Refactoring in a "Test First"-World

Jens Uwe Pipka
Daedalos Consulting GmbH
Ruhrtal 5
58456 Witten, Germany
+49 2302 979 0
jens-uwe.pipka@daedalos.com

ABSTRACT
Enforced by the "Test First, by intention" principle as proposed by Extreme Programming (XP), application code grew up with unit tests [1]. During the development process, this principle normally builds a stable base for restructuring the existing code continuously, e.g. to introduce new functionality. This is done by refactoring the application code and verified by running the existing unit tests. But applying Test First and refactoring consequently, some kind of paradox occurs: In many cases, refactoring application code also affects unit tests. So, the correctness of the refactored application code could not be verified anymore.

In a "Test First" World, the solution to refactor application code successfully and to prove this by the corresponding unit tests is quite clear: First, adapt the unit tests with respect to the target refactoring. Second, change the application code. Finally, run the unit tests. In this paper, we present how the Test First approach could be applied consequently during the refactoring process and how to keep unit tests synchronous with application code using Test First Refactoring.

Keywords
Automated Testing, Unit Testing, Refactoring, Test First, Extreme Programming.

1 INTRODUCTION

Unit tests provide a powerful technique to develop new functionality as well as to change existing parts. This is done by defining tests for a complete unit of work. By it, unit tests are also essential for refactoring [2].

Normally, a refactoring should not break running unit tests. Nevertheless, it is also possible that the refactoring of application code also affect existing unit tests. To sum it up, you can decide between the following situations:

1. The refactoring has no side effects on existing unit tests, e.g. Extract Method.
2. The refactoring has clear effects on existing unit tests, e.g. Rename Method. The affected unit tests can be adapted by easily.
3. The refactoring breaks existing unit tests that are coupled to tightly with the application code. This situation often occurs during structural refactorings, e.g. when dealing with generalization such as Extract Superclass.

In real life development, the second and third situation is quite common, i.e. that the refactoring of application code breaks existing unit tests (see Figure 1).

During software development and maintenance, this leads to the dangerous situation that application code and unit tests are no longer synchronous. Even worse, the application works as it should but the unit tests do not run anymore.

So, it is necessary to adapt the unit tests as well. The quick and dirty approach to do this is to refactor the application code first, to run the unit tests, to check their results, and finally to adapt the unit tests that have been broken. However, unit tests should be modified more carefully to preserve their original semantics [3].

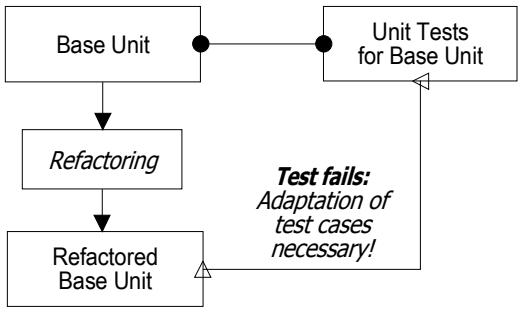

Fig. 1: Refactoring and Unit Testing

As we will show in this paper, a lot of control and safety is lost when refactoring is done in this way. Instead, we focus on adapting the unit tests first. Afterwards, the refactoring itself is applied on the application code. So, unit tests could be used for the program verification in the original sense again.

2 HOW TO KEEP UNIT TESTS SYNCHRONOUS WITH THE IMPLEMENTATION

During refactoring the program syntax is changed but the semantics still remain the same. The naive approach to refactoring is simple: First, apply the selected refactoring

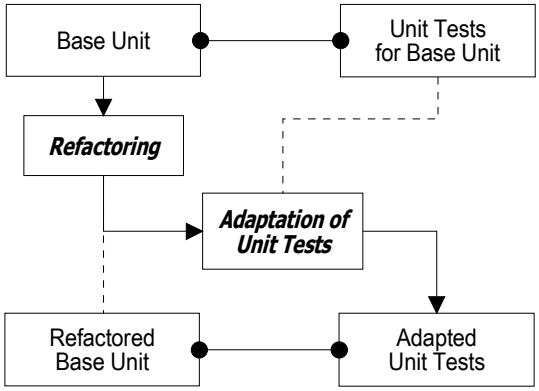

Fig. 2: Refactoring in a Traditional Test World

to application code. Second, run the unit tests. If the code was successfully refactored, the bar is green.

