Development Upside Down: Following the Test First Trail

Jens Uwe Pipka
Daedalos Consulting GmbH
Ruhrtal 5, 58456 Witten, Germany
Tel. 02302 979 0, Fax. 02302 979 199
Email: jens-uwe.pipka@daedalos.com

ABSTRACT.

Unit tests are widely accepted nowadays in object oriented programming to reduce the error rate in application code. Furthermore, refactoring serves a standard technique to clean up code and achieve a better code quality [1]. During the last years, both techniques prove their worth in many software projects. This is also strengthened by the development of powerful refactoring tools and flexible frameworks for unit testing.

It became obvious that both techniques complement each other well: On the one hand, unit tests serve as a safety net for refactoring - to preserve the behavior unchanged. On the other hand, refactoring facilitates further development of test and application code. Nevertheless, it turned out that developing test and application code in parallel is not enough for a couple of reasons. For example, there could remain application code that is untested. Furthermore, the definition of unit tests also sharpens the interfaces and therefore the software design is improved. These issues are addressed by Test Driven Development (TDD) that becomes an accepted concept during the development of central software components that have to provide high quality.

In TDD, new code is only written when an automated unit test has failed. In detail, you have to follow a clear procedure to add new application code: First, you write a little test that follows the intention and does not work in the first place. Next, you make the test work as quick as possible. Finally you eliminate all of the duplication that is created to get the test run. Again, refactoring is an important technique to bring the TDD approach to life: It is a common procedure to change the application code under the assumption that the "observational equivalence" is still given, i.e. all tests are still passing [2].

And here is where a new concept, Test Driven Refactoring, comes to play: In many cases, refactoring the application code also affects the test code. Thus, it is also necessary to adapt your unit tests. But how can your unit tests prove that the "observational equivalence" before and after the refactoring is still the same when they have to be changed themselves? The solution for this issue is to refactor application code driven by unit tests: First, you start to extend and adapt the existing unit tests itself with respect to the target refactoring. At this moment, you already can examine if the target refactoring really reflects your intention. Next, you apply the target refactoring to your application code. Finally, you run the unit tests and verify if the refactored application code behaves in the way you have expected it in your unit tests before.

Test Driven Refactoring is not only applicable when implementing code in TDD style. It is also an important approach when developing and refactoring in a more classic style. This is based on the fact that Test Driven Refactoring sorts out if a refactoring is applicable without changing the behavior of your application code. Furthermore, this approach defines a refactoring route and thus backs you up to find a specific way applying the target refactoring from the beginning to the end.

This practitioners report presents how unit tests and refactoring strengthen software quality. Beyond it, this report focuses how these concepts can be complemented by Test Driven Refactoring, which extends the "Test First, by Intention" approach towards refactoring [3][4]. It shows how the refactoring purpose as well as its system wide effects gets easier to grasp and application code could better be extended, maintained and reused.

1. Introduction

During the last years, application development gets more and more complex. Today, it is increasingly important that applications have to be delivered within time. Furthermore, it is essential that an application is highly customizable as well as adaptable to meet changing requirements.

Together with object oriented programming languages like Java that offer a wide choice of different technologies, application development often leads to muddled application code. Refactoring is one of the

core techniques to disentangle this tangled application code to enable extending as well as reusing existing application code. Nevertheless, refactoring is often used to clean up application code that is already in a muddled state. It does not prevent that application code runs out of style.

To back up the development of proper application code, Test Driven Development evolved in the last few years. In a traditional software development process, the application code is written first and later on additional tests are implemented (see Fig. 1). Thus, it is quite common that test code only covers a small part of the system. This kind of black-box testing also does not reveal anything about the system parts that have a high risk for the whole system but concentrates on specific functionality in separate. Even with the realization of system and integration tests, the feedback for the developer to locate and fix a certain problem is weak.

