Software Engineering for Industry Coursework 2: AcmeTelecom Billing System

Yufei Wang (yw6312, s5)
Paul Gribelyuk (pg1312, a5)
Yawei Li (yl8012, s5)
Jun He (jh1212, s5)
Xiaoxing Yang (xy212, s5)

December 4, 2012

1 Introduction

This report covers the analysis, testing, refactoring, and re-engineering of the AcmeTelecom billing system. We saw our responsibility as two-fold: to build a well-tested, easy to maintain, coherent piece of software, and to use that software to implement a billing algorithm which fairly bills clients according to the time they spent calling during peak and off-peak times. We relied heavily on Test Driven Development (TDD) practices in the way we devised tests, using unit tests to assert the state of our system, and mock tests to verify its behaviour. To put in place tests for the initial code base, we were forced to make decisions about design patterns present (such as the singleton pattern in `HtmlPrinter`) as well as make heavy use of interfaces to allow the JMock libraries to interact with out code. Once satisfied with the initial functionality, we wrote integration tests and opened the development process to the business expert by writing a DSL around the components most responsible for how the billing system determines rates for customers.

We used these specificaitons to guide us in the design phase, where we ultimately chose to preserve the structure of the `BillingSystem` object as the class which is the central component collecting call information and passing them on to the `BillGenerator` rather than deciding what those calls shoudl cost. We externalized that business logic by putting it in a separate class, thus making the original class more cohesive and simplifying the process to incorporate future changes. We used a modern version control system (*git*) and a continous integration server to manage the testing framework and automatically build after each commit. Dependency matrix analysis helped to gain a big-picture understanding of the codebase without reading every line.

We were also careful not to over-optimize and decided to keep the general design of the system in tact, since it also inter-operates with other external libraries throughout the AcmeTelecom organization. If we were to implement a brand new design, we would have loosened the coupling further by using Dependency Inversion model wherever possible and definining how interfaces should communicate rather than how concrete classes should change state.

2 Analysis of Original Code

The original code for the Billing System was well structured, though lacking any tests, so a look at the dependency structure was needed to begin to analyze where work would need to be done. We have included a diagram of this here: i

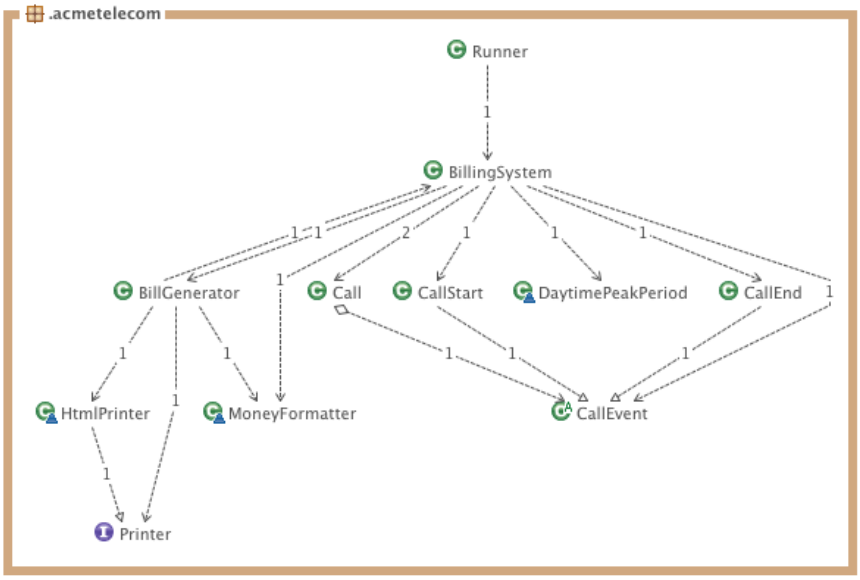

Figure 1: Original Dependency Structure of the Billing System

The 3-layer architecture is preferable in this case, since it is an in-house system with a small well-defined userbase and more a intricate architecture pattern would not be needed. The code also lacks interfaces to begin to easily write tests, especially with JMock, which requires an interface to mock behaviour.

The `CallEvent` is the basic building block of a `Call`, which consists of a `CallStart` and a `CallEnd`, which both inherit from `CallEvent`. The `BillingSystem` receives method calls to `callInitiated` and `callCompleted` and logs them to a `List<>` object. A call to the `createCustomerBills` calculates the costs of each call as it occurs in teh list, and then dispatches this list of `Calls` to the `BillGenerator` by invoking `send`. The `BillGenerator` uses an `HtmlPrinter` to output the results into HTML (in this example, to `System.out`). Other classes, such as `MoneyFormatter` and `DaytimePeakPeriod` act as helper classes.

3 Refactoring and Testing

Creating interfaces for testing purposes Creating Mock tests Creating Unit tests for container classes Breaking `HtmlPrinter` singleton interface and changing constructor to take a different `PrintStream` other than `System.out`.

Extracted calculation part of original `BillingSystem` into a new class to allow flexibility in future design.

4 Creating a DSL

5 Implementing the New Billing System

6 Acceptance Tests

- GivenPeakPeriod:
- GivenTheFollowingCustomers:
- GivenTheseCallsAreMade:
- GivenTheSystemIsInitialized:
- TheBillShows:

7 Conclusions and Recommendations

Tools used (Git, Jenkins, IntelliJ Coverage Report, IntelliJ Dependency Matrix). Problems enountered like refactoring for Tiny Types, Mocking with concreted classes rather than interfaces.