But in real life, refactoring is much harder: After refactoring the application code, it is possible that the unit tests do not run anymore. In a traditional development process, the problem is detected only when the unit tests are run after the application code was already refactored. In this case, the broken unit tests must be changed. This workflow is illustrated in Figure 2.

But what guarantees that only the unit tests are broken and not the refactored code? This question has always to be asked, even if tools like a Refactoring Browser are used to support the refactoring process. Otherwise, a refactoring tool has also to include a full semantic verification for the source code transformation.

XP proposes a Test First strategy to develop software differently: First of all, tests have to be defined. Only after that, it is allowed to implement application code Applying this approach consequently means to broaden it to refactoring.

But what does this mean in practice? First, it is necessary to find out if a refactoring could cause unit tests to fail. This could be done with a list of "critical" refactorings. Then, the refactoring workflow is modified with respect to the Test First concept as follows:

1. Adapt unit tests with respect to the target refactoring.
2. Run unit tests: For the situations presented in this paper, it is expected that the unit tests fail.
3. Refactor application code.
4. Run unit tests: If a unit tests is broken, check if the target refactoring is applied correctly. Go on with the previous step unit all unit tests are running again.

You can find an illustration of this refactoring workflow

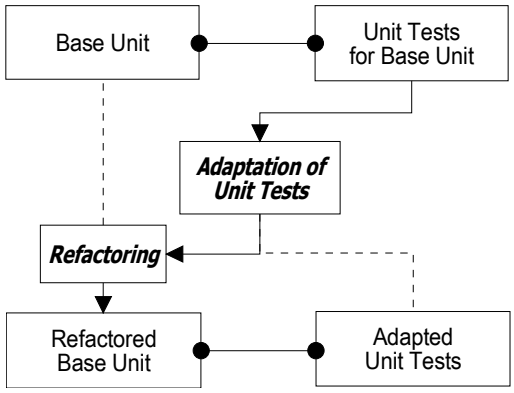

Fig. 3: Refactoring in a Test First World

in Figure 3.

3 TEST FIRST REFACTORING IN DETAIL

As mentioned before, refactoring application code could have different effect on unit tests. So, it is necessary to check whether or not a refactoring also affects test code.

It is out of the scope of this paper to provide a complete list of refactorings that could also have effects on unit tests. Instead, we concentrate on Test First Refactoring in the context of the following refactorings to illustrate the side effects of refactoring on application as well as on test code:

- *Rename Method*
- *Extract Superclass/Subclass*
- *Collapse Hierarchy*

These refactorings had been chosen because they show the problems to keep application and test code synchronous during refactoring. They also represent different refactoring activities: Rename Method changes the existing system as it is, Extract Superclass/Subclass introduce a new entity, and Collapse Hierarchy removes an existing one.

Furthermore, Extract Superclass/Subclass and Collapse Hierarchy are quite complex. Besides dealing with generalization, other refactorings are also used, like moving features between objects. So, these refactorings give a good overview of the consequences of Test First Refactoring.

Rename Method

First, we start with Rename Method. This is a typical example for the situation that a refactoring has clear effects on existing unit tests. Unit tests can be easily adapted to keep synchronous to application code by renaming all references to the renamed method.

If a tool like a refactoring browser is used, this is done automatically. Though, in many companies it is still common to use Java development environments without refactoring support, e.g. VisualCafe. So, it is important to find alternative ways to refactor the code safely. In a Test First style, this can be done as follows:

- Check all corresponding unit tests for all calls to the target method. Rename all references in the test classes. If necessary, adapt also the name of a test method.
- Run your unit tests. If the renamed method is referenced, this must fail. Test First Refactoring leads to an additional verification at this point: Only if the test fails with "method <new name> not found", the Rename Method refactoring is possible as foreseen. Otherwise, different error situations are possible, e.g. a method with the same name is already defined inside the inheritance hierarchy.
- Apply the refactoring: Rename the method and all its references inside the application classes.
- Run the unit tests again. If you have refactored your

code correctly, the bar should now be green again.

