockito guys] for test mock objects
* Junit [Junit.org] to write unit tests
* GitHub [gitHub.com] as a collaboration and version control platform.
* Derby [Apache] as a light weight Relational Database Management System (RDBMS) [Cite RDMS]
* Xtensible Stylesheet Language Transformation (XSLT) [W3C.org] for transformation of Xtensible Markup Language (XML) [W3C.org] to Hyper Text Markup language (HTML) [W3C.org]

# Design Patterns

This section outlines the Design Patterns [Gang of Four] used to implements the APS solution.

* Factory :

Used to encapsulate of Object creation

* Strategy :

Used to encapsulate the varying scraping algorithms

* Master/Worker:

The Master/Worker is used to implement the scheduling of scrape tasks. It is realised as Thread pool/Command pattern

* Singleton:

Only one instance of Thread pool can be created.

* Adapter:

Used to transform scraper interface into interface APS domain can understand

* Builder:

Used in place of the factory when complex objects have to be created in a step-wise manner

* Observer :

Used for event notifications

* Dynamic Proxy:

It is used to front for other objects to separate cross-cutting concerns for the business domain.

Comprehensive descriptions of the usage of each pattern are covered in each group member’s individual reports [6, 7, 8, 9, 10, 11].

# SOLID Principle

This section provides information on how the SOLID principles [Uncle Bob] are considered.

# Single Responsibility Principle (SRP)

Classes are designed with only reason to change [Uncle Bob]. The *ScrapeManagerImpl* class [11] is a good example of how the SRP has been considered.

# Open Closed Principle (OCP)

Classes are closed to modification but open to extension [Uncle Bob]. The *User* class [10] and its *Customer* child class are good examples of how the OCP has been considered.

# Liskov Substitution Principle (LSP)

Derived classes are substitutable for their base classes [Uncle Bob]. Again, the *User* class [10] and its *Customer* child class are good examples of how the LSP has been considered. The Customer class is substitutable for the User.

# Interface Segregation Principle (ISP)

Interfaces are fine grained and are client specific [Uncle Bob]. The *ScrapeManager* interface [11] is a good example of how the ISP has been considered.

# Dependency Inversion Principle (DIP)

Where need be, the APS code depends on abstractions not on concretions [Uncle Bob], and the abstractions do not depend upon details [Uncle Bob]. The details should depend on abstractions [Uncle Bob]. The examples provided in the sections 6.3.1, 6.3.2, and 6.3.3, are good examples of how the DIP has been considered.

Comprehensive descriptions of the usage of each SOLID principle are covered in each group member’s individual reports [6, 7, 8, 9, 10, 11].

# Dependency Injection

The APS implementation appropriately applies on dependency injection. The dependency injection framework used is the Spring Framework [Spring.org].

The classes do not know about their dependencies other than consumption of these dependencies [Josh Slides]. Appendix D shows a typical Spring Framework configuration file detailing how all the dependencies have been configured.

# Data Integrity and Error Handling

DDD specifications are used to verify and validate the integrity of data. An example of such a specification is: ***TelecoScrapedResultAdditonSpecification***.java [6].

Java’s Exception handling [reference] mechanism is used for error handling. The try-catch-finally [reference] blocks are used.

# Discussion

The material presented in the course [1] has been applied to solve the APS problem. A Problem Context Diagram, SBE, TDD, DDD, Design Patterns, SOLID principles, and Dependency Injection have been used to demonstrate how the analysis, design, and implementation of the APS requirement is fulfilled.

SBE has yielded a specification with key examples that can be input to TDD, DDD has yielded models that clearly map on to APS problem domain. Design patterns and SOLID principles have helped to produce good cohesive and extensible software components.

The

The resulting system however does not have on a Graphical User Interface (GUI) [GUI guys]. Emphasis has been put on the core business problem domain modeling with the notion that if this domain is modeled right, adding GUI should not prove cumbersome.

* SBE challenges
* TDD challenges
* DDD challenges (Shared Kernel)
* Design Patterns challenges
* SOLID challenges

# Conclusion

# References

[1] ELEN 7045

[2] Course Project

[3] Agile

[4] M.Jackson, Problem Frames: Analyzing and structuring software development problems, Addison-Wesley, 2001

[5] SBE

[6] bak

[7] Livious

[8] Ronald

[9] sibusiso

[10] silas

[11] Tumi

[12]Martin Fowler

[13] Gojko

**Appendix**

# APS Specification by Example scenarios and key examples

# APS unit tests

# APS DDD Layered Architecture

# Spring Framework dependency injection configuration