



# Introduction to Test Driven Development (TDD)

rain  
rel

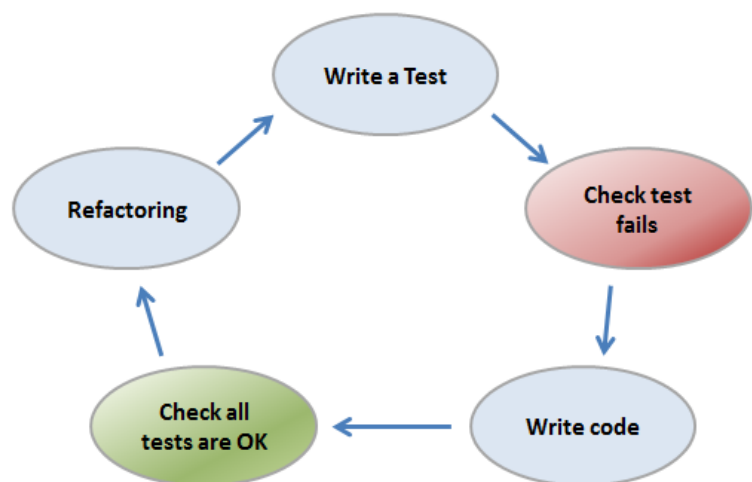
TWEET THIS

preneur

eloper  
ger  
or

*The Test Driven Development (TDD) is a software engineering practice that requires unit tests to be written before the code they are supposed to validate. Coming from the Agile world in which it is a basic practice of the Extreme programming (XP) method, TDD is nowadays recognized as a discipline in its own right that is also used outside the agile context. An overview and an introduction to a practice that you will not be able to do without!*

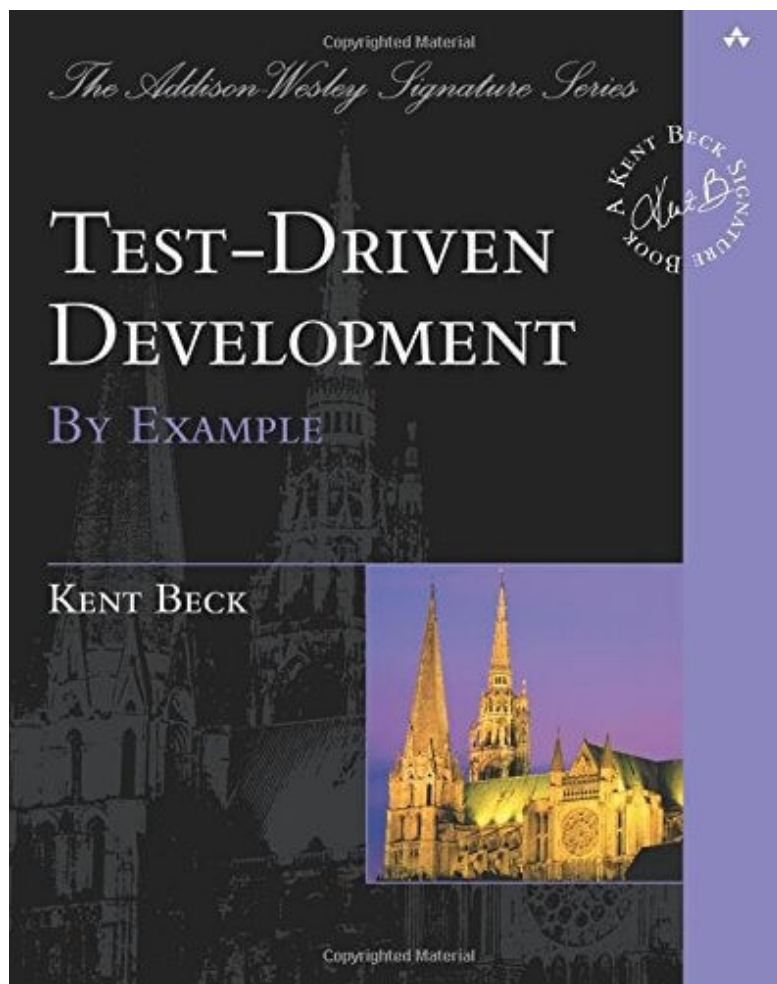
## Introduction to Test Driven Development



