TESTING

Refactoring Java, Part 3: Regaining business agility by simplifying legacy code

By making the code easier to understand, you also make it easier—and safer—to maintain.

by Mohamed Taman

December 4, 2020

The goal of refactoring is to improve code quality and encourage the modification of your software product. Refactoring makes code simpler. You have fewer code lines than you started with. Fewer lines of code mean fewer potential bugs.

In the first part of this three-part series, you saw how to drive agile coding with test-driven development (TDD).

In the second part, you refactored legacy code that was filled with technical debt, which made the code inefficient and hard to understand. Because old code is confusing, developers often aren't confident in their ability to alter a single line of code to add new functionality or to solve problems.

In that second article, you pinned down the legacy code to understand its behavior by creating tests with many techniques that cover the requirements document. You then used a code coverage tool to check whether there were any code lines that the test cases didn't touch. And finally, you checked all the `if`/`else` branches using branch coverage tools to reach 100% coverage of the code. Now that you understand that code, you can move to the next step: refactoring that code to make it simpler, make it run more efficiently, and make it easier to update to add or change functionality.

In this final article in the series, you will refactor to simplify the legacy code, remove duplication, and build more reusable objects. You'll also see how refactoring complements an agile workflow by exploring how to add new features to the simplified legacy codebase quickly.

You should read or reread the first two parts of this series to understand the Gilded Rose programming exercise, or kata, that you'll use here. The solution code of this kata is at my GitHub repository. You can clone it using this command:

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

The solution for this article is under the Gilded Rose folder. As before, you have two options:

- If you would like to follow along with me, download v2.0 of the repository, not the latest master version, to follow the article's steps.
- In case you would like to navigate the code, this article is divided into steps, and each step has a git commit for each TDD red-green-refactor change. When you navigate code commits, notice the differences between each step and the refactoring changes toward the kata's requirements.

All required software is listed in the first article. I am using the IntelliJ IDEA IDE on macOS, as I did in the first two articles. You should be able to follow along using other Java IDEs, though the screens will not look the same. The step names begin with "A3" to indicate that this is the third article in the series.

Simplify the legacy code by extracting content

Since you pinned down the legacy code in the previous article, you can safely simplify the legacy code by refactoring it.

A good place to begin is by extracting content. In the IntelliJ IDEA project browser, under `src > main > java`, select package `com.gildedrose`, and double-click the `GildedRose.java` class to open it. Looking at that legacy code, I notice several problems right away:

- First, this method is really long. It doesn't fit on this screen even if I change the font size. There's no way I could read the whole method at once.
- Second, there's a lot of complex logic that I want to simplify, such as lines 12, 13, 14, and 15. This logic is complicated, hard to read, hard to understand, and hard to change.
- There are many hardcoded strings and numbers, for example, in line 12, the string `Aged Brie`; at line 13, the string `Backstage passes`; at line 20, the hardcoded number 50; and at line 24, the hardcoded number 11.

Those all can be refactored, but this is too much to address all at once. Instead of doing one massive refactoring, you will do a series of microrefactorings to improve the code. Start with what might be the easiest refactoring, which is to extract hardcoded strings and numbers into constants.

Step A3-1: Refactor production code by extracting string constants. Do the following:

1. At line 12, right-click the string `Aged Brie`, and select `Refactor > Introduce Constant` or press `Option+Command+C`. Notice that IntelliJ suggests a proper name for this constant: `AGED_BRIE`.
2. Select the `Replace all occurrences` option, and press `Enter`.
3. Scroll up, and notice the new constant at line 4, which is shown in **Figure 1**.

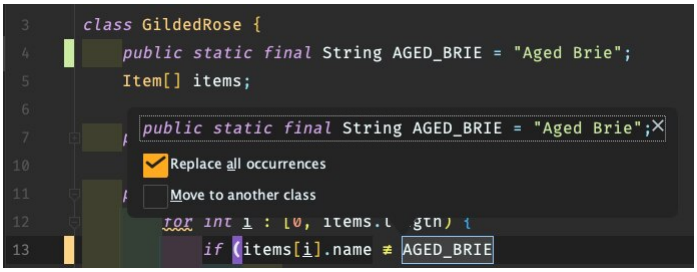

Figure 1. Introduce the string constant.

