TESTING

Refactoring Java, Part 2: Stabilizing your legacy code and technical debt

Pin-down tests are the key to stabilizing your legacy Java code.

by Mohamed Taman

October 23, 2020

Refactoring is your code improvement process, and the goal is to improve quality and encourage the modification of your software product. Refactoring makes the code simpler: You have fewer code lines than you began with. Fewer lines of code means fewer potential bugs.

To ensure that refactoring is taking place to enhance your code and make it simpler, you need to have rigorous testing, which is the refactoring foundation. Without well-defined test cases that verify your code behavior's correctness, you cannot be sure that you have not altered the code's external action. The refactoring process is about changing the internal structure of your code without affecting its external behavior.

The first article in this series, "Refactoring Java, Part 1: Driving agile development with test-driven development," introduced the Roman Numerals Kata, and I alternated between writing new code and then refactoring that code.

This second article and the third (final) article will focus on refactoring debt-ridden technical legacy code to improve its agility. You will practice refactoring to remove technical debt for existing legacy code that simulates a real-world scenario. To do that, you will use the Gilded Rose Refactoring Kata, which was written and posted to GitHub by Emily Bache.

In her blog post about the Gilded Rose kata, Bache describes it as follows:

> *"The basic plot of the Kata is that you've just been hired to look after an existing system, and the customer wants a new feature. Having a look at the*

> *code, you can see you're going to want to refactor it a little before adding the new feature, and before you do that, you're going to want some automated tests."*

Following that kata, let's start with legacy code that is full of technical debt, and then refactor it to the point that it will be trivial to add a new feature to the code. During that process, you'll pay off the technical debt and regain agility.

In this article, your first task is to stabilize the legacy code by writing pin-down tests that ensure you understand the legacy code. This helps ensure that future refactoring will not alter the code's external behavior.

First, though, what's the problem with technical debt?

Why is technical debt a problem?

Technical debt is code with problems that can be improved with refactoring. The technical debt metaphor is that it's like monetary debt. When you borrow money to purchase something, you must pay back more money than you borrowed; that is, you pay back the original sum and interest.

When someone writes low-quality code or writes code without first writing automated tests, the organization incurs technical debt, and someone has to pay interest, at some point, for the debt that's due.

The organization's interest payments aren't necessarily in money. The biggest cost is the loss of technical agility, since you can't update or otherwise change the behavior of the software as quickly as needed. And less technical agility means the organization has less business agility: The organization can't meet stakeholders' needs at the desired speed.

Therefore, the goal is to refactor debt-ridden code. You're taking the time to fix the code to improve technical and business agility.

Now let's start playing with the Gilded Rose kata's code and see how to stabilize the code, while preparing to add functionality quickly in an agile way.

Stabilizing that messy legacy code

One huge main problem with legacy code is that someone else wrote it. Many times, legacy code is a mess and poorly documented, and you simply don't understand it. You can't safely add additional features before making sure you understand the code; you have to be confident enough to make those changes without breaking anything. Let's see how to gain that confidence.

First, set up the environment. The solution code for this kata is at my GitHub repository. You can clone it as follows:

```
~$ git clone https://github.com/mohamed-taman/Agile-Software-Dev-Refactoring.git
```

The solution for this article is under the Gilded Rose module. You have two options to work with the Java code for this article:

- If you would like to follow along with me, please do. Simply follow the article's steps.
- But if you would like to navigate the code, this article is divided into steps, and each step has a git commit for each test-driven development (TDD) ***red-green-refactor*** change. When you navigate code commits, you can notice the differences between each step and the refactoring changes toward the final kata requirements.

All required software is listed in the first article, "Refactoring Java, Part 1: Driving agile development with test-driven development." I am using IntelliJ IDEA, as I did in the first article, but of course you don't need to use that particular IDE. The step names begin with "A2" to indicate that this is the second article in the series.

Step A2-1: Set up the kata. To remove technical debt using the Gilded Rose kata, load the kata's code as follows:

1. Go to the Gilded Rose GitHub page and either clone the repository or download the zip file. (I downloaded the zip file as shown in **Figure 1**.) Then, in your file browser, unzip the file.