The Agile movement has continued to progress over the past fifteen years, leading to new and more pragmatic practices aimed at providing the client with the program that meets his needs as quickly as possible. For this problem, **traditional methods based on the V-Model offer a**

**“Test-Last” approach** with unit tests written after the application code. **This approach has clearly shown its limitations over time** since, at best, written tests are adapted to the code and not the other way around, thereby introducing a confirmation bias. In the worst case, these tests are not even written since the code seems to work correctly, why waste time writing them?

The answer to this question is obvious to TDD supporters who are well aware that the investment made today on tests will be largely refunded in the future when adding new features or refactoring operations. Thus, **the TDD adopts a “Test-First” approach in which unit tests are written before code**, the only reason for which will be the successful execution of these tests. This idea, which dates back to ancient times, was formalized in the mid-1990s by **Kent Beck**, who made it **one of the pillars of the Extreme Programming (XP) methodology**. It was then necessary to wait until 2003 and the publication of the book **“Test Driven Development : By Example”** to see it complete the codification of practice. **The latter adds to the “Test-First” approach the notion of continuous refactoring** with a view to improving product code.



## TDD Principles

By **combining programming, unit test writing and refactoring, TDD is a structuring practice that allows to obtain a clean code**, easy to modify and answering the expressed needs which remains the first priority when developing an application. **The TDD has 3 phases:**

1. **RED.** First write a unit test in failure. The impossibility of compiling is a failure.
2. **GREEN.** Write as soon as possible the production code sufficient to pass this unit test even if it means allowing the “worst” solutions. Of course if a clean and simple solution appears immediately, it must be realized but otherwise it is not serious the code will be improved incrementally during the refactoring phases. The aim here is to obtain as soon as possible the green bar of success of the unit tests.

3. **REFACTOR**. This phase is often neglected but is essential because it eliminates possible code duplications but also makes it possible to make changes in architecture, factorization, presentation... This refactoring concerns both the production code and the test code and must not modify the external behavior of the program, which is materialized by a test execution bar that remains green.

The implementation of these 3 phases is done within 5 main stages which constitute the TDD cycle (Figure 1).