Look back down at line 13 and notice that the new constant is being used there. As you scroll through the rest of the file, in fact, you will see that every instance of the string `Aged Brie` has been replaced with this new constant. That's how tools can help you with refactoring.

Now, in the IntelliJ IDEA project browser, open the `src > test > java` folder and select the package `com.gildedrose`, and finally double-click `GildedRoseTest.java` to open it. Look at `GildedRoseTest.java` to see if the IDE made the constant changes there as well.

The change was not made: Smart as they are, tools don't know everything that you intend. So, the tool didn't replace the string `Aged Brie` with the new constant in the file `GildedRoseTest`. You have to do it manually, as follows:

1. Press `Command+R` (the keyboard shortcut to initiate the search-and-replace operation).
2. Type `"Aged Brie"` (with double quotes) to specify the text you want to search for; make sure `match case` is selected. Replace the text with `GildedRose.AGED_BRIE`.
3. Then click `Replace all` to replace all instances.
4. And finally, run the tests to make sure that you didn't change any legacy code behavior.

All the tests should be green, which means the pin-down tests helped you refactor with confidence that you didn't change any of the legacy code's behavior.

Step A3-2: Refactor production code by extracting number constants. The next easy task is to extract three of the hardcoded numbers (50, 11, and 6) into constants. I'm not sure that all the hardcoded numbers need to be refactored, but these three seem essential to the business logic of this application.

Start with the number 50 by performing the following steps:

1. Right-click `50`, select `Refactor > Introduce Constant` or press `Option+Command+C`. Give the constant a name. I think 50 represents the maximum quality, so `MAXIMUM_QUALITY` could be a good name to enter.
2. Select the `Replace all occurrences` option, and press `Enter`. And as expected, IntelliJ replaces all occurrences of the number 50 with the new constant `MAXIMUM_QUALITY`.
3. If you scroll up to the top of the file, in line 7 you will see the new constant's definition.

As before, the IDE's automatic changes were limited in scope. If you look at `GildedRoseTest`, you'll notice that IntelliJ did not automatically replace all instances of 50 with the new constant. You must do that yourself.

Press `Command+R` to initiate a search-and-replace operation. You are going to search for instances of 50 and replace them with `GildedRose.MAXIMUM_QUALITY`. Click `Replace All`.

Oops! Notice there's some red here. The search-and-replace operation broke something. The `qualityNeverMoreThanGildedRose.MAXIMUM_QUALITY()` method had the number 50 in the method name for this test; it was `qualityNeverMoreThan50()`. And it was changed to `qualityNeverMoreThanGildedRose.MAXIMUM_QUALITY`. So, adjust this by changing it to `qualityNeverMoreThanMaximum()`. The red should go away.

Rerun all the tests, and once again, everything should be green. This demonstrates that the pin-down tests are helping you to continue doing these microrefactorings to simplify the code.

I encourage you to repeat this for each of the hardcoded strings and numbers you see in `GildedRose.java`. I have already done that, and these refactorings are in the master branch, so you can compare my code to yours later.

What are the benefits of extracting the constants? You've increased your technical agility in two ways. First, it's easier for you to read the code. And second, if you want to change a string's value, you change it in only one line of code.

Simplify the legacy code by extracting variables and methods

Earlier, in looking over the code, you might have noticed a lot of duplication. One thing that's repeated many times is the phrase `items[i]`. It's in lines 18, 19, 20, 21, and so on. This duplication indicates a perfect candidate for refactoring by extracting a variable.

Step A3-3: Refactor production code by extracting variables. Here's how to make this change:

1. Highlight `items[i]`.
2. Right-click and select `Refactor > Introduce Variable` or press `Option+Command+V`.
3. Make sure you have selected the `Replace all 34 occurrences` option, so the IDE does the entire refactoring for you. IntelliJ suggests an excellent default name here: `item`.
4. Select the `Declare final` option and press `Enter` to accept that.

The IDE will refactor `items[i]` in lines 18, 19, 20, 21, and so on. Run the tests. Everything should pass: The pin-down tests continue to provide safety as you refactor the legacy code.

Step A3-4: Refactor production code by extracting methods. Other phrases repeated over and over include `item.name.equals(AGED_BRIE)`, `item.name.equals(BACKSTAGE_PASSES)`, and `item.name.equals(SULFURAS)`. These indicate that you might be able to simplify the code by extracting a method.

