# Test Driven Development

## **Wann kann man überhaupt testen?**

Es gibt im Wesentlichen 3 Zeitpunkte:

1. Test last (also Code first)
2. Test first (also Code last)
3. Nach jedem gefundenen Bug

Zu 1

Nachteile:

Der Entwickler neigt dazu, die Implementierung zu testen (die er selbst vor wenigen Minuten geschrieben hat), anstatt die Schnittstelle und ihr Verhalten der SUT zu testen.

Dies kann zu Tests führen, die:

* eng an die Implementierung gekoppelt sind (und deswegen angepasst werden müssen, wenn sich diese Implementierung ändert)
* den Entwickler (unbewusst) dazu verleiten, Test zu schreiben, von denen er weiß, dass sie gelingen werden.
* gar nicht geschrieben werden, weil sich der Entwickler sicher ist, dass sein Code funktioniert und/oder er Zeitdruck hat

Immerhin, in manchen Fällen, z.B. wenn man mit “legacy code” arbeitet, ist dies die einzige Möglichkeit.

Zu 2 Schreibt der Entwickler Tests vor dem eigentlichen Code, so veranlasst ihn das, über das Verhalten (die Schnittstelle und ihren Vertrag) des getesteten Objektes nachzudenken anstatt über dessen Implementierung. Außerdem führt dies zu einer hohen Code Coverage (bis zu 100%), was, trotz aller Bedenken hinsichtlich der *Aussagekraft* von Code Coverage, wünschenswert ist.

Test first kann auf alle Test Level angewendet werden: functional/end-to-end testing, integration testing und besonders unit testing.

Zu 3 Das erste, was man tun sollte, sobald ein Bug gefunden ist, ist sich davon abzuhalten, ihn sofort zu reparieren. Repariert man ihn nämlich „einfach so“, verliert man die Gelegenheit, ihn ein für alle Mal loszuwerden. Dieser Bug könnte später wieder auftauchen (z.B. weil jemand anderes die Reparatur rückgängig macht um, z.B. einen anderen Bug zu reparieren). Schreibt man jedoch einen Test bevor man ihn repariert, so stärkt man das „Sicherheitsnetz aus Tests“ und derselbe Bug wird nicht wiederkommen.

Daher Regel: Schreibe einen Test unmittelbar nachdem ein Bug gefunden wurde! Egal wie man vorgegangen ist (code first oder test first).

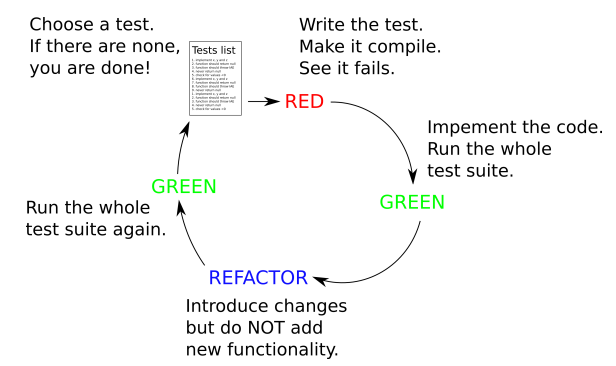
Dieser Test soll den Bug offenlegen (soll heißen, der Test soll das korrekte Verhalten erwarten und zunächst also fehlschlagen). Erst danach repariert man ihn und beobachtet wie der Test nun gelingt.

**Macht man aus dieser Regel ein Verfahren, so ist dieses Verfahren genau *Test Driven Development*!**

Schwierigkeiten tauchen dann auf, wenn der Bug nicht leicht zu orten ist, z.B., weil viele andere Bugs zur gleichen Zeit auftauchen.

Dies löst man üblicherweise so, dass man mit Tests auf höheren Ebenen beginnt (z.B. end-to-end test), und so nach und nach Informationen über die Ursache des Fehlers sammelt, die Stelle des verursachenden Bugs weiter einengt durch fokussiertere Tests (z.B. Integrationstests) bis man schließlich bei Unit Tests angekommen ist.

## **Der TDD cycle (rythm)**



Indem man Tests zuerst schreibt, bekommt man die Chance eine API zu designen, die für einen Client angenehm zu benutzen ist. Die Tests sind der erste Client für diese neu geborene API.