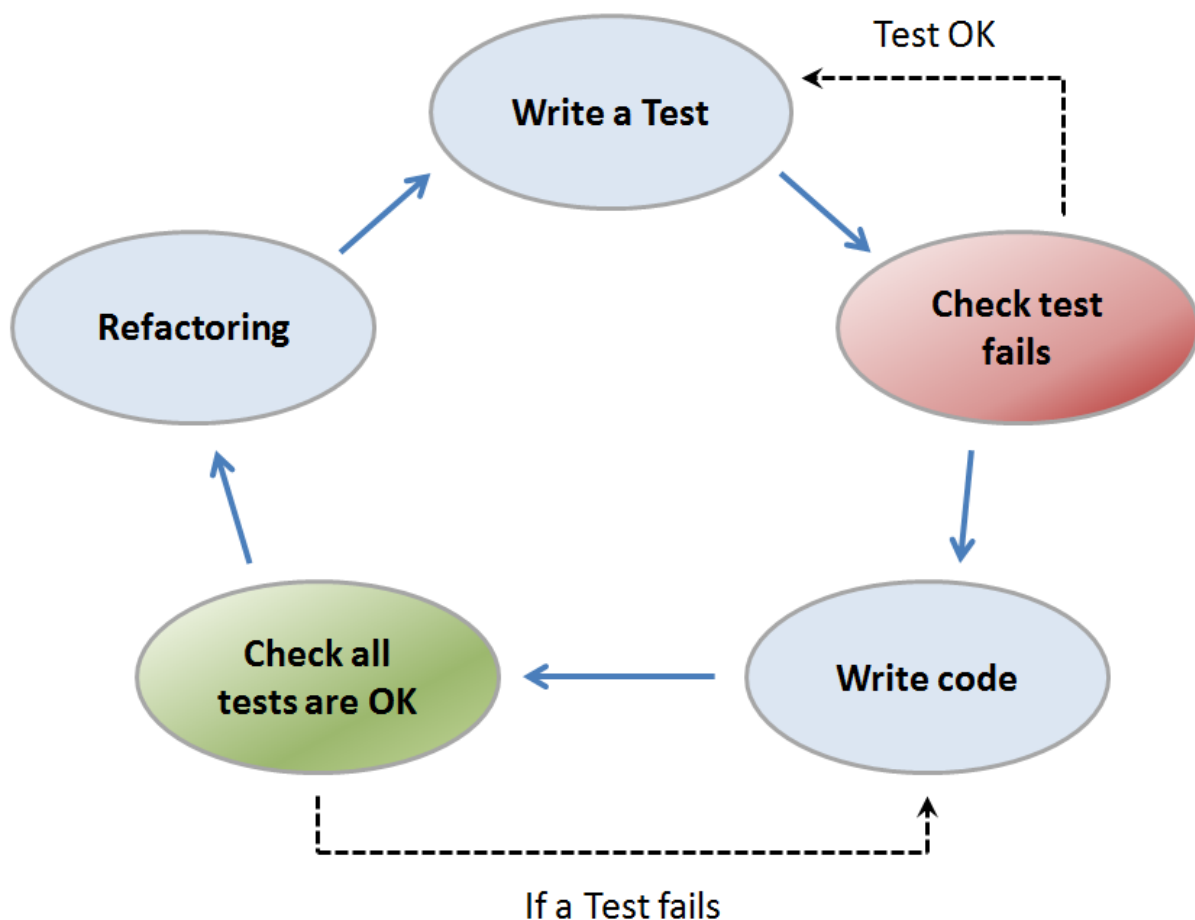


Figure 1 : TDD Cycle

**This cycle is short (10 minutes maximum) and repeats until all unit tests covering a functionality have been successfully completed.** These steps may seem simple, but they must be applied with the utmost rigour to take full advantage of the TDD. This strict monitoring of the rules makes it possible to obtain a quality code that complies with a certain number of

good development practices. Thus, the code obtained is satisfied with the essential (**KISS principle—Keep it simple, stupid**) not seeking to implement unnecessary services (**YAGNI principle—You Ain't Gonna Need it**) while factoring in code duplications (**DRY principle—Don't Repeat Yourself**) thanks to the continuous refactoring to which it is subject. In doing so, the realization of a program follows an incremental logic allowing the emergence of a flexible and modular architecture.

## Clean Tests

**TDD is not a miracle solution** for having an optimal unit test suite without effort. It is important to keep in mind that within this practice, **the test code is as important if not more important than the production code!** Taking care of the test code over time is therefore essential. To be effective, **a test must be clean, which is characterized by its readability**. This is obtained by performing a simple, clear and as dense a test as possible, i.e. making him say as many things as possible in a minimum of code. Thus, **a unit test should represent only one concept and contain only one assertion**.

Finally, **a clean test must follow 5 other rules** that can be easily memorized using the acronym **FIRST**:

- **Fast:** a test must be fast to be executed often.
- **Independent:** tests must not depend on each other.
- **Repeatable:** a test must be reproducible in any environment.
- **Self-Validating:** a test must have a binary result (Failure or Success) for a quick and easy conclusion.

- **Timely:** a test must be written at the appropriate time, i.e. just before the production code it will validate.

## A paradigm shift

The TDD approach is a paradigm shift for the developer. **A learning phase is necessary** during which the benefits will be felt more and more as the developer's know-how increases. The TDD is therefore to be considered as an investment in the future. The change also concerns the document repository with unit tests representing executable specifications and application documentation. Indeed, if tests are used to validate requirements, they can also describe them. Based on this observation, the major difficulty will lie in the developer's ability to develop a roadmap for complex functionality in the form of planned tests, and to be able to question it if necessary.