Look at the expression `item.name.equals(AGED_BRIE)` at line 19. I think the code is trying to express that this item is aged Brie. And at line 21, I also think the code is trying to say that the item is a backstage pass. Simplify the code and make the meaning more obvious, so the code will be easier to understand in the future.

Do the following to refactor `item.name.equals(AGED_BRIE)`, starting at line 19:

1. Highlight the entire `item.name.equals(AGED_BRIE)` expression, right-click, and select `Refactor > Extract Method`.
2. In the `Name` field, type a method name such as `isAgedBrie`, because that's the semantic intent of this expression.
3. Click `Refactor`. In my opinion, the original signature looks fine. That's the one on the left, where it says `isAgedBrie(Item item)`.
4. Click `Keep original signature`. The IDE then asks how many of these duplicate expressions you want to replace. Click `All`.

Scroll down to the bottom to see the newly created method at line 71; `isAgedBrie(Item item)` is the new method that the refactoring created. Run the tests to make sure you didn't break anything. Everything should be green.

Scroll back up to the top and look at lines such as 21 and 23 with expressions such as `item.name.equals(BACKSTAGE_PASSES)` and `item.name.equals(SULFURAS)`. I encourage you to repeat refactoring those in the same way.

Simplify the legacy code by refactoring arithmetic logic

Other kinds of expressions I see repeated in this code are simple bits of logic that increase or decrease a unit count. For example, at line 23, notice the line that says `item.quality = item.quality – 1`. Similarly at line 28, you can see `item.quality = item.quality + 1`. These expressions can be simplified.

Step A3-5: Refactor production code by simplifying arithmetic logic. Here is how to simplify these expressions:

1. At line 23, you can decrement the value of a variable by 1 by using `item.quality--`. After you do that, rerun the tests to make sure you didn't break anything. The pin-down tests are continuing to keep you safe.
2. Do the same thing at line 28; change it from `item.quality = item.quality + 1` to the simpler `item.quality++`, and rerun the tests.
3. There's another one in line 33. Every time you change something, rerun the tests to ensure you didn't make an error.
4. Scroll down; there's another one at line 39. Change it to use `++` and rerun the tests.
5. Change line 47 to use `--` and rerun the tests.
6. Change line 55 to use `--` and rerun the tests.
7. Line 59 is different. It says `item.quality = item.quality - item.quality`. I believe that's a very complicated way to say `item.quality = 0`. Make that change and rerun the tests.
8. The final one I think is at line 63, and `item.quality++` is the refactoring for that one. Rerun the tests. Everything should be green; the pin-down tests are continuing to keep you safe.

You are at the end of the method and have successfully refactored all the overcomplicated arithmetic.

Simplify the legacy code by refactoring Boolean logic

Looking at the code again, I see opportunities to simplify some of the logical expressions. Consider the code at line 51 and line 52, where the code says `if not isAgedBrie()` or `if not isBackstagePasses()`. I call this inverted logic or backward logic.

What is the code trying to do? It appears that if an `item` is not a `BackstagePass`, do this; otherwise, if it is a `BackstagePass`, do that. You can simplify the code this way: If the `item` is a `BackstagePass`, do something; otherwise, do something else.

You will refactor by flipping the logic around to make the meaning clearer. This is best done manually.

Step A3-6: Refactor production code by inverting the backward logic. At lines 52 through 60, refactor the code as follows:

1. At line 52, delete the "not" (`!`) symbol. At the end of the line, press `Enter` twice, type a right curly brace (`}`), type `else`, and type a left curly brace (`{`). You have recaptured what to do if it's not a `BackstagePass`.
2. Now cut line 61 and paste it at line 53. Clean up by deleting the dangling `else` and its curly braces `{}` at line 60.
3. Rerun the tests and determine whether you have kept the external behavior of the code. Everything should be green, so the pin-down tests are continuing to help you.

At line 51, you could do the same thing; I will leave this as homework for you to do.

Now, go to lines 19 and 20. This is a more complex logical statement that says if the item isn't `AgedBrie` and it isn't a `BackstagePass`, the code should do something. In this case, you can apply De Morgan's Laws to simplify this overly complicated logical expression. De Morgan's Laws state the following:

- `Not (A and B)` is the same as `Not A or Not B`.
- `Not (A or B)` is the same as `Not A and Not B`.