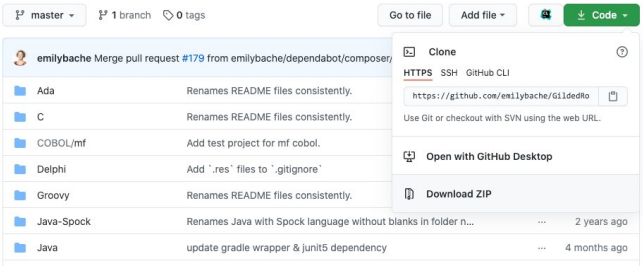

Figure 1. Downloading the Gilded Rose kata

2. Second, fire up IntelliJ IDEA, and at the welcome screen, click the `new project` link. On the next panel, on the left side, select `Maven`, and click `next`. The project name is `Gilded Rose`. Open `advanced` and fill in the following: The `artifact Id` is `gilded-rose`, while the `group Id` is `com.siriusxi.javamag.kata`. Click `finish`. In the bottom right corner of IntelliJ's screen, click `enable auto-import`.

3. In the IntelliJ project browser, open the `Gilded Rose` project, open the `src` folder, and open the `main` and `test` folders, as shown in **Figure 2**.

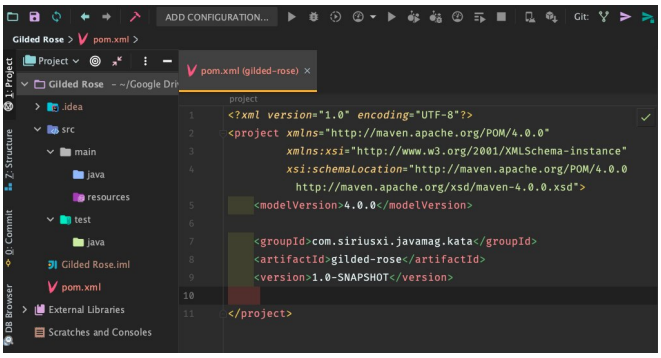

Figure 2. The Gilded Rose kata in IntelliJ IDEA

4. Navigate back to your file browser and change the view to `column view`. In the `Gilded Rose` folder that you unzipped, look for the `java` subfolder.

    a. Click the src `folder`, then click the `main` folder, and then click the `java` folder. Drag and drop `com` to the `src > main > java` folder in IntelliJ. Click `okay` to confirm that's what you want.

    b. Go back to the `src` folder, and click `test` and `java`. Drag and drop `com` to the `src -> test -> java` folder in IntelliJ. Click `okay` to confirm.

5. In IntelliJ's project browser, open the `com.gildedrose` package and double-click to open the `GildedRoseTest.java` file. Notice all the red; that's because fresh out of its GitHub repository, Gilded Rose isn't set up for IntelliJ. Let's fix that now.

    a. On line 9, put the cursor over `@Test`, and press `Option+Enter`. Select `add JUnit5.4 to the classpath`. All the red should be gone.

    b. There is a `foo()` method. That's a JUnit task, so let's run the test. Right-click the `GildedRoseTest` source file and select `run 'GildedRosetest'` from the menu. There's red; the `GildedRoseTest` has a test that failed.

    c. This red failed test is good; it is the first step of the red, green, refactor TDD three-step dance. This test fails because the expected value is `fixme`, while the actual returned value is `foo`. This test is broken on purpose to make sure it's red so you can see if everything is set up properly.

    d. Change the string at line 14 to `foo`. Rerun the test; it passes, and everything is now green.

Do a small refactoring. You've seen red and you've seen green. Now, let's improve the test code by renaming the `foo()` test method to a name that better describes what this test does. This test proves that the test framework is up and running. Rename the test code to `junitFrameworkWorks()`, and rerun the test to make sure nothing broke. Everything should still be all

green, as shown in **Figure 3**. Then, you are ready to move on to the rest of the kata.

