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

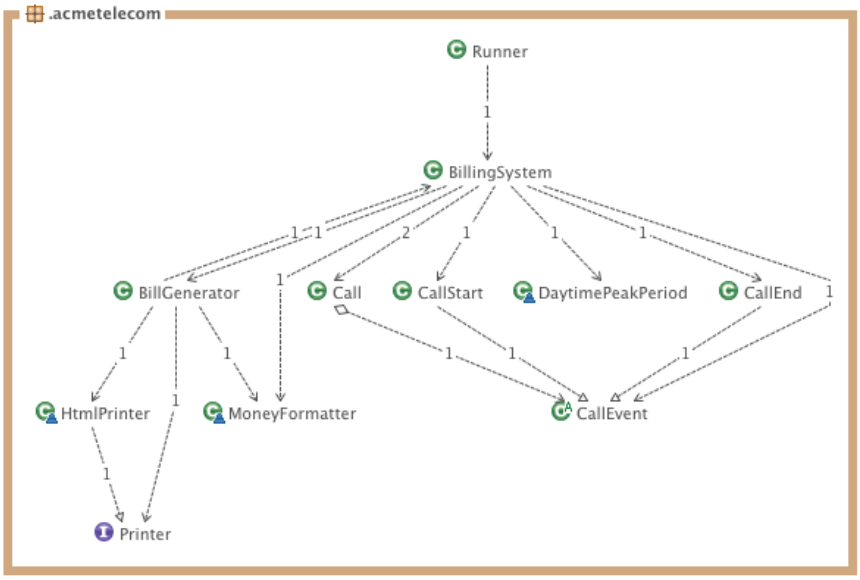


Figure 1: Original Dependency Structure of the Billing System

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

The original codebase design did not take tests into consideration. Therefore it is difficult to write unit tests and mock tests. One main factor is that the singletons `CentralCustomerDatabase` and `CentralTarriffDatabases` in `BillingSystem` class, and the singleton `HtmlPrinter` in BillGenerator prevent building mock tests for these classes. Another issues is dependency on concretion due to the lack of abstract layers, which causes high coupling and difficulty in test. The default time stamp for call initiation and completion is given a direct reference to system clock. However this reduced the efficiency of testing because we have to wait for each call to run its whole time.

We make several refactoring to tackle the problems described above. First we pull out all the calculation partition of the `BillingSystem` and put it in a new class called `BillingSystemLogical`. Then the function of `BillingSystem` is just to create an instance of `BillingSystemLogical` and to inject the objects and dependencies to it. Now the `BillingSystemLogical` is decoupled with the singletons `CentralCustomerDatabase` and `CentralTarriffDatabase`. For `BillGenerator` we make a constructor for it, taking one parameter which can be any instance of `Printer` interface. The bill will be printed via the given printer instance. In production, a `HtmlPrinter` will be passed to the generator, while in testing, we passed a `FakePrinter` to it.

```
public BillingSystem(){
    billingSystemLogical
            = new BillingSystemLogical(
                CentralCustomerDatabase.getInstance()
                ,CentralTariffDatabase.getInstance()
                ,new BillGenerator(HtmlPrinter.getInstance())
                ,new PeakSeperateOffPeakRateEngine());
}
```

The dependency on concretion is solved by introducing interfaces between coupling classes. We extract four main methods of `BillingSystem` into `Biller` interface. `BillingSystem` and `BillingSystemLogical` as mentioned above both implementing the `Biller` interface. Messages and requests send to `BillingSystem` will be passed to `BillingSystemLogical` for specific operation. We extract a `Generator` interface out of `BillGenerator` for later fake testing and mock testing. A part of rate calculation code was pulled out of `BillingSystemLogical` and made a rate calculator class called `ProfitableRateEngine`. Then we create an interface out of it called `RateEngine`. The core of our new billing system, which is a rate engine, is a class implementing this `RateEngine` interface.

A configurable time stamp could improve the test efficiency. This feature is implemented by generate a getter and setter methods for time stamp in `BillingSystemLogical` instead of using the default system clock. Thus in testing, the methods `callInitiated()` will use default system clock as a time stamp. Then we can avoid the waiting time by directly set the time to completion time.

