

**475 : Advanced Topics in Software Engineering**

**Coursework 2: AcmeTelecom**

**Group members**

|  |  |
| --- | --- |
| Antonio Azevedo | ajf211 |
| Yu Liu | yl12510 |
| Zhicong Liu | zl411 |
| Alexandros Panteli | ap4111 |
| Martin Rouaux | mr2111 |

**1 Analysing the legacy code**

**1.1 Refactoring and unit testing**

The AcmeTelecom codebase we received was not testable at all, this was due to a number of singletons, namely *CentralCustomerDatabase* , *CentralTariffDatabase* from the external system and *HtmlPrinter* form the system. Another significant factor was the dependency to system time, which can be viewed as a dependency to the System singleton. The main strategy was to make small changes to enable us to write unit tests for the legacy code. Having a test suite in place was paramount in order to implement the new feature with confidence.

|  |  |
| --- | --- |
| C:\Users\User\AppData\Local\Temp\attachment.ashx.tiff |  |

**Fig1.** *Structure and class diagram of the legacy code.* *The structure analysis shows the only violating dependency, from BillGenerator to BillingSystem.*

The first step was to analyse the current codebase and get familiar with it. After this, we got comfortable in using a repository since it would have been used throughout the coursework. One of the main goals was to be able to run the *Runner* example as is after adding the new feature; this would ensure we did not change the way the system is invoked.

The results of the runner depend on when the example is ran, and in order to simulate calls we have to actually wait the amount of time specified for each call. To be able to refactor the code we should have tests at hand but at the same time we cannot write any tests. We solved this cycle by doing small refactor-testing changes.

Some of the most important refactorings are shown below:

Firstly the instantiation of the *BillingSystem* is done through an instance of *ConfigurableBillingSystem* that actually contains all the logic. *ConfigurableBillingSystem* itself is decoupled from the external databases. *BillingSystem* is just injecting the dependencies and exists to cater to usage of the system as presented in the Runner example. Below is the new constructor for *BillingSystem*:

|  |
| --- |
| **public** BillingSystem() {  **this**.billingSystem = **new** ConfigurableBillingSystem(  CentralCustomerDatabase.*getInstance*(), // singleton  CentralTariffDatabase.*getInstance*(), // singleton  **new** BillGenerator(**new** HtmlPrinter(System.*out*)),  **new** OffpeakFairRateSelector(7, 19),  **new** SystemClock());  } |

A major issue was the dependency on the System clock. Our solution was to extract a Clock interface and inject instances of *SystemClock* (the real clock) for production code or a *FakeClock* for unit tests and end-to-end FIT tests. The advantage is that the *FakeClock* could be set at any time of the day and move forward and backward as necessary for each particular test. This made the tests run quickly and more importantly, call scenarios could be defined independently of the time of the day when the tests were run.

|  |
| --- |
| Customer c1;  Customer c2;  Calendar cal = Calendar.*getInstance*()  cal.set(Calendar.*HOUR\_OF\_DAY*, 18);  cal.set(Calendar.*MINUTE*, 30);  FakeClock clock = **new** FakeClock(cal);  ConfigurableBillingSystem bs = **new** ConfigurableBillingSystem(  customerDatabase,  tariffLibrary,  generator,  selector,  clock);  bs.callInitiated(c1.getPhoneNumber(), c2.getPhoneNumber());  clock.moveForward(*Minutes*(10));  bs.callCompleted(c1.getPhoneNumber(), c2.getPhoneNumber()); |

Calculation of the rates (used to charge a call) was extracted from the *BillingSystem*. A container class called *RateDescriptor* holds the rates that should be used for the call and what duration of the call should be billed at each rate. Of course, this allowed for the implementation of the new feature. The logic for selecting appropriate rates and duration is delegated to concrete implementations of *RateSelector* (e.g.: *OffpeakFairRateSelector*).

|  |
| --- |
| Collection<RateDescriptor> rates = selector.determineRates(tariff, call);  **for**(RateDescriptor descriptor : rates) {  cost = cost.add(...// Simply summarises rates returned by selector  } |

At this point we installed a CI server (Hudson) that would also run the entire unit test suite on each build. Parallel to refactoring the legacy codebase we started writing the acceptance tests (Fit documents) for the version with the added feature. We did this in order to guide refactoring, for example an issue that emerged from this is that we were depending on the *Tariff* enum. It was really obvious that this was the case when we wanted to specify the tariff rates in the acceptance test document. All the documentation was also versioned by the svn repository (this report, fit documents, figures etc.).

This was continued until we had refactored the whole codebase. Fit documents created by this point showcase the level of decoupling, since the whole environment for the test (e.g. tariff rates, the customers, call times e.tc) could be specified.