**Darum geht es bei TDD wirklich: das Design einer API!**

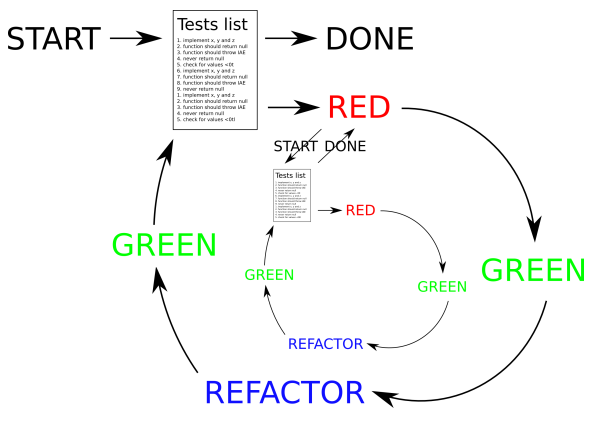
Als positiver Nebeneffekt gibt es keinen überflüssigen Code und nichts was den Code aufbläht:

Typische Fragen, die man sich dabei stellt:

* "Brauche ich wirklich diesen getter der diese collection zurückgibt, oder wäre es doch bequemer eine Methode zu haben, welche das größte Element diese Collection zurückgibt?“
* „Muss ich eine Methode wirklich jetzt schreiben, nur weil sie vielleicht irgendwann einmal gebraucht wird?“
* Sollte ich wirklich getter/setter auto-generieren, wenn eine immutable class passender wäre?“
* Brauche ich dieses JPA-Annotationsattribut wirklich, wo ich nicht mal weiss, was es genau bedeutet?

**Konzentriere Dich auf das, was der Client (also zunächst nur der Test-Code) wirklich braucht. Schreibe dafür und nur dafür Tests und nichts mehr!**

Der Ablauf (cycle/rythm) ist jedoch immer derselbe: Man fängt innen an mit kleinen Units bis diese eine größere Komponente mit möglicherweise externen Abhängigkeiten ergeben. Ein TDD-cycle auf Unit Test-Ebene dauert Minuten, eine TDD-cycle auf Integration Test-Ebene kann Tage oder Wochen dauern.



### **In welcher Reihenfolge soll man die niedergeschriebenen Tests abarbeiten?**

Dafür gibt es 3 gängige Verfahren:

1. Low-Hanging Fruits: Beginne mit etwas sehr Leichtem. Implementiere das Offensichtliche.
   1. Gut gegen „writer’s block“.
   2. Beispiele für low-hanging fruits:
      1. Schreibe ein Parameter-Check für eine Funktion
      2. Teste den Parser mit dem leeren String und bekomme null zurück
   3. Man kann so neue Einsichten gewinnen, die einem den nächsten Schritt aufzeigen
2. Most Informative functionality. Beginne mit dem Test, der die meisten Informationen über die zu implementierende Funktionalität gibt.
   1. Beispiel: Gib dem Parser einen ganzen Satz
3. First Typical Case, Then Corner Cases.
   1. Vorteil: Man hat gleich etwas für den Kunden von Wert geschaffen, selbst wenn man nach dem Typical Case die Arbeit an dieser SUT unterbrechen muss.

## **Der TDD cycle (rythm) am Beispiel einer einfachen Comparable-Implementierung**

Beispiel-Spezifikation: Implementiere eine class FootballTeam, die man mit anderen Teams vergleichen kann um letztendlich herauszufinden, wer auf den ersten Platz in der Liga kommt. Jedes Team enthält die Anzahl gewonnener Spiele.

### **Der erste TDD cycle**

1. **RED** – Schreibe einen Test. Sorge dafür, dass er kompiliert aber nicht mehr. Lasse ihn laufen. Er failt.

|  |
| --- |
| **Spezifikation**: Der Einfachheit halber designen wir eine Klasse FootballTeam welche die Anzahl gewonnener Spiele per Konstruktor übergeben bekommt. Zunächst: Teste, dass der Konstruktor funktioniert. |
| **public** **class** FootballTeamTest {  @Test  **public** **void** constructorShouldSetGamesWon() {  FootballTeam team = **new** FootballTeam(3);  assertEquals(3, team.getGamesWon());  }  } |
| **public** **class** FootballTeam {  **public** FootballTeam(**int** i) {  // **TODO** Auto-generated constructor stub  }  **public** Object getGamesWon() {  // **TODO** Auto-generated method stub  **return** **null**;  }  } |
| **Kommentar:** Schreibe nicht mehr als das Minimum zum Erfüllen der Anforderung (YAGNI). Implementiere nichts! Bei diesem ersten Schritt wollen wir nur den **Test-fail** sehen!  Lasse die notwenigen Klasse , Konstruktor und Methode von der IDE generieren.  Lasse den Test dann laufen. Er hat ein fail. Ist die JUnit-Fehlermeldung bei der Ausführung des Tests nicht sprechend genug, schreibe eine bessere. |