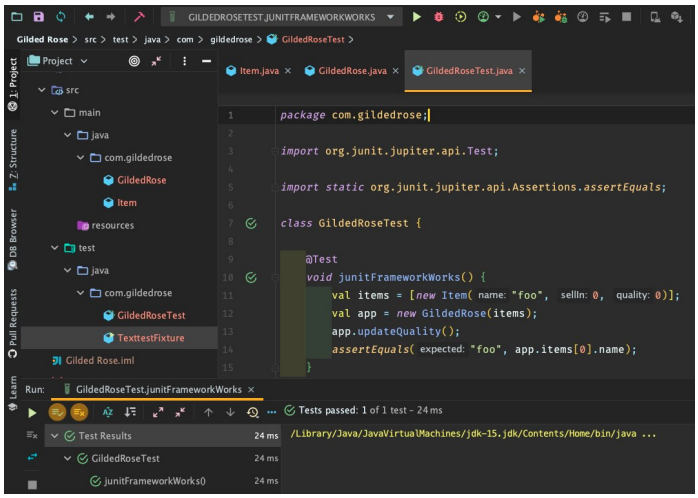

Figure 3. Green Gilded Rose tests after the first simple refactoring

Let's find the requirements for the Gilded Rose and integrate them into the project to make the documentation more accessible and convenient.

Step A2-2: Add the requirements to the project

To add the requirements to the project for ease of access, do the following:

1. Go back to your file browser and look for a file called `GildedRoseRequirements.txt`. Drag and drop it to `Gilded Rose` in your project browser.
2. Click `okay` to confirm.
3. Open that file to read it.

Wow—those requirements are confusing! Some items are difficult to understand, and some appear to contradict each other. This is intentional, because the Gilded Rose kata is designed to be a realistic simulation of a real-world coding problem.

As you scroll down, remember that your goal is to add a single new feature to the Gilded Rose. Let's look at the code itself. Under `src -> main -> java`, open the `com.gildedrose` package, and look at the `GildedRose.java` file.

Read the source code, and notice the following:

- The method called `updateQuality()` has many lines of code—more code than can fit on one screen. It's a very long method. So, there's a code smell right there.
- There are many `if`s, `else`s, `not`s, and `and`s. There's a lot of program logic that looks confusing and is hard to read.
- There are some repeated hardcoded numbers such as 50. There's a lot of confusing business logic, which implies

> technical debt, because the application may be hard to update.

Given the requirements document, which was confusing enough, you might have thought it would be easy to add the new feature, but now I suspect you are not so sure. This type of situation is common in the real world when you work with legacy source code. Someone else wrote the code, it's a mess, and you don't understand it. Again, this is where this kata is a realistic simulation of a real-world coding problem.

The test-refactor-add methodology for working with legacy code. To be able to add a new feature to this legacy source code, here's the methodology you'll use:

1. **Add pin-down tests.** Add tests: lots of tests. Those tests will serve two functions. They will help you understand the legacy code's external behavior, and they will also pin down the code's behavior so that if something breaks, you'll know it.
2. **Refactor.** Improve the legacy code by simplifying it. The pin-down tests will ensure that you don't break anything along the way.
3. **Add the new feature.** Given the simplified code, it will be much easier to add the new feature. You'll create that new feature using TDD.

Before you can add the first pin-down test, examine the test code that the kata author provided. Open the file called `GildedRoseTest` and notice how it works:

- Line 11 creates an array of items.
- Line 12 creates an instance of the `GildedRose` application passing in that array of items.
- Line 13 calls `updateQuality()`.
- Line 14 asserts the current state of the items in the array.

You will follow the same pattern as you write the system pin-down tests.

Step A2-3: Add the first pin-down tests

Go back to the requirements file again and look at the requirements. The first requirement says this:

> *"All items have a `SellIn` value, which denotes the number of days we sell the item. All items have a `Quality` value that indicates how valuable the item is, and at the end of each day, our system lowers both values for every item."*

Write a pin-down test for that behavior. First, go back to the `GildedRoseTest` class; at line 16 press `Enter` and add the `systemLowersValues()` method, as follows:

```
@Test
    void systemLowersValues() {
        Item[] items = new Item[] { new Item(
        GildedRose app = new GildedRose(items
        app.updateQuality();
        assertEquals(14, app.items[0].sellIn)
        assertEquals(24, app.items[0].quality
    }
```

The method `systemLowersValues()` tests `sellIn` and `Quality` as they decrease in value, and this is why it's called `systemLowersValues`. I'll write the rest of the test following the same pattern of the first method provided with the test class.

I have created an array of items, and I will assign some values greater than zero. `SellIn` is equal to 15, and `Quality` is 25, so you can distinguish the two.