**The TDD allows problems to be detected as early as possible**, which has the effect of reducing the cost of resolution but also the number of bugs. The code coverage provided by the unit test series is intended to be maximum with a minimum of well over 80%. **This safety net gives the developer the confidence to carry out the refactoring and continuous improvement work** essential to the success of the method. De facto, **the technical debt is better controlled** with a flexible, maintainable and reusable code that facilitates the addition of new functionalities.

Finally, **the ROI of the TDD is also reflected in an overall increase in developer productivity**. Indeed, if it pushes to write in general twice as fast, it leads to a real waste hunt during the refactoring phases which lead to half as much code. Once the learning is digested, we observe a paradigm shift in the developer's way of working with a shift from the debugger to the series of unit tests and the famous green bar synonymous with successful execution of the latter.

## Choose good tools

For best practice, the TDD must be coupled with good tools. An IDE like Eclipse with native JUnit support is essential. The use of plugins to facilitate the management of unit tests such as MoreUnit and Infinittest is strongly recommended. The latter automatically executes all unit tests at each code change, which reduces the feedback cycles that also lay the foundations for continuous unit tests. On the other hand, the use of code templates for unit tests is an important time saving in the repetitive TDD cycle. At the code level, for the generation of readable and flexible business objects, the design pattern Builder is essential. Finally, a good mastery of keyboard shortcuts saves valuable time.

## TDD in practice

As in martial arts, **the practice of a technique can be done in katas** where the developer must solve a specific problem from scratch. The site **<http://sites.google.com/site/tddproblems/all-problems-1>** offers a good variety of problems that are particularly suitable for resolution using the TDD approach. For our example, we choose a program to convert a number in Arabic format into its equivalent in Roman format.

Note that you can discover the **Roman Numerals Converter from Arabic Numbers TDD Kata** in video on YouTube :

## TDD Kata - Roman Numerals Converter from Arabic Numbers



As required by the TDD approach, we first write the *RomanNumeralTest* class which will contain the program's unit test series. The first requirement of the latter is that the input digit 1 gives the output Roman digit I which is implemented below:

We were running this first test in order to make it fail (Figure 2).

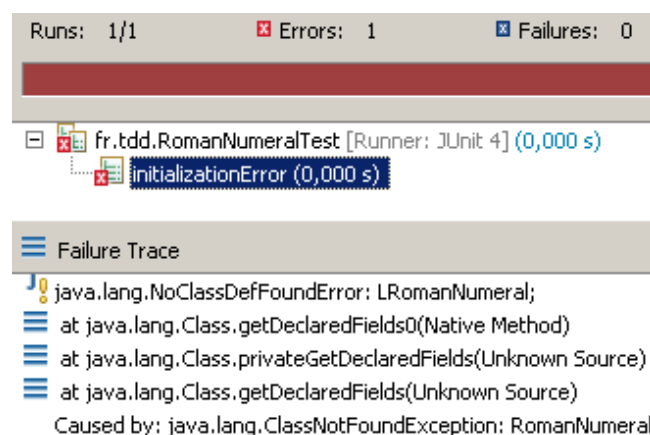


Figure 2 : RED step in the TDD Cycle

A compilation error being a failure, we proceed to write the production code that will satisfy this test. To do this, we place ourselves on the line



where the *RomanNumeral* class is located and use the keyboard shortcut Ctrl + 1 of Eclipse which offers a “Quick Fix” to generate an empty *RomanNumeral* class. We then generate the *intToRoman* method of this one always via this menu. At this stage, we write the simplest code to satisfy the unit test. In this case, it is a question of returning the string “I” at the output of the *intToRoman* method:

The test can then be successfully performed to move forward in the TDD cycle (Figure 3).

