

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 5, 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 our 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 specifications 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 should 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 continuous 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 defining 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: The 3-layer architecture is preferable in this case, since it is an in-house system with a small well-defined userbase and more an 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.

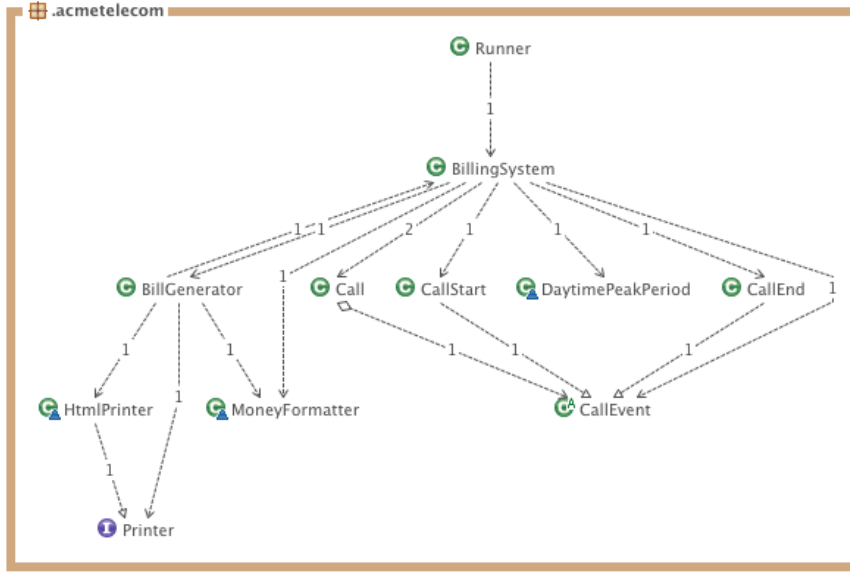


Figure 1: Original Dependency Structure of the Billing System

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 the 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 `CentralTariffDatabases` in `BillingSystem` class, and the singleton `HtmlPrinter` in `BillGenerator` prevent building mock tests for these classes. Another issue 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()
            , CentralTarriffDatabase.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

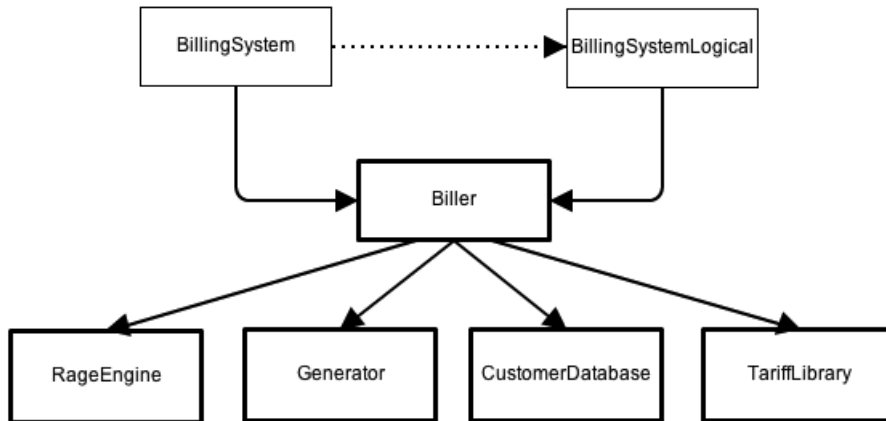


Figure 2: Structure after Refactoring

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.