1. **GREEN** – Implementiere die nötigen Funktionalität des SUTs auf einfachste Art. Lasse danach ALLE(!) Tests laufen.

|  |
| --- |
| **Implementierung**: Speichere den Wert des Konstruktor-Parameters als Field in der Klasse |
| **public** **class** FootballTeam {  **private** **int** gamesWon;  **public** FootballTeam(**int** gamesWon) {  **this**.gamesWon = gamesWon;  }  **public** **int** getGamesWon() {  **return** gamesWon;  }  } |
| **Kommentar:** Streng genommen ist die einfachste Art der Implementierung für die getGamesWon()-Methode sodass sie den Test besteht:  **public** **int** getGamesWon() {  **return** 3;  }  Solch eine dumme Implementierung ist dennoch hilfreich: Sie zeigt, dass bessere Tests gebraucht werden um die korrekte Funktionalität der Methode sicherzustellen. Hier könnte man parametrisierte Test durchführen.  Jetzt müssen ALLE(!) Tests laufen gelassen werden. Damit stellt man sicher, dass durch neue Funktionalität nicht alte kaputt gemacht wird. Unit Tests laufen schnell, weil sie keine externen Abhängigkeiten haben. |

1. **BLUE** – Refactor

|  |
| --- |
| **Refactoring**: unter Bewahrung gängiger OO/SE-Prinzipien wie KISS, DRY, SRP und anderen über einem Sicherheitsnetz aus Tests |
| **public** **class** FootballTeamTest {  **private** **static** **final** **int** ***THREE\_GAMES\_WON*** = 3;  @Test  **public** **void** constructorShouldSetGamesWon() {  FootballTeam team = **new** FootballTeam(***THREE\_GAMES\_WON***);  *assertEquals*(***THREE\_GAMES\_WON*** + " games passed to constructor, but "  + team.getGamesWon() + " were returned",  ***THREE\_GAMES\_WON***, team.getGamesWon());  }  } |
| **Kommentar**: An der SUT kann nichts refaktoriert werden. Nur am Test: Konstante ***THREE\_GAMES\_WON***. Die verbesserte Ausgabe im fail()-Fall wurde schon am Ende der ersten RED-Phase durchgeführt. |

1. **GREEN** – Führe ALLE(!)Tests erneut aus. Da dies der erste ist, führe nur diesen aus.

|  |
| --- |
| **Kommentar:** Falls nach Deinem Refactoring alle Tests gelingen, aber nun die Applikation kaputt ist, dann hast Du nicht wirklich refaktoriert. Du hast nur Code geändert.  Refactoring bedeutet sich oberhalb eines **Sicherheitsnetzes aus Tests** zu bewegen. Offenbar haben entscheidende Test gefehlt. |

1. **GREEN**– Falls nötig: Schreibe Tests für mehr Werte, darunter auch ungültige (vgl. Kommentar unter 8.3 2 )