Even if Rename Method is a very simple example, it already shows how helpful Test First is. So, it is verified whether or not a method with the chosen name is already visible in the class. Malfunctions and side effects are avoided directly at the beginning of the refactoring process.

Extract Superclass
Extract Superclass means to divide one class into two or more classes. So, at least one new class is introduced into the system. By default, no unit tests are provided for this new class.

Considering that unit tests should not test the internals of the design but the functionality of the unit, you have two different possibilities: First, you can leave unit tests untouched and run the tests when you have refactored your system class. Second, you can introduce additional unit tests for the new superclass. The existing unit tests are changed, so that the moved functionality is tested with the new superclass. Unit tests referencing the original class stay unchanged.

Both alternatives have their advantages. In the first case, the behaviour is exactly the same as before if all unit tests run. This is what you want to prove. However, you introduce an untested class in your application that will be used in future. So, the second case is also important: If the extracted superclass could be used inside the system, it is essential to provide the corresponding unit tests for this class. Otherwise, new functionality could be provided relying on code that is not tested anymore. So, this is a pragmatic way to increase the test coverage of application code.

The decision what to do depends on the situation you develop: If you extract an abstract superclass, it is fairly easy: You do not need to adapt your unit tests because the extracted superclass could not be used on its own.

But if the extracted superclass can be used alone, you should first introduce a new test class. During the refactoring process, the corresponding unit tests are moved to this new class one by one. This is done in small steps: First, you move a unit test to the new test class. This tests a part of the old class that should be moved to the extracted superclass. Next, you run both sets of unit tests, i.e. the one for the old classes and the one for the extracted class. Then, you refactor the application code, i.e. you move the corresponding part from the subclass to the new superclass. Finally, you run the unit tests again. After the bar gets green, you can go on with this action until all parts have been moved to the new superclass.

Extract Subclass
In contrast to the previous example it is necessary to adapt your unit tests anyway. If you extract a subclass, the original test cases will not run anymore, e.g. because a tested method is moved to the subclass. Reason for this is that the behaviour of the original class is restricted to a functionality that is common for all subclasses that are foreseen.

Again, you have two alternatives: First, you can change the corresponding unit tests by exchanging the old class against the extracted subclass. Second, you can create a new test class for the extracted subclass. All unit tests for methods that you want to move to the new subclass have to be moved the new test class.

Both alternatives could be useful. The first alternative is implemented very quickly and guarantees that the extracted subclass has still the same behaviour as the original before. But once more, you introduce a class in your system without its own test code.

So, the second alternative seems to be the better choice: You start with creating a new test class for the extracted subclass. Then, you move the unit tests one by one to this new class. After that, you run both sets of unit tests, the set for the original as well as the set for the new class. Finally, you refactor the application code and run all unit tests again. The advantage is, that you provide a set of running unit tests for the extracted class. So, it is treated like any other class in the system and could be used as well in the same way.

Collapse Hierarchy
Finally, we discuss which adaptations inside existing unit tests are necessary for Collapse Hierarchy. This refactoring is suitable every time when a superclass and a subclass are very similar. It is obvious that this refactoring will break your tests every time a class is accessed that has been removed.

Applying the Test First strategy, the following steps will guarantee the semantic equivalence of the implementation before and after refactoring the code.

First, the corresponding unit tests are changed by renaming all references to a class that should be removed to the name of the merged class. If possible, the different tests should also be merged to a single one. It is not really necessary to do this, but there are several advantages: First of all, you can reduce the number of test object that have to be initialized. Furthermore, your unit tests are clearly arranged after the merge and it is easier to maintain them.