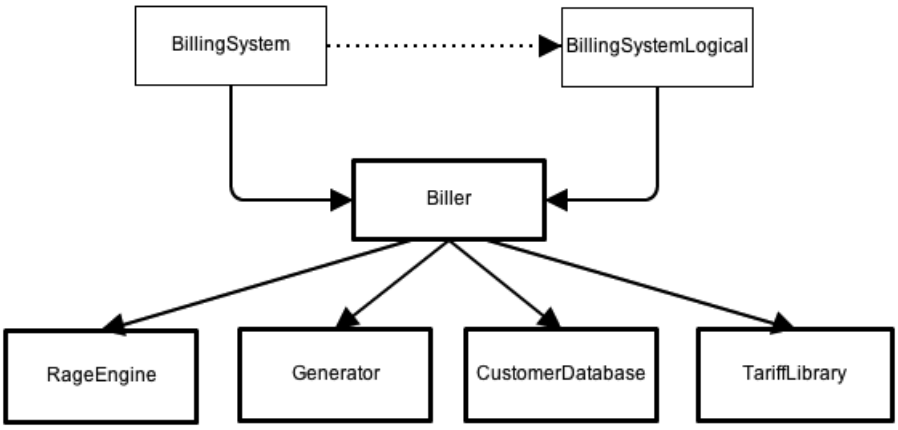


Figure 2: Structure after Refactoring

4 Creating a DSL

The original code base use primitive types directly for, for instance, caller, callee, and cost. We thought that was not very comprehensive and could be easily used in wrong order. The solution is introducing tiny types. The class `Caller` and `Callee` are the replacement of `String`, which both have a `toString()` method to get the name. Similarly, we created `Cost` class in replacement of `BigDecimal` which has addition, comparison and getter methods.

Several classes exist whose constructor takes three or more parameters or the parameters have the same type. For example, the constructor for `Customer` takes three string parameters which can be very confusing when initiating an object. For a class as such, we create a builder for it. In total, we have five builders: `CallBuilder`, `CallEndBuilder`, `CallStartBuilder`, `CustomerBuilder`, `LineItemBuilder`.

```
Customer cus1
        = CustomerBuilder.aCustomer()
                .withFullName("Alice")
                .withPhoneNumber("07777777777")
                .ofPricePlane("Default plan")
                .build();
```

5 Implementing the New Billing System

As mentioned in the "Refactoring and Testing" section, we extracted the fee calculation part out of the the `BillingSystem` class. A new package `rate` was created under `com.acmetelecom`, dealing with specific cost calculation. An interface `RateEngine` was provided to communicate with other parts of the system.

```
public interface RateEngine {
    public Cost calculateCost(Call call, Tariff tariff);
}
```

The interface only contains one public method which requires a `Call` and the `Tariff` for that call and returns the `Cost`. There are several advantages of extracting the cost calculation into a seperate package. Firstly, we decoupled the original system, making it more reusable. Additionally, the whole system is more flexible to be implemented with different rate calculation in the future because different implementations are possible for one interface. This feature immediately helped us when implementing the new billing system. The concrete implementation of the original cost calculation logic was built into the `ProfitableRateEngine` class, which will compute cost under the peak rate as long as the call duration is touching peak. In order to calculate the cost fair to customers, alternatively, we created a new implementation of `RateEngine`, called `PeakSeperateOffPeakRateEngine` to tackle the new cost calculation standard. The switching cost of the original billing system then is very low as we can easily achieve new features by changing `RateEngine` instance initialization from `ProfitableRateEngine` to `PeakSeperateOffPeakRateEngine` without breaking any other codes.

All new cost calculation logics were handled within the new rate engine implementation. The general idea is that we introduced two variables `peakTime` and `offPeakTime`. For different call lasting situations, in order to get cost, we multiply the `peakTime` and `offPeakTime` to different rates. The difficult point for this idea is that, we had to analyze the call duration such that we can accurately seperate the peak time and off peak time. For making the class tidy and readable, a helper function `calculateDuration(Call call)` was added to analyze the call duration. Since we relied on the TDD development process, we've prepared all possible call senarios (mentioned later).It is convinient to program the duration logic based on the senarios. There are basically two situations; a call finished in one day and a call lasted overnight. For each situation, different conditions could be applied around the peak start point and peak end point. In the algorithm, we make use of the `Calender` API, helping us retrieve the Hour, Miniute, and Second from a `Date` type. By comparing the call start point and call end point to those critical peak points, we differentiate various situations and figure out the peak time and off peak time.

6 Acceptance Tests

We adopted the Framework for Integrated Tesing (FIT) to exam our end-to-end test. In FIT html requirement documents, we wrote 9 different cases according to different factors, i.e. starting point, ending point, and duration (As shown below). This covered all possible situations a customer may come across. i

In the requirement documents, by default, the system is initialized as peak time starts at 7:00:00 and ends at 19:00:00. A customer information table will be given to represent all customers that are eligible to have a bill. Different calls are speicified in the Call table, which simulates the call process. The last table is showing the bills and this is the table that will be examed by the test. In order to execute FIT test, we wrote some classes to parse these tables.