Fig. 1: Traditional Software Development Process

Using TDD, for any new or extended functionality the corresponding test code has to be written before the application code. Thus, the system grows up with automated unit tests and no part of the system remains untested. Furthermore, the close coupling of test and application code provides a distinct feedback and speeds up the development as well as the maintenance of the code.

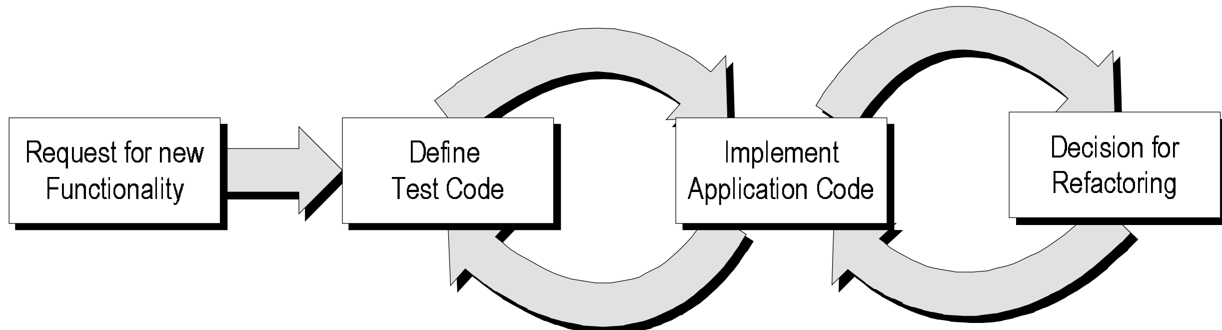

Fig. 2 Test Driven Development Process

Nevertheless, it is necessary to refactor the code continuously. Following TDD, the application code has to be refactored to become as simple as possible. The refactoring should be backed up by unit tests that should be run before and after the refactoring. The driving force behind it is the refactoring of application code: Tests are only changed when the refactoring of the application code breaks existing unit tests. This is done by refactoring the application code and verified by running the existing unit tests (see Fig. 2). But in many cases, refactoring application code also affects unit tests. Thus, the correctness of the refactored application code could not be verified anymore.

Here is where Test Driven Refactoring comes to play: In this approach, refactoring also follows the "Test First" paradigm: If a refactoring has to be applied, the unit tests are checked first and if necessary adapted

with respect to the target refactoring. Only then the application code is changed. When the unit tests run successful again, the refactoring is finally finished and the next development task can start (see Fig. 3).

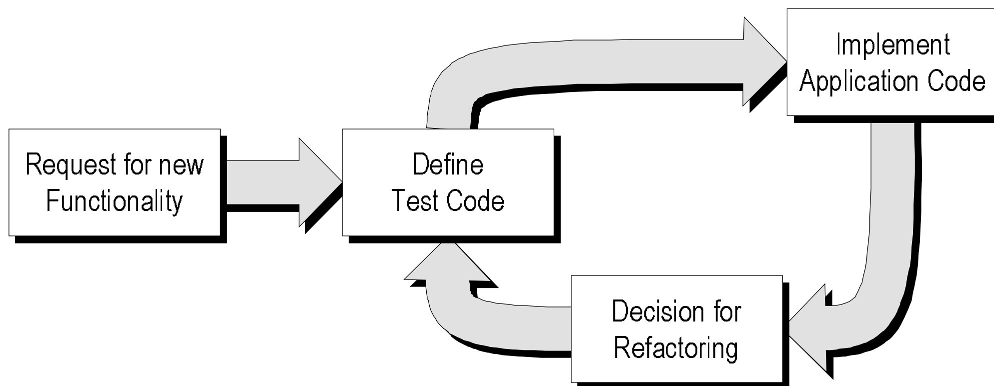

Fig. 3: Test Driven Refactoring Process

2. Improving Software Quality following the Test Driven Development Trail

In daily life, test-driven development supports the creation of stable software systems. Nevertheless, it is a major change of the development process itself. Therefore, we start with a simple but complete definition how TDD is applied:

- Write a little unit test that specifies a small set of the functionality that should be implemented.
- Run the unit test: It has to fail because the functionality is yet not defined.
- Implement the application code that is necessary to pass the unit test.
- Refactor the code until it has the simplest design possible for the specific functionality.