|  |
| --- |
| Mehr Testwerte: Wähle Werte, die zu folgenden 3 Gruppen zählen: erwartete Werte (der sogenannte „happy path“), Grenzwerte und ungewöhnliche Werte |
| @RunWith(JUnitParamsRunner.**class**)  **public** **class** FootballTeamTest {    **public** Object[] nbOfGamesWon() {  **return** *$*(0, 1, 2);  }  @Test  @Parameters(method = "nbOfGamesWon")  **public** **void** constructorShouldSetGamesWon(**int** nbOfGamesWon) {  FootballTeam team = **new** FootballTeam(nbOfGamesWon);  *assertEquals*(nbOfGamesWon + " games passed to constructor, " +  "but " + team.getGamesWon() + " were returned",  nbOfGamesWon, team.getGamesWon());  }    **public** Object[] illegalNbOfGamesWon() {  **return** *$*(-10, -1);  }    @Test(expected = IllegalArgumentException.**class**)  @Parameters(method = "illegalNbOfGamesWon")  **public** **void** constructorShouldThrowExceptionForIllegalGamesNb(  **int** illegalNbOfGames) {  **new** FootballTeam(illegalNbOfGames);  }  } |
| Kommentar: Da die Testmethode  @Test(expected = IllegalArgumentException.**class**)  constructorShouldThrowExceptionForIllegalGamesNb() schon die Implementierung quasi vorgibt, steht es offen, ob dafür ein neuer TDD-cycle begonnen werden soll oder ob dies in diesem Nachgang gleich mit implementiert wird. Ein Argument, diese Implementierung nur als Nachgang aufzufassen ist, dass in der Spezifikation nur *implizit* negative Werte ausgeschlossen waren. In der Implementierungsphase (GREEN) jedoch ist nach der einfachst-möglichen Implementierung nach dem Wortlaut der Spezifikation gesucht. Das Argument gegen die Idee, dies während der Refaktorierungsphase zu implementieren ist, dass dort keine neue Funktionalität implementiert werden darf. |
| Die endgültige Implementierung des Konstruktors lautet:  **public** FootballTeam(**int** gamesWon) {  **if** (gamesWon < 0) {  **throw** **new** IllegalArgumentException(  "Not possible to have less than 0 games won! (was + " + gamesWon + ")");  }  **this**.gamesWon = gamesWon;  } |

### **Vorteile von TDD**

Jetzt, nach dem ersten erfolgreichen TDD-cycle Durchgang kann man schon die Vorteile von TDD erahnen:

* Aller Code ist unit-tested
* Der Code wurde geschrieben um Tests zu erfüllen – es gibt keine überflüssigen Teile des Codes, die nur deswegen geschrieben wurden, weil er vielleicht irgendwann einmal nützlich sein könnte (YAGNI).
* Die kleinste Menge an Code zu schreiben um den Test gelingen zu lassen führt zu einfachen Lösungen (KISS)
* Dank der Refaktorierungsphase ist der Code sauber und lesbar (DRY)
* Es fällt leicht, zum Coding zurückzukehren nach Unterbrechnungen: nimm einfach den nächsten Test und beginne damit den nächsten TDD-cycle.

**Wenn man erst einmal Test First zur Gewohnheit gemacht hat, dann wir man sich unwohl fühlen, Code zu schreiben ohne vorherige failing tests.**

### **Der zweite TDD cycle**

Nun zur Hauptspezifikation, der Vergleichbarkeit von Football Teams

1. **RED** – Schreibe einen Test. Sorge dafür, dass er kompiliert aber nicht mehr. Lasse ihn laufen. Er failt.

|  |
| --- |
| **Spezifikation**: Die SUT soll Comparable sein |
| @Test  **public** **void** shouldBePossibleToCompareTeams() {  FootballTeam team = **new** FootballTeam(***THREE\_GAMES\_WON***);  *assertTrue*("FootballTeam should implement Comparable",team **instanceof** Comparable);  } |
| Letzte Version von FootballTeam |
| **Kommentar:** **Test-fail**, da FootballTeam nicht Comparable impelementiert. uncommon test: Tests mit dem instanceof-Operator sind selten. Hier jedoch legitim, da es von Clients gefordertes Verhalten ist (Collections.sort() ) |

1. **GREEN** – Implementiere die nötigen Funktionalität des SUTs auf einfachste Art. Lasse danach ALLE(!) Tests laufen.

|  |
| --- |
| **Implementierung**: implements Comparable (von der IDE generiert) |
| **public** **class** FootballTeam **implements** Comparable<FootballTeam>  {  **(…)**  @Override  **public** **int** compareTo(FootballTeam o) {  // **TODO** Auto-generated method stub  **return** 0;  }  } |
| **Kommentar:** Jetzt müssen ALLE(!) Tests laufen gelassen werden. Damit stellt man sicher, dass durch neue Funktionalität nicht alte kaputt gemacht wird. Unit Tests laufen schnell, weil sie keine externen Abhängigkeiten haben. |

1. **BLUE** – Refactor

|  |
| --- |
| **Refactoring**: nicht möglich und auch nicht notwendig |
|  |
| **Kommentar**: Weder an der SUT noch am Test kann etwas refaktoriert werden. |

1. **GREEN** – Da nichts refaktoriert wurde, gibt es hier nichts zu testen.