I have created an instance of the Gilded Rose application with those items. I called the `updateQuality()` method, and I made a couple of assertions based on what I think the behavior should be:

- The first assertion is about the `SellIn` value. I think `SellIn` will go down by one; let's see if that's what happens.
- The other assertion is about the `Quality` value. I expect the `Quality` value to decrease by one. Let's find out if it does.

Rerun the tests to see what happens. As shown in **Figure 4**, the tests are all green. This proves something about the behavior of the system: When I pass an item with a `sellIn` value of 15 and a `Quality` value of 25, and I call `updateQuality()` as in line 21, the `SellIn` value decreases by one and the `Quality` value decreases by one.

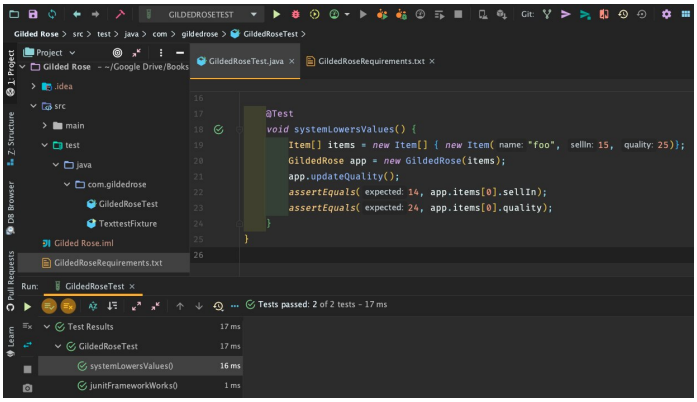

Figure 4. Running the first pin-down test

Step A2-4: Add more pin-down tests

Pin down a second behavior. Let's look at the requirements and add another pin-down test. At line 18, the document says this:

> *"Once the sell-by date has passed,* `Quality` *degrades twice as fast."*

Add a pin-down test to confirm that behavior. At line 24, press `Enter` twice and add a method called `qualityDegradesTwiceAsFast()`. This time, just copy and paste from the previous tests, so you don't have to type all the code in by hand.

The requirement says that if the item can no longer be sold, which is the same as saying `SellIn` is zero, then `Quality` degrades twice as fast. So, you want something with a `SellIn` value of zero and a `Quality` value of, say, 17, which is greater than zero. To confirm the expected behavior, you'd expect `Quality` to decrease from 17 to 15, indicating it degraded twice as fast as before. Here is the final method:

```
@Test
    void qualityDegradesTwiceAsFast() {
        Item[] items = new Item[] { new Item(
        GildedRose app = new GildedRose(items
        app.updateQuality();
        assertEquals(15, app.items[0].quality
    }
```

Run the test. It's green. You have successfully pinned downed another behavior.

Pin down a third behavior. The next item in the requirements document, at line 19, says the following:

> The quality of an item is never negative.

Let's add a pin-down test for that. At line 32, press `Enter` twice and add a method called `qualityIsNeverNegative()` by copying and pasting. Start with a `Quality` of zero, and see if it goes negative (which is wrong) or if it stays at zero (which is desired). Here's the method:

```
@Test
    void qualityIsNeverNegative() {
        Item[] items = new Item[] { new Item(
        GildedRose app = new GildedRose(items
        app.updateQuality();
        assertEquals(0, app.items[0].quality)
    }
```

Run the test to see if the behavior is correct. Everything is green, so that behavior has been pinned down as well. And now you've gotten the hang of it, right? Let's tackle a different problem, that of technical debt.

Step A2-5: Refactor the pin-down tests

Previously, when you added the new pin-down tests, you copied and pasted the code. That's a clue that these tests injected new technical debt, because of the DRY principle: "Don't repeat yourself." The last test method has lines like these:

```
Item[] items = new Item[] {new Item("foo", 0,
GildedRose app = new GildedRose(items);
app.updateQuality();
```

Those lines are repeated over and over in the tests, along with phrases such as `app.items[0].something`. That's cumbersome, prone to errors, and difficult to maintain. So, let's simplify as follows. Scroll up to the top of the file. Inside the method `junitFrameworkWorks()`, at line 10, press `Enter`.