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("0777777777")
        .ofPricePlane("Default plan")
        .build();
```

5 Implementing the New Billing System

As mentioned in the "Refactoring and Testing" section, we extracted the fee calculation component from the **BillingSystem** class. A new **rate** package was created under **com.acmetelecom**, to handle cost calculations, with a **RateEngine** interface provided to communicate with other parts of the system.

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

The interface contains a public method taking a **Call** and a **Tariff** for that call as inputs and returning its **Cost**. There are several advantages to isolating the cost calculation in a separate package. First, this breaks the close coupling initially present along functional components, making the resulting system more reusable. Also, this approach increases the flexibility implement different rate calculations in future iterations. Lastly, we were able to use this to implement the rate calculation methodology in the billing system. Initially, the original cost calculation logic built into the **ProfitableRateEngine** class computes the cost at the peak rate if the call overlaps with the peak period. To calculate the fair cost to customers, in compliance with new regulations, we created a new implementation of **RateEngine**, called **PeakSeperateOffPeakRateEngine**. Thus, the development cost of switching to this new billing system is very low as we can and future rate changes only involve creating new classes implementing **RateEngine** without any further changes in the code.

This new rate engine handles all cost calculation logic. The variables `peakTime` and `offPeakTime` keep track of how much time is spent during the respective billing periods, and the final cost of the call is then the product of these variables by the rates being charged at these respective times:

$$C = t_{peak} \cdot p_{peak} + t_{off-peak} \cdot p_{off-peak} \quad (1)$$

The challenging component of implementing this is the calculation for the call duration to accurately separately between the peak time and off peak time. To make the class tidy and readable, we added the `calculateDuration(Call call)` helper function to handle this calculation. Using the TDD development process, we wrote tests to cover every combination of peak and off-peak times during a 24 hour period. It is convinient to program the duration logic based on the 9 senarios. Generally, calls can either end on the same day they start or on the following day. In each case, different conditions could be applied around the peak start point and peak end point. In the algorithm, we make use of the built-in Java `Calender` class, to retrieve the hour, minute, and second from a `Date` object. By comparing the call start point and call end point to those critical peak points, we differentiate distinct situations and figure out the peak time and off peak time.

6 Acceptance Tests

We adopted the Framework for Integrated Tesing (FIT) to conduct our end-to-end testing. The FIT html code specifies 9 different tests to account for different starting points, ending points, and durations (As shown below). This covers all possible scenarios in a 24 hour period. By default, the requirement documents initialize the

The First Day			The Second Day	
Off-peak time	Peak time	Off-peak time	Off-peak time	Peak time
	last whold peak time			
off-peak to off-peak				
off-peak to peak				
	peak to peak			
	peak to off-peak			
		peak to nextday off-peak		
		off-peak to nextday off-peak		
		peak to nextday peak		
		off-peak to nextday peak		
7:00	19:00	24:00	7:00	

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

peak time to start at 7:00:00 and end at 19:00:00. A customer information table is 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 shows

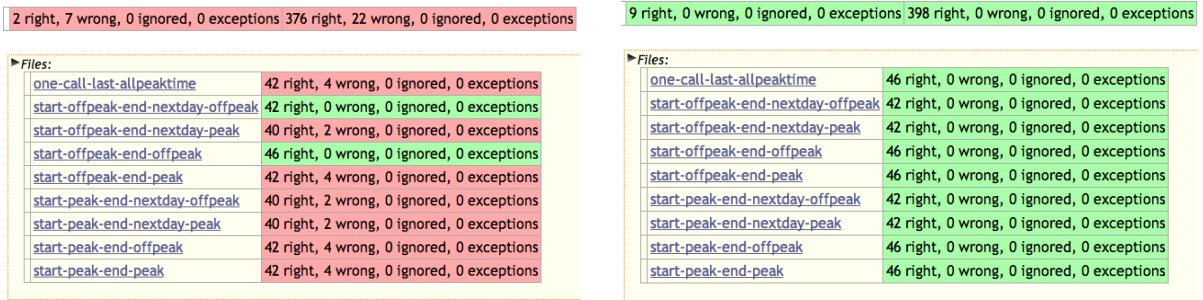


Figure 4: FIT Results Comparison

the bills and this is the table that will be examined by the test. In order to execute FIT test, we wrote some classes to parse these tables.

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 comparison

We also added three fake classes `FakeCustomerDatabase`, `FakeGenerator`, and `FakePrinter`, to facilitate our FIT test. This is because `CentralCustomerDatabase` is externally implemented using a singleton pattern, and does not allow instantiation in a testing environment. While there is a `CustomerDatabase` interface, we created our own fake database implementation (with an `ArrayList` underlying data structure) to save customer information for FIT purposes. `FakeGenerator` and `FakePrinter` are created to provide the table output format in the requirement.

As we showed in Figure 4, most of the FIT scenario tests failed with the original rate calculation engine, as one would expect given that the system was not configured for the new requirements. After refactoring, and with the new `RateEngine` implementation, all tests passed, showing that the system is now compliant with the new requirements.

7 Conclusions and Recommendations

Throughout this assignment, we adhered to the software engineering principles learned in lectures. We used tools such as git (source control), Jenkins (continuous integration), JUnit and JMock (testing), IntelliJ Dependency Matrix (code structure analysis), IntelliJ Coverage Report (assessing how much of the project has been tested), and FIT (for acceptance tests). Furthermore, we adhered to sound software

engineering principles by using appropriate design patterns such as tiny types for **Caller**, **Callee**, and **Cost**, and dependency inversion for **RateEngine** calculations. One of the problems we encountered was in the refactoring stage of the development, when we attempted to add tiny types. Because the external libraries return **BigDecimal** for cost and **String** for a customer phone number, we had to be very careful to what extent these tiny types were used throughout our code. Although JMock requires class interfaces to work, we found a way of mocking a concrete class for cases where objects from external libraries needed to be mocked and no interface was available.

Tools used (Git, Jenkins, IntelliJ Coverage Report, IntelliJ Dependency Matrix). Problems encountered 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

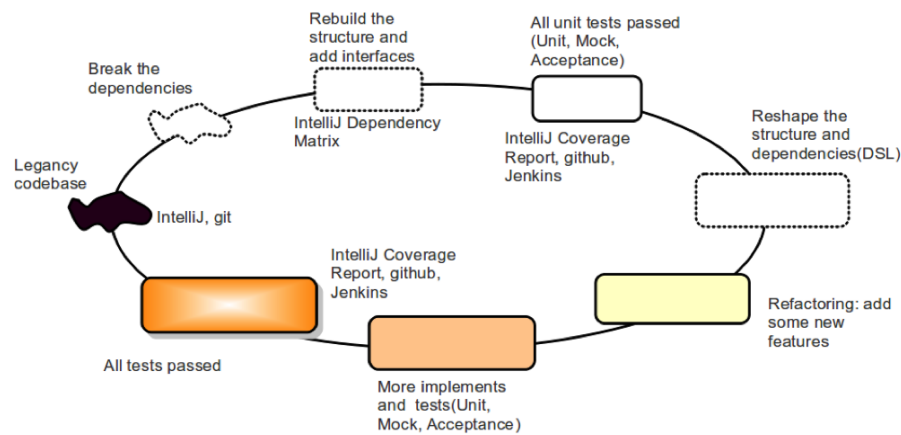


Figure 5: Development Process

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 be-

ginning 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, it's 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. What's 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.)