This can be restated using this code-friendly notation:

```
!a && !b   is the same as  !(a || b)
!a || !b   is the same as  !(a && b)
```

Using De Morgan's Laws, instead of saying `if not isAgedBrie and not isBackstagePass` you can say `if it's not isAgedBrie or BackstagePass`. Rerun the tests to ensure you didn't change any of the externally observable behavior. All the tests should be green. The pin-down tests are continuing to keep you safe.

I encourage you to look for opportunities to simplify the logic expressions in the rest of this code.

Simplify the legacy code by clustering business logic

The business logic is very convoluted in the `GildedRose.java` file. For example, it is impossible to find the exact code logic that handles a normal item. The code for handling aged Brie is all over the place in the `updateQuality()` method. The same is true for backstage passes and `Sulfuras`. (`Sulfuras` is a legendary item that should never be sold and whose quality should never decrease.) The code is interspersed throughout the entire method. But what about other items?

What you want to try to do is cluster the business logic for each type of item in its place, so it's easy to see how you handle a normal item, aged Brie, and so on.

Step A3-7: Refactor production code by grouping related business logic. Start by writing a handler for normal items. Add a couple of blank lines around line 19 and line 20 and start writing the following code:

```
final Item item = items[i];
      if (isNormalItem(item)) {

      } else {
        if (isAgedBrie(item) || isBackstagePa
          if (item.quality < MAXIMUM_QUALITY)
            item.quality++;
      .......
```

Note: You coded a reference `isNormalItem(item)`, and of course the tests show red, because you haven't implemented that yet. So, if an item is a normal item, you are going to do the logic for normal item stuff; otherwise, do the special items stuff. Then scroll down and add the right curly bracket to complete the `else` at line 75.

Scroll back up again and implement the `isNormalItem(item)` method. The Gilded Rose specification says the following for a normal item:

- The `sellIn` always decreases by 1, and the `quality` always decreases by 1.
- The minimum value for `quality` is 0.
- If the item's `quality` is less than 0, set the item's quality to 0.

Here's how to implement the `isNormalItem(item)` method:

```
if (isNormalItem(item)) {
    item.sellIn--;
    item.quality--;
```

```
    if(item.quality < 0)
       item.quality = 0;
      }
```

The first reference to `NormalItem()` is still red. Here's how to fix that: Put your mouse pointer over the method, press `Option + Enter`, select `create method,` and accept all the defaults by pressing `Enter`. The code will return a Boolean value that states that the product is a normal item if it is not one of the special items, as in the following:

```
private boolean isNormalItem(Item item) {
    return !(isAgedBrie(item) || isBackstageP
}
```

It should be green now, which indicates this code will compile and run. Run the tests and see what you get for the result. Oh, no: One of the tests fails, as shown in **Figure 2**:

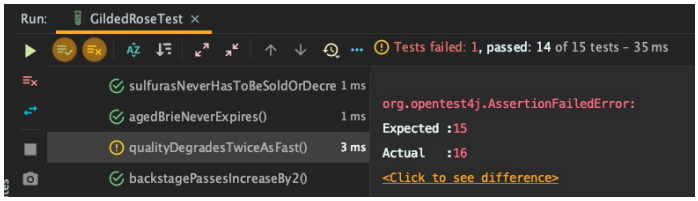

Figure 2. One of the tests failed.

What happened? The backstage pass degrades twice as fast as other products. The test expected to see a value of 15, but the actual value is 16.

Look at that test code; double-click the test named `qualityDegradesTwiceAsFast()`. This method asserts the following: If the `sellIn` is 0, `quality` decreases by 2. It looks like one of the business logic cases is missing. The good thing is that the pin-down tests help ensure you didn't break any existing external behavior.

Go back to `GildedRose.java`. How can you make this test green? I think what you are looking for is something like this:

```
if (isNormalItem(item)) {
          item.sellIn--;
          if (item.sellIn == 0) {
              item.quality -= 2;
          } else {
              item.quality--;
          }
          if (item.quality < 0)
              item.quality = 0;
} else {
.........
```

Run the tests and see whether the results are green. If they are, you have restored all the original external behavior.