Design the refactoring before writing the code for it. Look for a method that returns an item; call this `createAndUpdate()`, as shown below. It will take two inputs: a `sellIn` and a `Quality`:

```
...
Item item = createAndUpdate(0, 0);
...
```

Of course, that method doesn't exist yet, so it shows up in red in the editor. Have IntelliJ create the method. It's going to return an item: The first input is an int, to be renamed as `sellIn`, and the second input is also an int, and the name will be `Quality`. Finally, cut the repeated lines 12 through 14, and paste this repeated code into the new method so it looks like this:

```
private Item createAndUpdate(int sellIn, int
    Item[] items = new Item[] {new Item("foo"
    GildedRose app = new GildedRose(items);
    app.updateQuality();
    return app.items[0];
}
```

Notice line 13: There's some red. That's a pre-execution compiler error, so change `app.items[0]` to `item.name`. Run the tests and make sure nothing broke. Everything is still green.

There's one little thing to adjust: At line 12, there's a blank line. Delete that blank line and rerun the tests. It might seem unnecessary to rerun tests after merely deleting a blank line—but the real point is to practice the movements repeatedly so when you get to real code, it'll be a habit to always rerun tests whenever the code changes.

Now, another thing to refactor here is the location of this new private method. Its current location is between two of the tests; move it to the top of the file, so all the tests are grouped. Do this by cutting the method and pasting it at line 9. Rerun the tests to make sure nothing broke. Everything is still green, so you can continue.

Now, make these same changes to the rest of the test methods. After *each* method, rerun the test, and if it's green, move to the next. At the end of the process, all the tests should be green, and the test code will be much easier to read and maintain.

Create a pin-down test for the next requirement. Look back at the requirements document to find more pin-down tests to create. The next is at line 20, and it says this:

> *"Aged Brie increases in quality the older it gets."*

Add a test for that behavior, and follow the pattern from previous pin-down tests, for example:

```
@Test
    void agedBrieIncreasesInQuality() {
        Item item = createAndUpdate(15, 25);
        assertEquals(26, item.quality);
    }
```

Run the test and see if it's green. Uh oh: It's red. So open it, and inspect the test. It looks like `Quality` even went down by one, as shown in **Figure 5**.

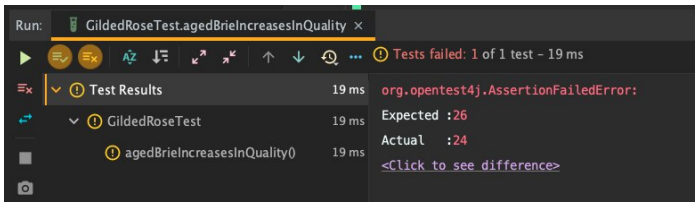

Figure 5. The test for aged Brie failed.

That's not the expected behavior, and you might know what went wrong. You created a regular item instead of an item of type `Aged Brie`.

Step A2-6: Refactor by changing the method signature

Go back to the method `createAndUpdate()` and modify it to allow you to change different products. As you can see, currently it creates an item with the name `foo` instead of `Aged Brie`. Fix this by doing a small refactoring:

1. Right-click the `createAndUpdate()` method. Select `Refactor` and `Change Signature`.

2. Click `plus` and notice that the `type` of the new parameter is a `String` and the name of the variable is `name`, which is the item name. The default value will be `foo`, and in this way, IntelliJ will update all existing code with the new parameter. Press `Enter` there.

3. You want this to be the first parameter in the method, so move it up to the top and click `Refactor`. Then it is refactored for you.

The IDE added the new parameter in the method signature, and then it updated the rest of the code. This is one of the reasons I like to let IntelliJ do the refactorings; for many of the refactorings, IntelliJ automates the process and updates all the code to ensure that the updated code works correctly.

Notice something else at line 10: The code is still using `foo` as the name of the item. Instead of the hardcoded string `foo`, the item should be the input parameter `name`. Go back down to the new test for aged Brie. Change the `name` of the item type to `"Aged Brie"`, run the test, and see if it's green. Yes; it is! This pins down another required behavior of the legacy system: When an item has the name `"Aged Brie"`, its quality goes up every day.

This step introduced a new refactoring type called a *change signature*. Use it to change a method by adding or removing its input parameters or its return type.

Step A2-7: Create pin-down tests for the remaining requirements

Based on the requirements document, add additional pin-down tests, which you can see on GitHub. Do the same before continuing so that you can practice more.