Afterwards, it is time to run the modified tests: The unit tests should be broken for all references to methods and variables that are defined in a class that should be removed. Otherwise, these methods could be overwritten in the subclass. Applying Collapse Hierarchy would be a bad choice then, because it would break the implemented behaviour.

Else, you can merge the selected classes as described in the Collapse Hierarchy refactoring. Finally, run your unit tests again. The bar should be green again if you have completed the refactoring.

4 WORKING WITH TEST FIRST REFACTORING

Now, have a closer look on an example to show the advantages of Test First Refactoring in practice. To illus-

trate the general line of action, we have chosen *Extract Method* as an example.

We use a *Cash Machine* that provides some internal method getCash() to manage withdrawing. There is also a specialized cash machine EUCashMachine that offers the opportunity to withdraw a certain amount as cash or as a cheque (see Figure 4).

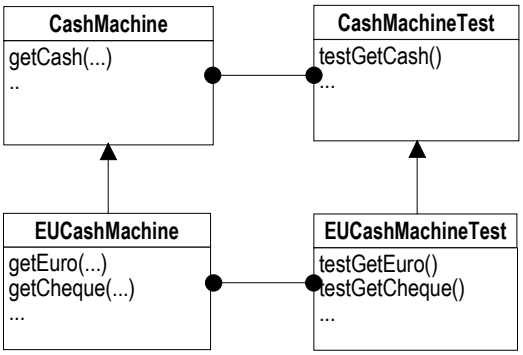

Fig. 4: Class hierarchy for Rename Method

Inside the class EUCashMachine, the method getEuro() should be renamed to getCash(). Applying Test First Refactoring, the following steps are carried out:

- Modify EUCashMachineTest: All references to getEuro() are renamed. Furthermore, testGetEuro() becomes testGetCash().
- Run unit tests: A method with the name is found. That must not happen, so we check our refactoring. A method with the same name already exists in the superclass, so we have to choose another name. Here, we chose customerGetCash().

  Nevertheless, at this point you should always have a closer look at the existing method and its behaviour, because it can perhaps be reused or at least renamed if an ambiguous name had been chosen.

- Modify EUCashMachine: All references of getEuro() are renamed to customerGetCash().
- Run unit tests: customerGetCash() is not found. This is exactly what is expected, so we can go on this time.
- Refactor application code: Rename getEuro() to customerGetCash().
- Run unit tests: If the bar gets green again, everything is okay.

Finally, we have successfully refactored the application code. Test First Refactoring has saved us from changing the system in a way that was not intended.

5 CONCLUSION

XP is based on Test First Design to support agile software development. However, this practice is very often not applied during refactoring. Nevertheless, it is a good choice to adapt your unit tests first and then to refactor your code.

Very often, unit tests are coupled (too) tightly to application code, because they grew up with it from scratch on. In this case, you should normally change unit tests to test the functionality of the unit instead of design internals. But if time has run short and this is always true in software development, it is quite common to fix unit tests only to make them run again.

To make the development process smarter and to introduce more safety, it is essential to adapt unit tests before refactor your code. Test First Refactoring forces to adapt unit tests first of all. Doing this, it is guaranteed that all tests are preserved from removed components as well as expanded to new components during the refactoring process.

Furthermore, Test First Refactoring provides additional checkpoints that reveal missing or overseen prerequisites. This prevents that a developer changes the system behaviour during refactoring. The Software gets more reliable if the Test First approach is applied consequently in all stages of the development process - and this includes refactoring.

REFERENCES

1. Jeffries R., Anderson A., Hendrickson C.: *Extreme Programming Installed*. Addison Wesley, 2000.
2. Fowler, M.: *Refactoring – Improving the style of existing code*. Addison Wesley, 1999.
3. Deursen A. van, Moonen L., Bergh A., Kok G.: *Refactoring Test Code*. In the Proceedings of the 2nd International Conference of eXtreme Programming and Flexible Processes in Software Engineering, Cagliari, 2001.
4. Langr J.: *Evolution of Test and Code via Test-First Design*. At http://www.objectmentor.com, March 2001