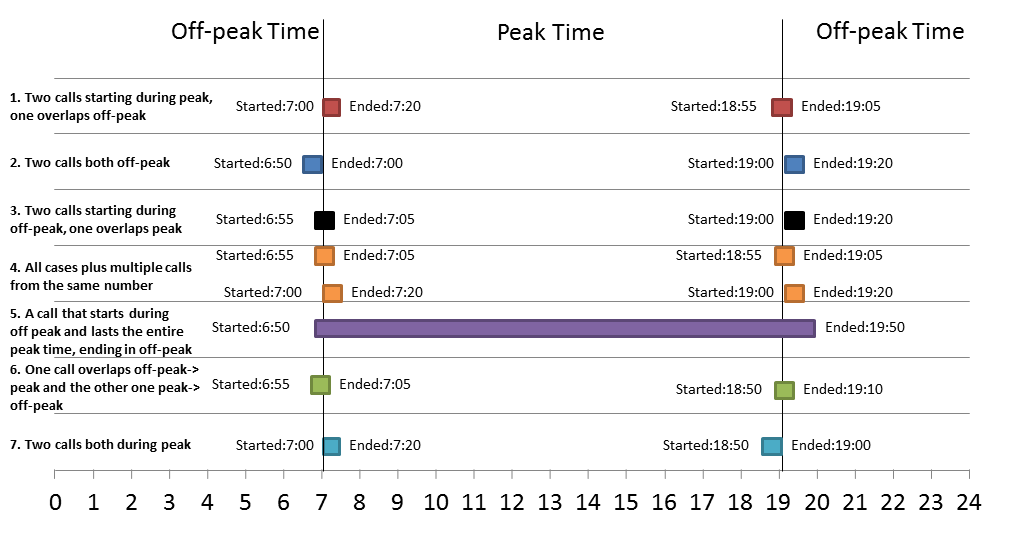
**1.2 Acceptance testing**

Since this application was already “accepted” to be put into production, we deemed unnecessary to write acceptance tests for the existing code. Nevertheless all useful information that could have been extracted by doing so was done by running the acceptance tests for the new version of the system against the older business logic. This was easily achieved by dependency injection in the fixture code.

**2 Adding the new feature**

**2.1 Acceptance testing**

Seven Fit documents were written in total covering a wide range of scenarios. The cases tested are illustrated by the figure below.



**Fig 2***. Phone calls for each acceptance test scenario (assuming 7:00-19:00 peak period)*

The fixture code for these Fit documents was written outside the normal TDD loop and it was straightforward. This was due to the high level of decoupling done beforehand.

As noted in a previous paragraph, running these documents against the original business logic shows the differences between the two. Specifically as suspected on-peak only calls give the same results but in the case of off-peak calls, usage of boundary values shows a small anomaly with the older implementation. Specifically if a call ends on 7:00 (am) exactly then the whole call is charged peak rate. Semantically we believe this should not have been the case even with the original version of the system because one could argue that there is no actual “overlap” (i.e. the precision in which the legacy software calculated if it was overlapping peak time was too coarse).

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Fig 3***. New acceptance tests against the older business logic clearly show what the new feature should implement. In the first case the results match (both calls exclusively during peak time), in the second case as expected they don’t (overlap with peak time). Finally the third report shows that Alice was charged peak rate for the whole call even though technically the whole call was during off-peak*

It is important to note that the Fit documentss assumed a 24hour time format of the en-UK Locale therefore they will pass only on a system set to use this Locale. The Fit documents along with the Fit reports are included in the /doc/ directory.

**2.2 Implementation**

Before implementing the new feature we configured Hudson to also run and generate reports for the FIT documents and also report on the code coverage we had from the tests (using Cobertura). This helped the team to have constant information about the state of the latest build. Since the acceptance tests were in place and they were failing for the right reasons (new logic not yet implemented), we decided to tackle the development of the new feature. This was done using the TDD methodology. Due to the refactoring work done previously, the rate logic was injectable from a higher layer and it was described by a concrete implementation of the *RateSelector* interface.

|  |
| --- |
| **int** secsOffpeak = **this**.getSecondsOffpeak(call);  **int** secsOnpeak = **this**.getSecondsOnpeak(call);    rates.add(**new** RateDescriptor(tariff.offPeakRate(), secsOffpeak));  rates.add(**new** RateDescriptor(tariff.peakRate(), secsOnpeak));    **return** rates; |

In general, the idea is to split the call into off-peak and peak parts (e.g. duration of the call that was during peak, and duration that was offpeak) with the related rates. In order to calculate the call cost the *BillingSystem* will iterate over the (rate,duration) set (each tuple contained in *RateDescriptor*). *OffpeakRateSelector*, which implements the current logic will return a set containing only one (rate,duration) tuple for the whole duration. The rate will be peak if the call overlaps peak time. *OffpeakFairRateSelector ,* implementing the new logic, will do this splitting noted above and return two *RateDescriptors* if applicable.

|  |  |
| --- | --- |
| D:\Imperial\AUTUMN TERM\ADV TOPICS SOFT ENG\COURSEWORK 2\AcmeTelecom\trunk\doc\structure-diagram.tiff |  |

**Fig 4***. (left) Final structure diagram as produced by Structure101. There are no violating dependencies in the production code. (right) Simplified final class diagram. The detailed version is included in the /doc/ directory.*

Due to the small size of the change, we only needed one iteration of the acceptance test scope TDD loop.