Now you have an opportunity here to simplify the code even more. You could extract line 20 to line 27 into a single, separate method that encapsulates the business logic for the handling of a normal item. Try to do that by performing the following steps:

1. Right-click the highlighted code, and then select `Refactor > Extract > Method`.
2. Create a private method. Call it `handleNormalItem()`. It takes an `item` as its input. Click `Refactor`.
3. All that code has been compressed into a single line, at line 20, as shown below:

```
if (isNormalItem(item)) {
    handleNormalItem(item);
} else {
.......
```

Run the tests and make sure you haven't changed any of the external behavior. All tests should be green. The pin-down tests continue to provide safety to ensure that you haven't broken anything.

Step A3-8: Group the remaining related business logic. You should go through the logic and add handlers for each of the remaining item types. (Or copy the code from the master branch so you'll be at the same point as the next section.)

Final touches for the refactoring project

When you open `GildedRose.java` now, do you notice how much simpler the `updateQuality()` method is?

```
public void updateQuality() {
  for (int i = 0; i < items.length; i++) {
    final Item item = items[i];

    if (isNormalItem(item)) {
      handleNormalItem(item);
    } else if (isAgedBrie(item)) {
      handleAgedBrie(item);
    } else if (isBackstagePasses(item)) {
      handleBackstagePasses(item);
    } else if (isSulfuras(item)) {
      handleSulfuras(item);
    } else {
..........
```

The code is straightforward to read now. It's easy to see what happens if the application is processing a normal item, handling aged Brie, and so on. However, there are still a couple of

refactoring improvements you can make before adding new functionality.

Step A3-9: Refactor production code by cleaning superfluous code. All the code starting at line 28 in the final `else` clause appears to be unnecessary. The `if` and all `else if` statements from line 20 through line 27 handle everything for normal items and special items. Simplify the code by deleting all these code lines in the final `else` clause. And rerun the tests. The tests should all be green. You don't need that code!

Now look at the beauty of the new version of the `updateQuality()` method:

```
public void updateQuality() {
    for (int i = 0; i < items.length; i++) {
      final Item item = items[i];

      if (isNormalItem(item)) {
        handleNormalItem(item);
      } else if (isAgedBrie(item)) {
        handleAgedBrie(item);
      } else if (isBackstagePasses(item)) {
        handleBackstagePasses(item);
      } else if (isSulfuras(item)) {
        handleSulfuras(item);
      }
   }
 }
```

It looks perfect. It all fits on one screen, and it is very easy to read.

Step A3-10: Do a little more simplification. Are you satisfied? I think you can get rid of all the `if` and `else if` statements.

Instead of asking whether an item is a normal item and then handling it if it's a normal item, you can simply plan to handle it if it's a normal item. To do that, refactor `handleNormalItem()` to contain the `if isNormalItem()` test, as follows:

```
private void handleNormalItem(Item item) {
   if (isNormalItem(item)) {
       item.sellIn--;
...........
```

In the `updateQuality()` method, delete line 20, and then delete the `else` at line 22. Run the tests and make sure you didn't change any of the external behavior. All the tests should be green. The pin-down tests are continuing to help you as you simplify the code more and more.

For one last refactoring, change the name of the `handleNormalItem()` method to more accurately express

what it does. To do that, right-click the method, select `Refactor > Rename`, and change the name to `handleIfNormalItem()`.

Run the tests and make sure all are still green. Now the method's name accurately represents what it does.

I encourage you to repeat this series of microrefactorings on the remaining `if` and `else if` clauses in `updateQuality()` to simplify the code more.

Add a new feature to the refactored code

Look at the `updateQuality()` method now.

```
public void updateQuality() {
  for (final Item item : items) {
    handleIfNormalItem(item);
    handleIfAgedBrie(item);
    handleIfBackstagePasses(item);
    handleIfSulfuras(item);
  }
}
```

It's so simple and clear, and this will make it easy for you to add new behavior.

Before doing that, consider what you've done so far. You started with a large chunk of legacy code that was riddled with technical debt. In the second article, you added pin-down tests to stabilize the code. In this article, you refactored the code to simplify it.

Now, add new functionality to the legacy code.

Open the `GildedRoseRequirements.text` file. At line 29, the new feature request says the following:

> The Conjured items quality degrades twice as fast as normal items.

To add this feature to the refactored legacy code, you'll do the following red-green-refactor three-step dance of test-driven development, which was covered in the first part of this series:

1. First, start by writing a new test that is red.
2. Write minimal code until that test passes and until all the tests turn green.
3. Then improve the code, and refactor as needed.