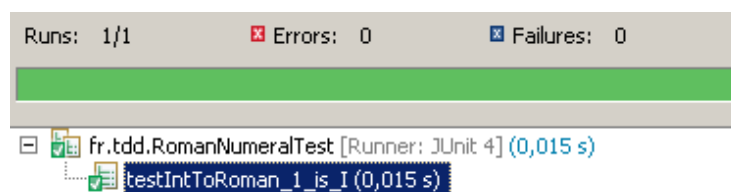


Figure 3 : GREEN step in the TDD Cycle

We are now entering the refactoring phase, which is very fast here since the code contains no duplication and no improvements to be made. The cycle starts again with the addition of a new test:

Once the test has failed, we modify the *intToRoman* method of the *RomanNumeral* class in order to pass this test successfully, which gives us the following code:

As the unit tests turn green, we move on to refactoring. It is preferable to have only one exit point for a method and it is recommended to use braces for an if condition even with a single instruction. Which gives us:

The tests remain green, we expand them with an additional requirement concerning the number 3 which must give the Roman number III as an output. This new requirement puts our unit test in failure and it is necessary to write the following production code to validate it:

The refactoring step highlights a possibility of optimization using a loop that decreases the value of the Arabic digit passed at the input and consequently adds a Roman bar to the Roman digit rendered at the output:

The tests always passing successfully, we are now interested in the number 10 which must give at the output the Roman number X :

Following the failure of the test, the production code is written. Here, we realize that our previous algorithm is not adapted since we are in the presence of a new Roman numeral the X. To remedy this, we add a test to treat the Arabic numeral 10 in a specific way:

The tests pass correctly and the refactoring phase does not require any special work. We add an additional test with the value 20 which must give the Roman numeral XX. This new requirement does cause the associated unit test to fail. The following production code is then written:

This code is sufficient to pass our series of unit tests. The refactoring phase allows us to take a step back and see that it is possible to optimize

our algorithm for Roman numerals containing the X. It is easy to see that a loop would be more efficient than an if / else if. The code is therefore modified as follows:

The JUnit test bar is always green, refactoring has not changed the external behavior of the method. The study of the production code allows us to observe that the work carried out is essentially the same for the figures X and I, i.e. a loop with decrement, which indicates a possibility of factorization. A new design path is thus emerging for our algorithm consisting in introducing 2 tables linked by their index containing the Roman numerals and their Arabic equivalent respectively.

Our unit tests validate this refactoring and remain correct for the number 30. Since it is not possible to defeat the unit test with 30, it is not necessary to change the production code. This is repeated with the numbers 11 and 33 based on the Roman numerals X and I for which the algorithm remains operational. On the other hand, the Roman numeral V leads to a failure of the associated unit test. **We are therefore starting again on a full TDD cycle.** The simplest solution is to add the Roman numeral V and its Arabic equivalent in our tables to check if the current algorithm becomes operational. The tests are going green, so that's the case. For refactoring, **a question arises as to whether it is not better to replace the 2 indexed tables linked by a Java map?** As the values contained in the tables are ordered in decreasing order with a path via 2 interlinked loops, the current solution is preferable because it is simpler and adheres to the **KISS principle**.

We complete our unit tests with the number 4 whose corresponding Roman number is IV. The test is in failure which leads us to modify the production code to pass it. The first approach is to add a new for this

particular case, which turns the test green. The following refactoring highlights that the number IV should be considered as a symbol in its own right. Its addition in the digits table allows to delete the previously added `yew` via the shortcut CTRL + D. The test bar, which is always green, guarantees that the external behaviour of the program has remained the same.

**The TDD development of our program continues** and we are completing step by step our series of unit tests (Figure 4) and our tables with all the Roman numerals and their Arabic equivalents.