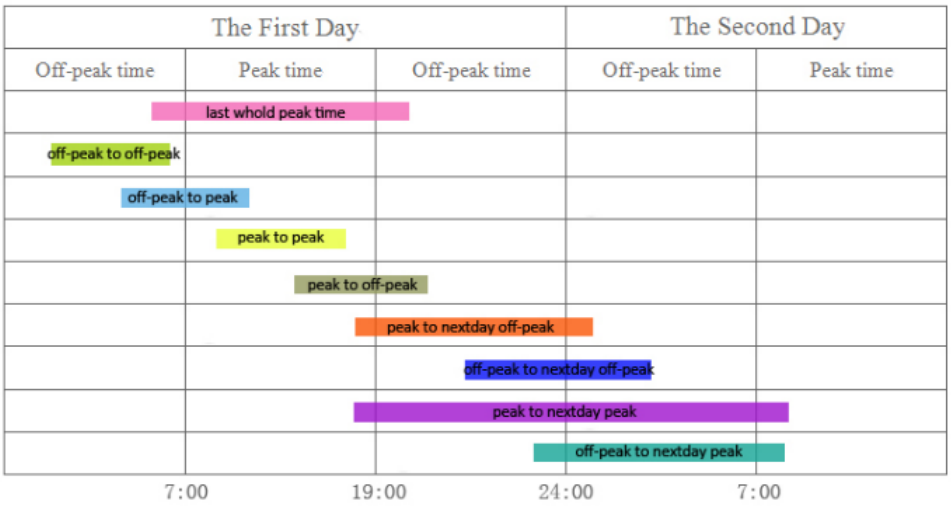
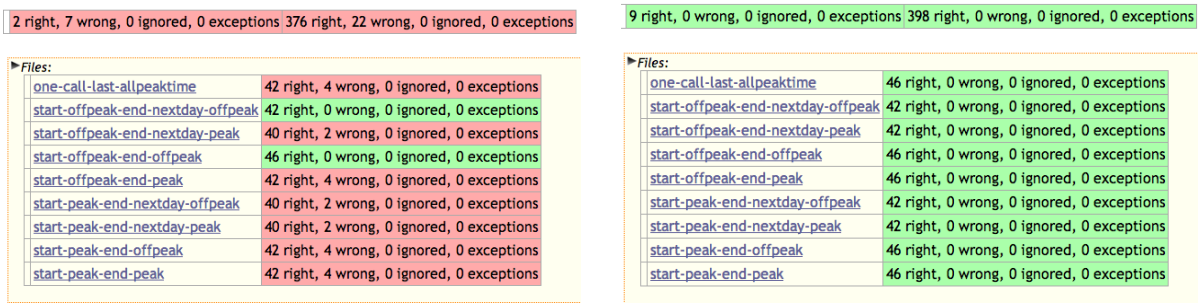


Figure 3: Acceptance-Test Scenario (assuming 7:00-19:00 peak period)



2 right, 7 wrong, 0 ignored, 0 exceptions | 376 right, 22 wrong, 0 ignored, 0 exceptions

Files:

one-call-last-allpeaktime	42 right, 4 wrong, 0 ignored, 0 exceptions
start-offpeak-end-nextday-offpeak	42 right, 0 wrong, 0 ignored, 0 exceptions
start-offpeak-end-nextday-peak	40 right, 2 wrong, 0 ignored, 0 exceptions
start-offpeak-end-offpeak	46 right, 0 wrong, 0 ignored, 0 exceptions
start-offpeak-end-peak	42 right, 4 wrong, 0 ignored, 0 exceptions
start-peak-end-nextday-offpeak	40 right, 2 wrong, 0 ignored, 0 exceptions
start-peak-end-nextday-peak	40 right, 2 wrong, 0 ignored, 0 exceptions
start-peak-end-offpeak	42 right, 4 wrong, 0 ignored, 0 exceptions
start-peak-end-peak	42 right, 4 wrong, 0 ignored, 0 exceptions

9 right, 0 wrong, 0 ignored, 0 exceptions | 398 right, 0 wrong, 0 ignored, 0 exceptions