Step A3-11: Add the new-feature test case. Open `GildedRoseTest.java` and scroll down to the end of the class to create a test case for the new feature that will handle `Conjured` items. Per the requirements, the method will look like this:

```
@Test
  void conjuredDegradeTwiceAsFast() {
      Item item = createAndUpdate(GildedRose.
      assertEquals(23, item.quality);
  }
```

To test the class, you have created an item with a `SellIn` value of 15 and a quality of 25. And you assert that the quality decreases by 2, so the item quality goes from 25 to 23.

The item named `GildedRose.CONJURED` is red in the IDE, and the code won't compile because `CONJURED` doesn't exist yet. To fix that, do the following:

1. Put the mouse pointer over `GildedRose.CONJURED`, press `Option + Enter`, click `Create Constant Field in GildedRose`, and then press `Enter`.
2. Press `Enter` again to accept the type being `String`, and that string will be `Conjured`. Press `Enter`.

You have written enough code that the code will compile and run, so run the new test. That test goes red, which is precisely what you'd expect because you haven't actually implemented any of the code's new behavior yet.

Look at the actual result shown in **Figure 3**: The item's quality decreased by 1 instead of 2. In other words, `Conjured` items are being treated as normal items.

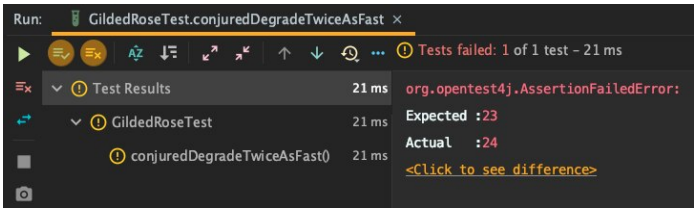

Figure 3. The Conjured item test failed because the new behavior has not yet been implemented.

Step A3-12: Implement the new feature's business logic. As in previous cases described in the earlier articles, you should write the least amount of code that makes the test turn green. Go back to the `updateQuality()` method, and continue following the same pattern you did for other items, adding a new method `handleIfConjuredItem(item)` at line 23.

Of course, it turns red because the method doesn't exist yet. Press `Option + Enter`, and create that method and its implementation as follows:

```
private void handleIfConjuredItem(Item item)
    if (isConjured(item)) {
      item.sellIn--;
      item.quality = item.quality - 2;
```

```
    }
}
```

In line 28, `isConjured(item)` is red. Press `Option + Enter`, and create method `isConjured`. It will return `true` if the item's name equals `CONJURED`. You should have implemented enough that the code will compile and run, so run the test.

This test is still red! Can you see why? It looks like the quantity decreased by 3. This is probably what happened: You have the method called `isNormalItem()`, but it doesn't know about the new `Conjured` item type yet. Add it to the list of item types: An item is a normal item if it's not a backstage pass, a `Sulfuras`, aged Brie, or `Conjured`.

Rerun the tests: They should all be green! If they are, you have successfully implemented the new feature.

Look at the `updateQuality()` method, and again, it is now straightforward to read and modify. The pin-down tests have helped you keep everything safe.

Conclusion

You have practiced several different refactoring techniques in this series:

- Rename variable
- Rename method
- Make method static
- Move method
- Move class
- Inline code
- Extract method
- Extract constant
- Extract variable
- Change signature
- Simplify arithmetic
- Invert Boolean expression
- Simplify Boolean expression
- Group related business logic
- Delete unused code

Continue practicing these techniques. You can repeat the code kata multiple times. The more you practice, the more muscle memory you will build so you will be better prepared to apply these refactorings in your daily coding challenges. All this will lead you to simplify your code to build and deliver high-quality software.

Dig deeper

- Refactoring Java, Part 1: Driving agile development with test-driven development
- Refactoring Java, Part 2: Stabilizing your legacy code and technical debt
- Test-driven development: Really, it's a design technique
- Gilded Rose refactoring kata by Emily Bache
- Using comments to design classes
- Simplified test-driven development with Oracle Visual Builder
- Book: *Refactoring to Patterns* by Joshua Kerievsky
- Book: *Refactoring: Improving the Design of Existing Code* by Martin Fowler
- De Morgan's Laws

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