Discover more behaviors with code coverage tools. Although you have created many pin-down tests to cover all the documented requirements, perhaps you are not sure if all the tests cover all the requirements. Maybe there's something else that needs to be protected that you don't know about.

To identify any missing requirements, you can use code coverage tools to pin down the external behavior of untouched lines of code and construct new tests to reach them. To do this, run the Gilded Rose test using IntelliJ's coverage tools; you should find comparable features in other development environments.

1. In the right corner of the IntelliJ window, click the icon that looks like **Figure 6**, which is for `Run with code coverage`. A code coverage summary panel pops up, showing the tests' code coverage results.

2. Double-click the `com.gildedrose` package, and see the results for the two production code classes. The

`GildedRose` class has 93% code coverage, and `item` has 83% code coverage, as shown in **Figure 7**.

Figure 6. The code-coverage icon.

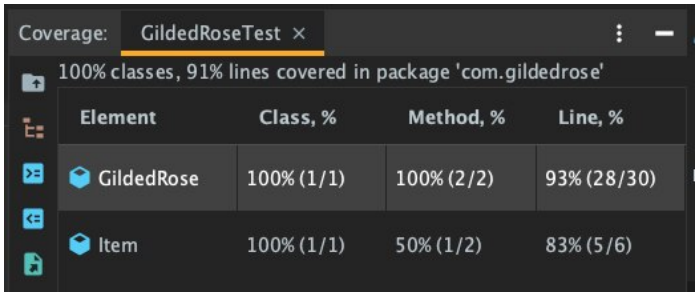

Figure 7. Code coverage panel

You haven't looked at the `Item.java` class yet, so do that now. In the source code window's left side, notice the green marks from lines 11–15. Those indicate that the tests have touched these lines of code. But at line 19, you can see a red marker, as shown in **Figure 8**. That indicates the tests have not touched this line of code.

Figure 8. Class code coverage information showing that line 19 has not been touched by the tests

Line 19 is a `toString()` method, which is overriding the default method. Let's make a judgment call here that it's not necessary to write a test that executes this code line.

Going back to the `GildedRose.java` source code, the good news is that there are many green markings on the left side, which means the tests have touched most of the code. It looks like there are only two lines of code, 55 and 56, that have not been touched yet, as shown in **Figure 9**.

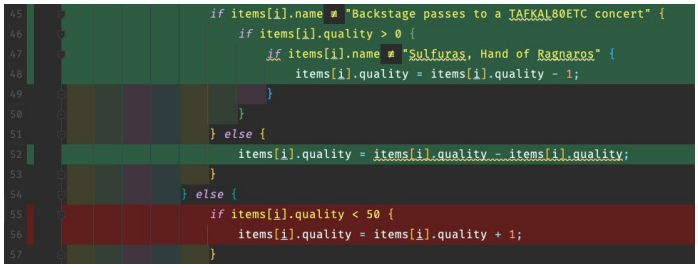

Figure 9. GildedRose class code coverage showing lines 55 and 56 have not been touched by the tests

Close the coverage summary panel and look at these lines of code. How do you create a test that touches these code lines? Look around, and notice the embracing `if` statement at line 43. You can imply the following unwritten requirement: When the code encounters an item with a `sellIn` value less than zero and with the item type `Aged Brie`, the quality will go up by one.

Step A2-8: Pin down a test based on what was discovered by code coverage

It's time to write a pin-down test that reaches lines 55 and 56. Go back to the `GildedRoseTest` class and add the new test, as follows:

```
@Test
    void agedBrieNeverExpires() {
        Item item = createAndUpdate("Aged Bri
        assertEquals(0, item.sellIn);
        assertEquals(43, item.quality);
    }
```

This creates a new item with the name `"Aged Brie"` that has a `sellIn` of 0 (which should get you into that code at lines 55 and 56) and a `Quality` of 42. You'd expect the `sellIn` value to stay 0 and the `Quality` to increase because the quality of aged Brie always increases.

Run the test to see what happens. Oh no; the test is red. Why? It says it expected the `Quality` to be 0, and the actual value was -1. It's apparent now that the actual behavior is that when the `sellIn` value is 0 for `Aged Brie`, `sellIn` still is decreased by 1. At line 90, put a -1 instead of 0.

```
assertEquals(-1, item.sellIn);
```

By the way, this is different from TDD for a new feature, because the goal here is to understand the legacy code, and that means determining what it is currently doing—even if that is different from what is in the requirements document or different from what you think it should do.

