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 6, 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 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 externalised 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 organisation. 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 original 3-layer architecture is preferable in this case, since it is an in-house system with a small well-defined userbase, so a more intricate architecture would not be needed. The absence of interfaces meant that writing tests without refactoring was not possible, especially mock tests which require interfaces to mock behaviour.

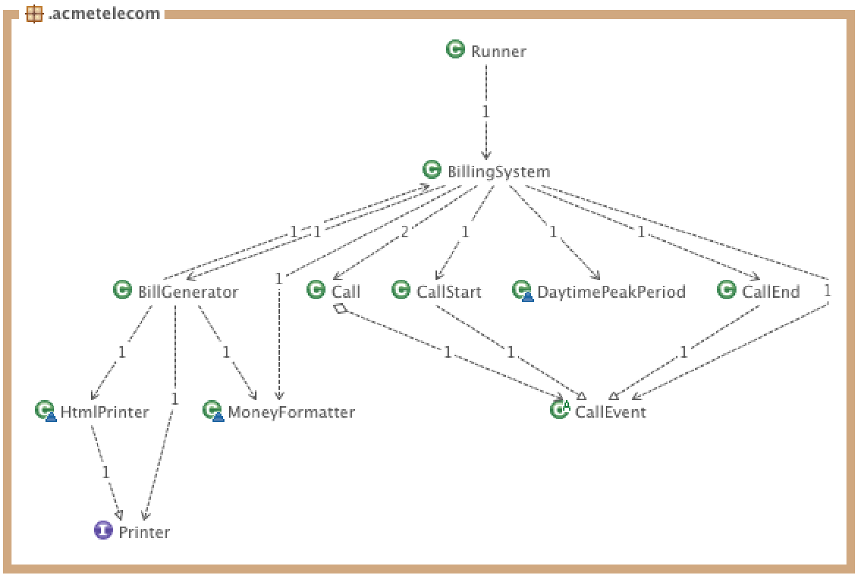


Figure 1: Original Dependency Structure of the Billing System

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 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

To write tests, we first needed to extract interfaces to objects requiring tests. Furthermore, the external classes `CentralCustomerDatabase` and `CentralTarriffDatabases` and the local `HtmlPrinter` use the singleton pattern, which makes the constructor private and does not allow us to test their functionality directly. Another issue we faced is the code's dependency on concretion, which causes high coupling. Further difficult in testing calls also arose in the way the system handled timestamps. It relied on the system clock, thus not allowing us to test calls without waiting for a specified passage of time.

We began with creating `FakeCallStart` and `FakeCallEnd` classes, allowing us to initialise them with specific timestamps, which is useful for testing calls with pre-specified time periods. We then extracted an interfaces for `BillGenerator`, `BillingSystem` and, allowing us to test behaviour of classes which depend on them via mock tests.

Next, we decided to extract the calculation logic from `BillingSystem` into a new class called `BillingSystemLogical`. In our code, the `BillingSystem` is as a front-end which delegates `callInitiated`, `callCompleted`, and `createCustomerBills` messages appropriately. Now, the `BillingSystemLogical` is decoupled from the external database classes. By modifying its constructor to accept `Printer` interface-compatible objects, we inverted the dependency of `BillGenerator` on `HtmlPrinter`. For testing, we chose to break the singleton pattern in `HtmlPrinter` by adding another constructor takes an alternate `PrintStream`, allowing us to write a test to verify what it prints by redirecting the output. Our research showed this to be the best course of action (weblink: *Google Testing Blog*). The bill is printed via the provided printer instance. In production, an `HtmlPrinter` is passed to the generator, while in testing, we passed in a `FakePrinter`. Below is the code we use to create a billing system:

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

The dependency on concretion is reduced by introducing interfaces between coupled classes. We extract four main methods of `BillingSystem` into `Biller` interface. `BillingSystem` and `BillingSystemLogical` both implementing this interface. Messages and requests sent to `BillingSystem` are 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 from it called `RateEngine`. The core of our new billing system, which is a rate engine, is a class implementing this `RateEngine` interface. A configurable timestamp could improve the test efficiency. This fea-

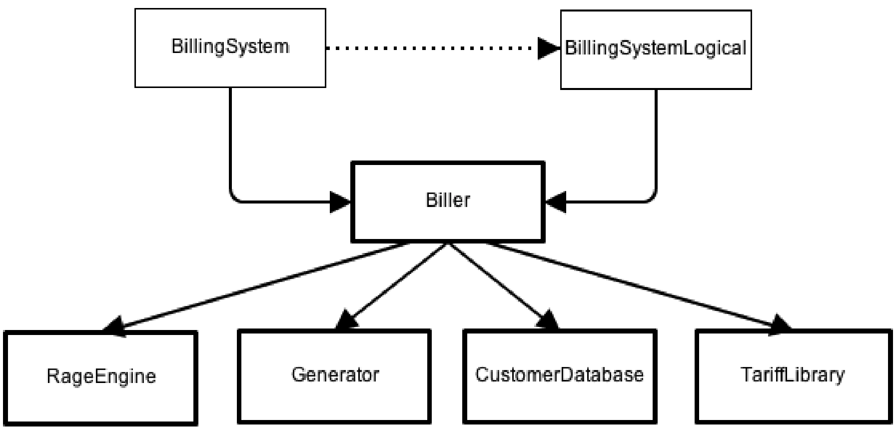


Figure 2: Structure after Refactoring

ture is implemented by generating 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.