As you see, the steps necessary for implementing new code are quite easy. Additionally, you need also some rules to back up the development process:

- The test is written first
- Everything that could possibly break has to be tested
- All tests are run all the time

But why do we turn the development process upside-down, what are the benefits then? Following this trail offers a lot of advantages:

- The intention of the software implementation becomes the driving part of software development.
- Dependencies between different parts are minimized and interfaces kept clean. In contrast to that, untested code is often highly coupled, a sign for poor object oriented design.
- The software grows up with unit tests from scratch on, i.e. no part of the system is completely untested. Furthermore, the definition of automated unit tests enable to run regression tests at any time.

- Development speeds up, because application code concentrates on tiny bits of functionality. Thus, the application code is built up continuously and complexity is growing slowly.

An important issue to introduce this concept in your daily development work is the integration as well as usability of a testing tool in the development process. We use the JUnit framework as test environment, the de-facto standard for unit testing in Java. Implementing and running unit tests using the JUnit framework has many advantages. With JUnit, testing is closely integrated with development and test code is coupled tightly to application code. So, it is possible to build test suites incrementally that will help to focus on the development efforts as well as to identify errors. Furthermore, JUnit itself is implemented in Java and available as source code. Hence, it could be extended easily [5].

Refactoring is one of the core functionalities in TDD: But even when applying TDD consequently, the test first concept is broken when existing code is refactored: In many cases, refactoring application code also affects unit tests [6]. Thus, the correctness of the refactored application code could not be verified anymore. For example, consider the “Collapse Hierarchy” refactoring. 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 merged into another class as result of the “Collapse Hierarchy” refactoring. A refactoring that could break existing unit tests is also called a “critical refactoring” in the scope of this paper.

In the following we present our approach to widen TDD to refactoring and how to apply the Test First idea consequently in practice.

3. Remain firm: Applying Test Driven Refactoring

TDD requires refactoring your code continuously. Therefore, TDD without refactoring is not applicable. Nevertheless, it is still important how refactoring is integrated in the development process. The intention is to write a unit test that is not successful. Next, you implement your application code. Finally, the unit test is successful. Only if this run is successful, the next step should be taken.

In the previous chapter, we have already described the initial concepts to apply TDD. To widen it to refactoring, the refactoring process is done as follows: First, adapt the unit tests with respect to the target refactoring. Second, change the application code. Finally, run the unit tests. From a practitioner’s point of view, it is essential to apply the development concepts as homogeneous as possible. That is why we extend the Test First approach towards refactoring. Thus, we present in the following how it could be applied consequently during the refactoring process and unit tests are kept synchronous with application code using TDR.

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

- The refactoring has no side effects on existing unit tests.

- The refactoring has clear effects on existing unit tests. Nevertheless, the affected unit tests can be adapted easily.
- The refactoring breaks existing unit tests that are coupled too 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. 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 pass anymore and are thus out of sync (see Fig. 4).