**3 Metrics**

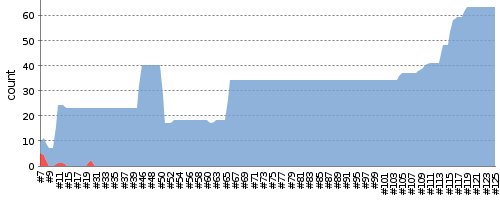
In the following paragraphs a number of metrics are presented for the “released” version. The released version is: **AcmeTelecom v2.0b , Rev 111 , Build 125.**

**3.1 Code size**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Unit tests | Fixtures | Fakes/Mocks | Production | Total |
| Lines of code (exc. comments) | 1189 | 199 | 129 | 701 | **2218** |

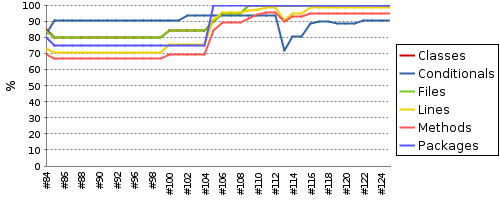
**3.2 Test related**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Unit tests | Acceptance tests (cells tested) | Total |
| Number of tests | 62 | 134 | **196** |



**Fig 6***. Test result trend.*

The test result trend as reported by Hudson is shown above. The weird peak and the later reduction of the total number of tests were due to a configuration error in the build file.



**Fig 7***. Code coverage (by the tests) as reported by Cobertura*

*.*

Cobertura reports a satisfying percentage of code coverage by our tests.

**4 Future features**

Since our release has high test coverage, a detailed class diagram available, executable documentation (fit-docs) and finally no cyclic dependencies, we believe any team could find its way around this codebase. Apart from the above, the classes are mostly decoupled from specific implementations therefore it would be easy to:

1. Add a different *Printer*, or print to a database etc.
2. Use different databases apart from the ones included in the external.jar. *BillingSystem* is just an implementation of the Biller interface that uses the databases provided whilst *ConfigurableBillingSystem* (that implements all the logic) is decoupled from the databases used.
3. *OffPeakFairRateSelector* returns only two rates for this version but another implementation of *RateSelector* could use three (e.g. offpeak,peak and weekend) or more .In general the *RateSelector* interface is designed in a way it allows for e.g. promotional rates based on the Calendar date/time or based on the caller/callee (e.g. Friends and family). An implementation of *RateSelector* could implement the business logic in case the charging policy changes again. The information needed to implement these is passed through the constructor.

**5 Miscellaneous techniques**

**5.1 Unit testing**

Most unit tests were written taking into account readability and ease-of-understanding of the actual code. In order to achieve this, we used the Builder pattern extensively, applied method chaining and micro-types for value types (e.g.: *Hours*, *Minutes*, etc). We implemented fluent interfaces to describe domain entities such as calls and customers.

|  |
| --- |
| **final** Customer c1 = *aCustomer*().  named("John Smith").  withPhoneNumber("447766406373").build();  Call call = *aCall*().  from(customer1).  to(customer2).  startingAt(  buildStartTime(*Hours*(15),*Minutes*(30),*Seconds*(0))).  endingAt(  buildStartTime(*Hours*(15),*Minutes*(45),*Seconds*(0))).  build(); |
|  |

In some cases, we used our own subclasses of Hamcrest matchers to make assertions easier to understand for less technical people.

|  |
| --- |
| *assertThat*( rates,  *anyRateDescriptor*(*equalsTo*(  *aRateDescriptor*().  charging(Tariff.*Standard*.peakRate()).  withDuration(*Minutes*(15)).build()))); |

We also made use of mock objects to test collaboration between the main components of the system:

|  |
| --- |
| **final** List<Customer> customers = **new** LinkedList<Customer>();  customers.add(c1);  context.checking(**new** Expectations() {{  allowing(customerDatabase).getCustomers();  will(*returnValue*(customers));  oneOf(generator).send(  with(*equalTo*(c1)),  with(*iterableWithSize*(0)),  with(*anyString*()));  }}); |

**5.2 CI Server build configurations**

Hudson was configured to build four times a day regardless of changes in the repository. On top of that Hudson built the project if there were any changes in the repository in the last 5 minutes (queried the repository every 5 minutes). In case of a failed built or a stable built after failing some builds Hudson emailed all the group members.

**5.3 Mocking the final Tariff enum**

In order to mock the *Tariff* enum (exclusively for the tests) , we used a legacy jMock library that can be used to mock concrete classes. On top of this jDave-unfinalizer had to be used as a javaagent for the jmv. Even though a bit contrived we were able to make a *FakeTariffLibrary* that returns rates as defined by the Fit documents.