### **Der dritte TDD cycle**

1. **RED** – Schreibe einen Test. Sorge dafür, dass er kompiliert aber nicht mehr. Lasse ihn laufen. Er failt.

|  |
| --- |
| **Spezifikation**: Welches soll der nächste Testfall sein? Wir entscheiden uns für das Verfahren 8.2.1 3 First Typical Case then Corner Cases. Zunächst ein Typical Case: 2 Teams mit verschiedenen gamesWon, wobei jenes Team mit dem verglichen wird eine kleinere Anzahl gamesWon hat. |
| @Test  **public** **void** teamWithMoreMatchesWonShouldBeGreater() {  FootballTeam team\_2 = **new** FootballTeam(2);  FootballTeam team\_3 = **new** FootballTeam(3);  *assertTrue*("team with " + team\_3.getGamesWon() + " games won should be ranked before " +  "team with " + team\_2.getGamesWon() + " games won", team\_3.compareTo(team\_2) > 0);  } |
| **Kommentar:** Die IDE-generierte Implementierung der Methode compareTo() sollte fehlschlagen.  Lasse den Test laufen. Er hat ein fail. Ist die JUnit-Fehlermeldung bei der Ausführung des Tests nicht sprechend genug, schreibe eine bessere: |

1. **GREEN** – Implementiere die nötigen Funktionalität des SUTs auf einfachste Art. Lasse danach ALLE(!) Tests laufen.

|  |
| --- |
| **Implementierung**: nur so, dass dieser Testfall gelingt |
| @Override  **public** **int** compareTo(FootballTeam o) {  **if** (gamesWon > o.getGamesWon()) {  **return** 1;  }  **return** 0;  } |
| **Kommentar**: Sie und ich wissen, wie man das am einfachsten implementiert:  **return** **this**.gamesWon - o.gamesWon;  Aber darum geht es nicht. Diese Implementierung kann als letztes Refactoring ganz am Ende angewendet werden. Hier geht es um den TDD-Ablauf mit einem einfachen Beispiel an dem man ihn nachvollziehen kann.  Jetzt müssen wieder ALLE(!) Tests laufen gelassen werden. Damit stellt man sicher, dass durch neue Funktionalität nicht alte kaputt gemacht wird. Unit Tests laufen schnell, weil sie keine externen Abhängigkeiten haben. |

1. **BLUE** – Refactor

|  |
| --- |
| **Refactoring**: Hier: nur den Bezeichner des Parameters lesbarer machen |
| @Override  **public** **int** compareTo(FootballTeam otherTeam) {  **if** (gamesWon > otherTeam.getGamesWon()) {  **return** 1;  }  **return** 0;  } |

1. **GREEN** – Führe ALLE(!)Tests erneut aus.

### **Der vierte TDD cycle**

1. **RED** – Schreibe einen Test. Sorge dafür, dass er kompiliert aber nicht mehr. Lasse ihn laufen. Er failt.

|  |
| --- |
| **Spezifikation**: Der nächste Testfall: 2 Teams mit verschiedenen gamesWon, wobei jenes Team mit dem verglichen wird eine größere Anzahl gamesWon hat. |
| @Test  **public** **void** teamsWithLessMatchesWonShouldBeLesser() {  FootballTeam team\_2 = **new** FootballTeam(2);  FootballTeam team\_3 = **new** FootballTeam(3);  *assertTrue*("team with " + team\_2.getGamesWon()  + " games won should be ranked after the team with "  + team\_3.getGamesWon() + " games won",  team\_2.compareTo(team\_3) < 0);  } |
| **Kommentar:** Lasse den Test laufen. Er hat ein fail. Ist die JUnit-Fehlermeldung bei der Ausführung des Tests nicht sprechend genug, schreibe eine bessere. |

1. **GREEN** – Implementiere die nötigen Funktionalität des SUTs auf einfachste Art. Lasse danach ALLE(!) Tests laufen.

|  |
| --- |
| **Implementierung**: nur so, dass dieser Testfall gelingt |
| @Override  **public** **int** compareTo(FootballTeam otherTeam) {  **if** (gamesWon > otherTeam.getGamesWon()) {  **return** 1;  }  **else** **if** (gamesWon < otherTeam.getGamesWon()) {  **return** -1;  }  **return** 0;  } |

1. **BLUE** – Refactor: Hier gibt es nichts zu tun.
2. **GREEN** – Da nicht refaktoriert wurde, brauchen die Tests nicht erneut ausgeführt werden.