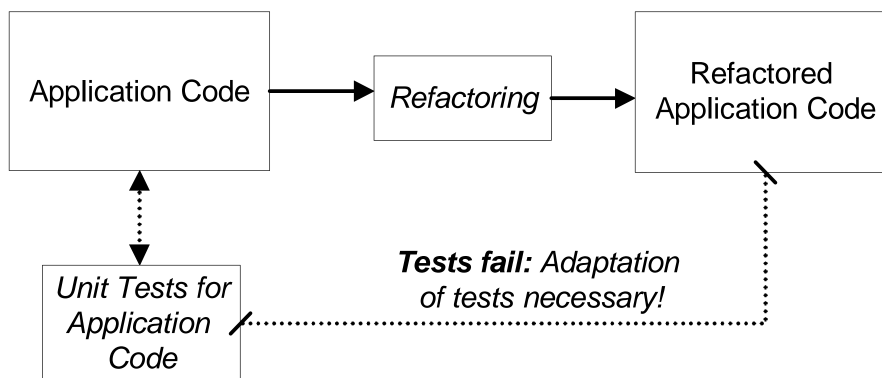

Fig. 4: Refactoring breaks Unit Tests

Following the Test First Trail, the unit tests also have to be adapted first to preserve their original semantics. It is necessary to define a specific refactoring route first. This is the same approach as during TDD: The specific problem is broken down until a concrete task is identified. Thus, it is necessary to find out if a refactoring could cause one or more unit tests to fail. This is done using a list of critical refactorings that potentially could break the unit tests as mentioned before. It is part of our ongoing research to update this list continuously. Therefore, it is necessary to identify critical refactorings and to provide a step-by-step guide how to perform these refactorings especially with respect to the unit test adaptation following the test first concept.

In general, 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 the application code.
4. Run unit tests: If a unit test is broken, check if the target refactoring is applied correctly. Go on with the previous step until all unit tests are running again.

Following this approach, all effects of a refactoring on the test as well as of the application code are considered. The consequences for the application code are applied to the test code first. Already at this stage,

it is checked whether the chosen refactoring is suitable for the implementation or not before modifying the application code. Thus, the application code gets much cleaner applying TDR consequently.

4. Impacts on the Software Development Process: Test Driven Refactoring in Practice

The principles as presented before already reveal that following the “Test First Trail” is not easy going. Instead, it is obvious that this approach requires a lot of discipline and turns traditional development upside down by starting with the test implementation. To support this working model, it is necessary to define a set of rules that are applied throughout the whole development process:

- Each requirement must be break down to a technical task that can be described by unit tests before starting to implement anything.
- For each technical task, the development starts with the implementation of the appropriate unit tests, i.e. at least the common behavior of the technical task must be checked by the unit test implementation.
- The application code behavior relies on the corresponding unit tests.
- Any change within the application code is reflected first in the corresponding unit tests. This is also applied for any refactoring that could break the test code.
- After any change of test or application code, at least the corresponding unit tests have to be run.
- When finishing a technical task, all unit tests have to be run successful.
- No critical part of the software should remain untested. If possible, this should also be applied to third party components that implement a specific behavior. At least these components should be tested indirectly by the unit tests of the accessing application code.

But why do we change the development in such a radical way? Our experiences with TDD was excellent, so we decided to apply it consequently during all our development activities and therefore extend it to refactoring. In detail, the following positive effects took place:

- As mentioned before, it is essential to break down an abstract requirement to a concrete development task when you develop your software test-driven. If you can define a unit test, you know the interface and behavior of the application code implicitly. Therefore, the interface gets much cleaner compared with traditional development.
- The same applies to the application code itself. Starting with the expected result of an implementation unit helps to concentrate on the core functionality. Thus, the application code is focused on the real task rather than on additional future functionality.
- The test code serves as an additional interface and usage description of the application code. Thus, test-driven development also defines some kind of showcases how to use the application code.

- These effects help to integrate new team members much faster. Test and application code provide an easy starting point for a special part of the software.
- Working with clean code and clear interfaces speeds up implementing new, extending existing and refactoring old application code.
- Last but not least, your application code grew up with test code that can be executed automatically. Therefore, your system becomes much more reliable and changes are possible throughout the whole system. And here, TDR is also an important factor to improve development: TDR also strengthens to change working code without the fear to break it.