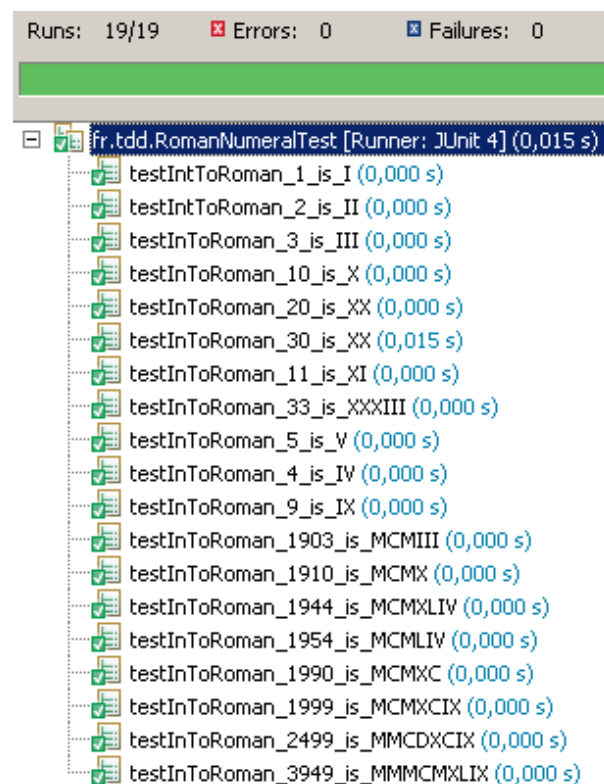


Figure 4 : Series of Unit Tests obtained

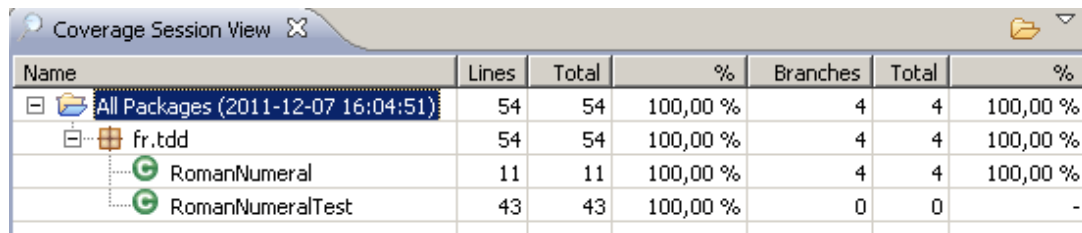
This gives us after about ten increments the following tables:

```
ARABIC_DIGITS = {1000, 900, 500, 400, 100, 90, 50, 40,
10, 9, 5, 4, 1};
```

```
ROMAN_DIGITS =
```

```
{"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
```

The addition of new unit tests with Arabic numerals such as 1954 or 3949 will not ultimately require any changes to the `intToRoman` method of the production code. The unit tests series obtained as a result of this development in TDD mode offers us maximum code coverage (Figure 5).



Name	Lines	Total	%	Branches	Total	%
All Packages (2011-12-07 16:04:51)	54	54	100,00 %	4	4	100,00 %
fr.tdd	54	54	100,00 %	4	4	100,00 %
RomanNumeral	11	11	100,00 %	4	4	100,00 %
RomanNumeralTest	43	43	100,00 %	0	0	-

Figure 5 : Code coverage of our program

## Conclusion

This introductory article on TDD will have shown the power of this practice that every developer must have in his toolbox. To make the most of it, it is necessary, as we have demonstrated in our example, to apply its rules strictly. **The paradigm shift brought about by TDD involves a learning phase before the developer can be fully operational and his productivity significantly increases.**

Based on the example of conversion of Arabic numerals to Roman numerals in this article, a good exercise is to perform the reverse conversion method in TDD. Finally, making katas on the TDD problems of the site presented in this article is the best way to progress. In short, you will have understood it, to progress in TDD a single watchword, valid for any technique, practice!

To discover great books to develop effective Java Unit Tests and learning more on TDD, you can read the following article :

Discover the 7 Best Books to develop effective Java Unit Tests

And if you search the Best Books for learning Java Programming, you can find my selection just here :