Therefore, this test is technically red. Instead of fixing the legacy code to change the potentially erroneous behavior of sold-out aged Brie, write a new pin-down test that verifies that current behavior, so you can see whether that behavior changes in the future.

Rerun the tests. Oops; it's still red. Why? The expected result is 43, and the actual result is 44. The `Quality` increases by two instead of by one. Pin down that behavior at line 91 and rerun the test.

```
assertEquals(44, item.quality);
```

Everything is green again, so this test pins down that behavior for aged Brie. Rerun again with code coverage to see if all the lines of code are covered: Go back to the `GildedRose.java` source code file. The tests cover lines 55 and 56, and they are now green, as shown in **Figure 10**.

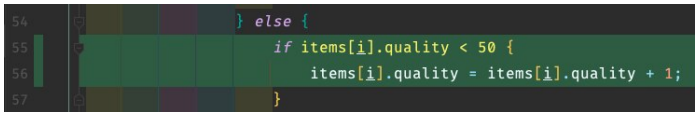

Figure 10. All the code in GildedRose is now covered by tests.

Another technique is branch coverage. You are almost done writing pin-down tests that stabilize the legacy code. You have used the code coverage tool to ensure the tests cover 100% of the legacy code lines. But what about branches?

Look at all the `if`/`else` statements in `GildedRose.java`. What do you think about them? Let me give you my opinion: They are all marked in green according to code coverage, which is good, so tests have hit these lines of code, but have they done so sufficiently? Maybe. Maybe not.

Consider the `if`s that could evaluate to either `true` or `false`. The pin-down tests may not have evaluated both conditions, and as such, it's unclear whether you have hit each of the `if` lines for both the `true` and the `false` cases. For this reason, the next task that you should do is to run tests with the branch coverage tool.

To enable branch coverage in IntelliJ, do the following:

1. Select `Run/debug configuration` and then `Edit Configurations`, as shown in **Figure 11**.

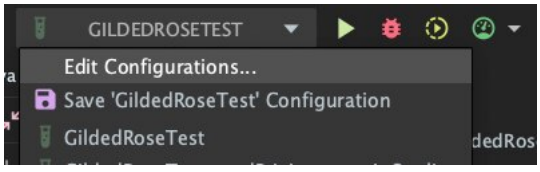

Figure 11. Editing the run/debug configurations in IntelliJ to enable branch coverage

2. On the right side, select the `Code Coverage` tab and then the `Tracing` option, and then click `okay`.

To verify the correct behavior, click `Run with Coverage` again. Back in `GildedRose.java`, did you notice something different? You should see that some of the green lines are now marked dark green (in my case, but it might be a different color based on your theme). Dark green means that you have executed this line of code, but you have hit only one of the `true` or `false` sides of the branch, as shown in **Figure 12**.

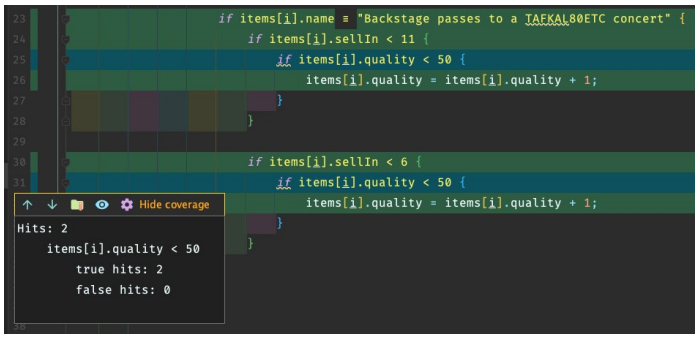

Figure 12. GildedRose branch coverage

Clearly, there's some work to do. The next task is to add pin-down tests that hit both sides of each of the `if` statements to ensure that both the `true` and `false` paths are verified.

Step A2-9: Pin down branch coverage

Start with line 25 and think about how to reach that line of code. Go up to line 12 and think about what it means. I think it means the following: If the item's name is "Aged Brie" or "Backstage pass" and the `Quality` is less than 50, the quality should increase. If it is a backstage pass, what you care about is whether Backstage passes and whether the `sellIn` value is less than 11, so if it is close to the date of the concert and the item's quality is less than 50. Increase the item's `Quality` again, so the `Quality` will increase by two in that case.