Furthermore, it is necessary to use the right tools to support this kind of development. This is essential, because the whole development cycle should be as consistent as possible. The best results are possible if the next step is the conclusion from the previous one as already shown in the general principles before:

- A handy but nevertheless flexible, powerful unit testing framework is needed. It has to be embedded in the programming language and development environment. For Java, JUnit is the perfect companion to start with TDD.
- To apply TDR, it is very useful if the integrated development environment (IDE) supports at least a small set of refactoring tools. In contrast to the traditional development process, those automated refactoring tools could be used to start with refactoring the test. After modifying the test code, the refactoring can be applied to the application code.
- The better the integration of unit test and refactoring tools in the IDE is, the better the Test First approach works. With Eclipse, a free IDE is available that integrate JUnit as well as a powerful set of refactorings. The same is true for IDEA, another Java IDE with powerful automatic refactoring support. Together with other useful Java tools, these kinds of IDEs could be used most effective for TDD and TDR.

5. Experiences

Finally, the crucial question is whether or not is it worthwhile to apply TDD and TDR? At the beginning, it is hard to apply TDD and TDR within the daily work consequently, because it turns the traditional, well known development upside down. Nevertheless, after the first successful steps into this world, the acceptance is increased heavily. The benefits are high; development becomes easier and more reliable.

We used TDD and TDR in a project that realizes a web front-end for a Cobol application. This was a critical project, because at the beginning interfaces changed quite heavily as well as a complex infrastructure has to be supported. Thus, it was essential to implement a testable system already during development to locate an error within the application code early enough. Starting with traditional development it was quickly clear that tests were only implemented when something went wrong – and this was already too late and changes within the application code were quite expensive.

Therefore, we decided to enforce the implementation of unit tests. The first reaction to this suggestion was divided in two: Despite from the fact that apparently the number of tests is increased as intended, how is it possible to define a test without the knowledge what application code has to be tested? At the beginning, it was necessary to coach the way to break down requirements to technical tasks and to extract appropriate tests from these technical tasks. Furthermore, it was also necessary to customize the development environment as well as the development process to get the expected results. But within a short time, the advantages became obvious: Errors could be fixed within minutes, at any time it was possible to run the unit tests and to locate as well as to fix critical application code.

This positive experiences lead to another effect: If a malfunction is overseen and becomes obvious later on, it is normal that any fix is initially motivated by another unit test that complements the existing unit tests – because it is much easier to verify the application by a unit test as to test it manually.

At some point of time, it becomes obvious that traditional refactoring does not fit into this process: Unit tests do no longer run, even if the application code has still the same behavior. Thus, we extend the approach to refactoring and motivate each refactoring by changing the unit tests first. This lead to the situation that the refactoring process is fitted in the daily project life just as any other technical task.

After all, the application code is stable and clean, documented and examinable by test code. Furthermore, the software was developed in time with much more test code as originally expected and therefore much less errors as feared. Thus, the mistrust against TDD and TDR disappeared finally.

6. Conclusion

At the end, the crucial question remains: Is it really possible to use TDD and TDR in a software Project? The answer is as always “yes, but”…

“Yes”, because TDD and TDR provide a practical way to implement software in an acceptable time with a high quality and a low error rate. It does not speed up development at the beginning, but it reduces maintenance costs as well as the number of errors. Furthermore, the existence of automated unit tests backs up any changes and extensions. Thus, the development speeds up in long term.

“But”, because the team must follow the test driven trail consequently. TDR complements TDD and offers additional safety when existing application code including the corresponding unit tests have to be refactored. It follows the same principles and is therefore the consequent realization of “Test First” thinking: The test comes first, in any situation.

Thus in the end, TDD and TDR support something that seems on the one hand not very spectacular, but is on the other hand extremely rare in software development: It produces software in time and in quality.

References

[1] Fowler, Martin: *Refactoring – Improving the style of existing code*. Addison Wesley, 1999.

[2] Beck, Kent: *Test Driven Development by Example*. Addison Wesley, 2002.

[3] Jeffries R., Anderson A., Hendrickson C.: *Extreme Programming Installed*. Addison Wesley, 2000.

[4] Pipka, Jens Uwe: *Refactoring in a "Test-First" World*. Proceedings of the XP 2002.

[5] JUnit Homepage, http://www.junit.org

[6] Westphal, Frank: *Testgetriebene Entwicklung mit JUnit*. dpunkt Verlag, to appear in 2003.