Files:

one-call-last-allpeaktime	46 right, 0 wrong, 0 ignored, 0 exceptions
start-offpeak-end-nextday-offpeak	42 right, 0 wrong, 0 ignored, 0 exceptions
start-offpeak-end-nextday-peak	42 right, 0 wrong, 0 ignored, 0 exceptions
start-offpeak-end-offpeak	46 right, 0 wrong, 0 ignored, 0 exceptions
start-offpeak-end-peak	46 right, 0 wrong, 0 ignored, 0 exceptions
start-peak-end-nextday-offpeak	42 right, 0 wrong, 0 ignored, 0 exceptions
start-peak-end-nextday-peak	42 right, 0 wrong, 0 ignored, 0 exceptions
start-peak-end-offpeak	46 right, 0 wrong, 0 ignored, 0 exceptions
start-peak-end-peak	46 right, 0 wrong, 0 ignored, 0 exceptions

Figure 4: FIT Results Comparision

Class Name:	Description:
SystemUnderTest	Create instances for testing
GivenTheSystemIsInitialized	Reset the test system
GivenPeakPeriod	Set the peak time period
GivenTheFollowingCustomer:	Store customer information into FakeCustomerDatabase
GivenTheseCallsAreMade:	Record calls into billing system
TheBillShows:	Generate customer bills for comparision

It is worth noting that we added three fake classes `FakeCustomerDatabase`, `FakeGenerator`, `FakePrinter`, to facilitate our FIT test. This is because CentralCustomerDatabase is a singleton from other system and we do not have access to that database. While there is a CustomerDatabase interface, we can have our own fake database implementation based on ArrayList to save customers information from FIT requirements. FakeGenerator and FakePrinter are created to fullfil the table output format in the requirement.

When running the FIT scenarios running with the original rate calculation engine, we failed in most FIT test because the actual costs were not produced as expections. After refactoring with the usage of the new RateEngine implementa-

tion, we made all tests pass, which means we met the need of specification.

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.

Conclusions and Recommendations To get our project releasing as a commercial product, we followed the instructions of how to put together a deployment pipeline. During the whole deployment, we used IntelliJ IDEA as the development environment, Git for version control and Jenkins for continuous integration and automatic deployment.

We felt more comfortable to use git for version control and collaboration because we used Git in the last course work and we are more familiar with it than other version control tools. The procedure of our product (codebase) //The dashed line means product without tests (unreliable and fragile) //The continuous line means product after tests (solid and strongly-structured) //This image can be modified if anyone wants to change it

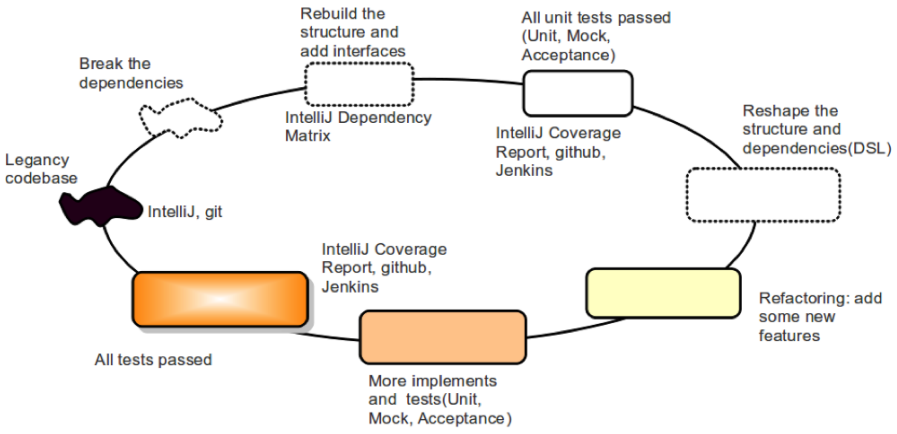


Figure 5: Development Process

i

At first we found the legacy codebase was not well-structured and testable after analysis with the help of IntelliJ Dependency Matrix. We had to break the dependencies and create new interfaces to do unit tests of the main methods and functionalities. Then the shape of the codebase is much clearer after all the tests passed. To achieve the requirements which assigned by Stephen, we should first write requirement analysis report (or RFP requirement for Proposal) at the beginning and then write the acceptance tests. We can monitor the conditions and coverage of the tests by IntelliJ Coverage Report. Before we started refactoring it, itÕs quite important to make our code more like natural language by taking the advantages of domain specific language. During refactoring, we did tests and build

with Ant and integrated by Jenkins to make sure every change we made was correct. WhatÕs more, Jenkins will send us emails if our jobs are failed.

As we assumed that the project might be re-engineered by others in the future, we divided it into different modules and DSL might give a clear concept of our code. (I think if we add some proposals, such as requirement or tests, will make this part more brilliant. Products armed with reports are much more easier for future developers. And it may add some points.)