What would it take to reach line 25? Go back to `GildedRoseTest.java`, and try to construct a test. Here's an idea: At the end of line 92, press `Enter` twice and add this new test case:

```
@Test
void backstagePassMaximumQuality() {
  Item item = createAndUpdate("Backstage pass
  assertEquals(50, item.quality);
}
```

This test case is about backstage passes and their maximum quality. In the requirements document, it says the maximum quality is 50. In the code above, you created an item called `Backstage pass` with a `sellIn` value of 10 (which is less than 11) and a `Quality` of 48 (which is less than 50). The test should show that the quality has increased by two, going from 48 to 50. Run the test to see if I indeed understood the behavior correctly. The test is green, so I nailed it.

Wait; there's one more thing: To hit the other side of the `if`, both the `true` and the `false`, you need a backstage pass with a `Quality` of 49, and you want to a test that validates the quality does not increase beyond 50. Copy the two lines 96 and 97 and

paste them at line 99. Next, create another test that starts with a quality of 49, for example:

```
item = createAndUpdate("Backstage passes to a
assertEquals(50, item.quality);
```

Rerun the test. It looks like I understood the expected behavior. Now run the test with branch coverage to see if it hits both sides of the branch at line 25 in `GildedRose.java`. Yes; that line is now marked in green, as shown in **Figure 13**.

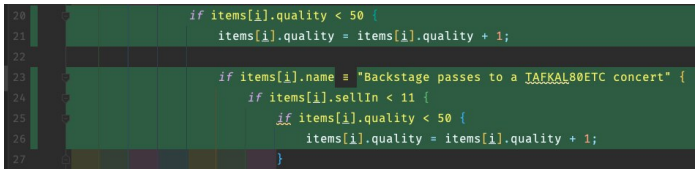

Figure 13. The test successfully pinned down both branches at line 25.

Your next step is to add the rest of the pin-down tests and achieve 100% branch coverage.

Step A2-10: Final pin-down tests for full branch coverage

If you get stuck, I have added the pin-down tests to reach 100% coverage in the code repository, as shown in **Figure 14**, so you can compare your trials with them.

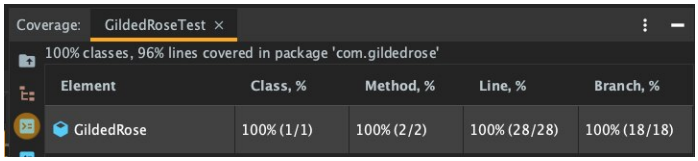

Figure 14. Full branch coverage achieved in the pin-down tests

Conclusion

You have discovered many steps and techniques for stabilizing legacy code that has outstanding technical debt—preparing you to handle inefficiencies and errors introduced due to sloppy and careless programming.

First, you learned how to create pin-down tests for legacy code to understand its behavior; you used many techniques to cover the requirements document. You also used the code coverage tool to check whether there was any code untouched by the test cases. And finally, you checked all the code `if`/`else` branches using a branch coverage tool to ensure 100% test coverage of the code.

Now, you should be confident enough to start refactoring the legacy code to make it run better. That's what will happen in part 3, the final article: You will use refactoring to simplify the legacy

code, remove duplication, and build more reusable objects. Finally, you'll see that refactoring complements an agile workflow by exploring how to add a new feature to the simplified legacy codebase.

Dig deeper

- Refactoring Java, Part 1: Driving agile development with test-driven development
- Test-driven development: Really, it's a design technique
- JUnit 5—A special issue of *Java Magazine*
- Interview with Kent Beck, the parent of JUnit and creator of TDD
- Unit testing your application with JUnit
- Simplified test-driven development with Oracle Visual Builder
- Gilded Rose Refactoring Kata by Emily Bache

Mohamed Taman

Mohamed Taman (@_tamanm) is the CEO of SiriusXI Innovations and a Chief Solutions Architect for Effortel Telecommunications. He is based in Belgrade, Serbia, and is a Java Champion, and Oracle Groundbreaker, a JCP member, and a member of the Adopt-a-Spec program for Jakarta EE and Adopt-a-JSR for OpenJDK.

Share this Page