### **Der fünfte TDD cycle**

1. **RED** – Schreibe einen Test. Sorge dafür, dass er kompiliert aber nicht mehr. Lasse ihn laufen. Er failt.

|  |
| --- |
| **Spezifikation**: Der nächste Testfall: 2 Teams mit gleicher gamesWon |
| @Test  **public** **void** teamsWithSameNumberOfMatchesWonShouldBeEqual() {  FootballTeam teamA = **new** FootballTeam(2);  FootballTeam teamB = **new** FootballTeam(2);  *assertTrue*("both teams have won the same number of games: "  + teamA.getGamesWon() + " vs. " + teamB.getGamesWon()  + " and should be ranked equal",  teamA.compareTo(teamB) == 0);  } |
| **Kommentar:** Ärgerlicherweise gelingt dieser Test! Das ist schlecht, denn wir haben so einen Schritt im TDD-cycle übersprungen. Wir wissen so nicht, warum er gelingt. Liegt es daran, dass die SUT wirklich die gewünschte Funktionalität implementiert? Oder ist der Test falsch geschrieben (und würde z.B immer gelingen)?  In diesem Fall gibt es 2 Möglichkeiten:  a) Wenn man sicher ist, dass das gewünschte Verhalten implementiert ist, dann muss man nur sicherstellen, dass der Test wirklich einmal gelaufen ist (kann in großen Teams mit strenger Arbeitsteilung und umfangreichen Test-Suites schon mal übersehen werden)  b) Wenn man sich nicht sicher ist, bricht den Code um den Test versagen zu sehen, z.B. so: |
| **public** **int** compareTo(FootballTeam otherTeam) {  **if** (gamesWon > otherTeam.getGamesWon()) {  **return** 1;  }  **else** **if** (gamesWon < otherTeam.getGamesWon()) {  **return** -1;  }  **return** 18723;  } |

1. **GREEN** – Implementiere die nötigen Funktionalität des SUTs auf einfachste Art. Lasse danach ALLE(!) Tests laufen.

|  |
| --- |
| **Implementierung**: nur so, dass dieser Testfall gelingt |
| @Override  **public** **int** compareTo(FootballTeam otherTeam) {  **if** (gamesWon > otherTeam.getGamesWon()) {  **return** 1;  }  **else** **if** (gamesWon < otherTeam.getGamesWon()) {  **return** -1;  }  **return** 0;  } |
| **Kommentar**: in diesem Fall muss nur der Fehler-aufzeigende Code wieder rückgängig gemacht werden zur Version davor (RED) |

1. **BLUE** – Refactor: Nun die Fälle geschickt zusammenfassen

|  |
| --- |
| @Override  **public** **int** compareTo(FootballTeam otherTeam) {  **return** **this**.gamesWon - otherTeam.gamesWon;  } |
| **Kommentar**: eine radikale Codeveränderung! (tatsächlich bekannt als die Standard-Implementierung). Wird sie allen Tests standhalten? |

1. **GREEN** – Erneut werden ALLE(!) Tests ausgeführt. Es zeigt sich, dass alle Fälle auch nach diesem radikalen Refactoring noch gelingen. Somit haben wir für die weitere Entwicklung ein Sicherheitsnetz aus Tests auf das wir aufbauen können.

|  |
| --- |
| The actual implementation of the test is rather a detail of TDD. Much more important is  the mindset of practicing baby steps, the mindset of gaining insights and evolving through  rapid feedback, the mindset of leveraging trial & error as a methodology, where errors  are not failures but valuable insights that guide the evolution of the project.  — Jonas Bandi (<http://blog.jonasbandi.net/> ) |

|  |
| --- |
| Every new feature starts with a test. The main objective of this test is to focus  on requirements and code design before writing the code. A test is a form of an  executable documentation and can be used later on to get an understanding of what  the code does or what are the intentions behind it.  (…)  In real-world situations, you wouldn't get such detailed requirements, but dive right  into tests that would act as both requirements and validation.  (…)  By writing or modifying the test first, the developer is focused on requirements before starting to work on a code. This is the main difference when compared to writing tests after the  implementation is done. An additional benefit is that with tests first, we are avoiding the danger that the tests work as quality checking instead of quality assurance.  (…)  Refactoring can be done on any part of the code at any time, as long as all the tests are successful.  The best moment to refactor something is when someone sees an opportunity to make it better. It doesn't matter who wrote it or when; making the code better is always a good thing to do.  — “Test-Driven Java Development” – Packt Publishing 2015 |