Lastly, the original code used primitive types for caller, callee, and cost. This approach is error-prone, so we introduced the tiny types `Caller` `Callee` and `Cost`. This reduces potential for errors and makes the code more readable.

4 Testing

Generally, we used JUnit to test for the state of objects, and JMock to tests for behaviour. We wrote the following tests:

-
-
-
-

5 Creating a DSL

In the original code, 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();
```

The builders perform the following functions:

- CustomerBuilder: this class builds a Customer object by building up phone number, name, and price plan.
- CallBuilder: allows the user to create a Call object using start and end times
- LineItemBuilder: creates a line item object consisting of a call and an associated cost

6 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:

$$C = t_{peak} \cdot p_{peak} + t_{off-peak} \cdot p_{off-peak} \tag{1}$$

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 convenient to program the duration logic based on the 9 scenarios. 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 `Calendar` 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.

7 Acceptance Tests

We adopted the Framework for Integrated Testing (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 initialise

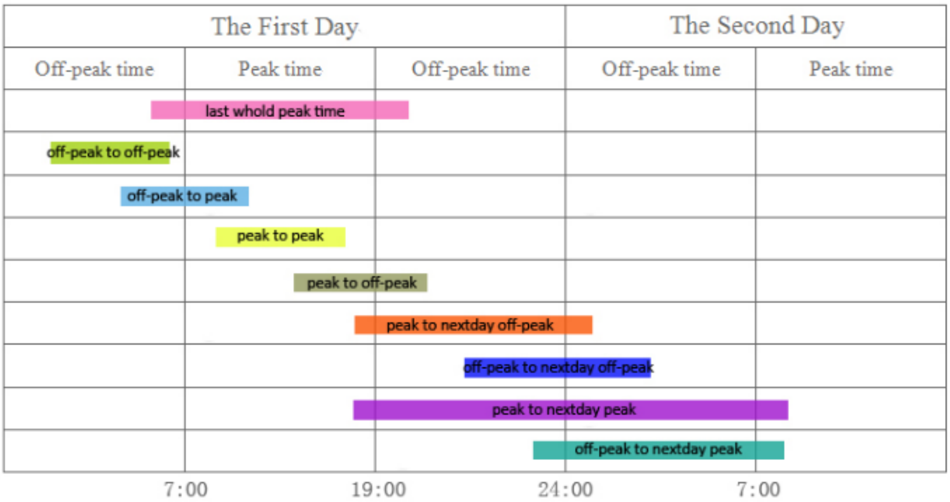
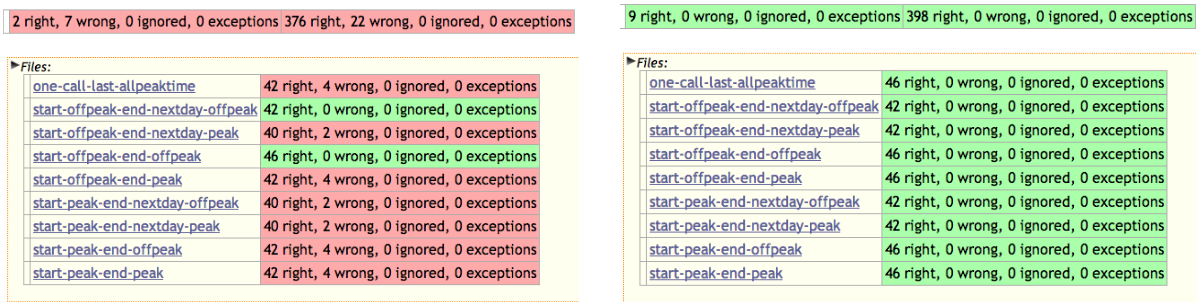


Figure 3: Acceptance-Test Scenario (assuming 7:00-19:00 peak period)

the 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 specified in the Call table, which simulates the call process. The last table shows 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.

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.



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 Comparison

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.

8 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.

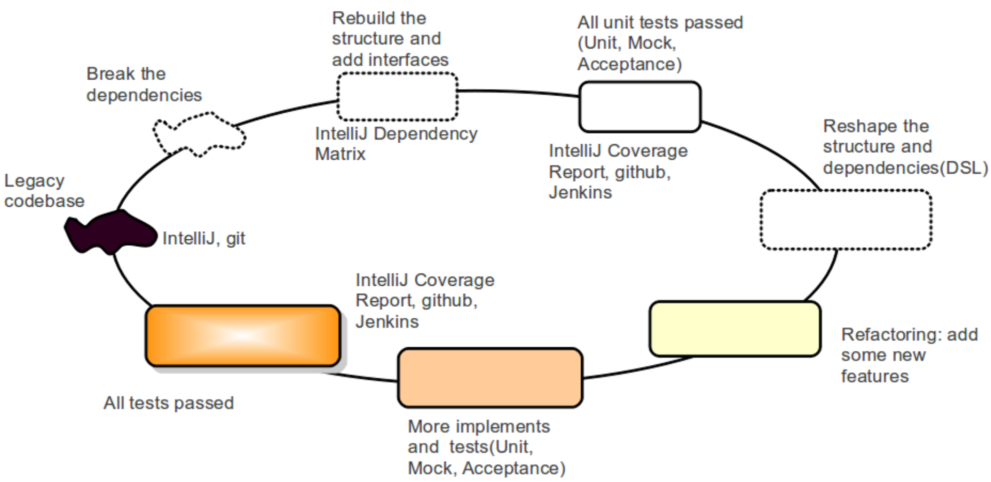


Figure 5: Development Process