Most important things to know about classic TDD

TDD in general

- *Motivation*
  - Never been afraid to touch some mazy legacy code fearing you'd break it?
  - Unit tests are like a manual.
  - *"Any fool can write code that a computer can understand. Good programmers write code that **humans** can understand" (*Martin Fowler)
- Precondition
  - Domain knowledge must be available (p.e. by a domain expert or a domain specific language)
- Some Rules
  - No upfront design assumptions. Design emerges completely from the code, hence it solves over-engineering problems.
  - Tests must be *mutually independent*
  - Never refactor with failing tests (or no tests)
  - Only use what you can control
  - Rule of 3 for duplication cleanup (duplication is cheaper than abstraction in terms of coupling)

TDD approach

- Baby Steps
  - Only write as much of code for a unit test to fail.
  - Only write as much of production code for making a failing unit test pass
- Internal test structure
  - Arrange
  - Act
  - Assert
- The three ways forward in Test Code
  - Red
  - All Green
  - Refactor
- The three ways forward in Production Code
  - Fake implementation
  - Obvious implementation
  - Generalization trough triangulation
- Only one execution path per test
- Naming Test-Classes
  - Test a single Class: [ClassName]Sould vs. [ClassName]Tests
  - Test a feature: [FeatureName]Tests
- Naming Test-Methods
  - [behave]With[Inputs] vs. test[MethodName][expected behaviour]With[Inputs]
  - Beispiel: BankAccountSould.have_the_balance_increased_after_a_deposit vs. BankAccountTests.testDepositIncreasesBalance

  => It dosen't really matter witch naming convention is used, but it is important, that name of the testmethod mirrors the business rule under test.

Transformation priority premises

- Ruleset to avoid overengineering

#	TRANSFORMATION	STARTING CODE	FINAL CODE
1	{} => **nil**		`return nil`
2	nil => **constant**	`return nil`	`return "1"`
3	constant => **constant+**	`return "1"`	`return "1" + "2"`
4	constant => **scalar**	`return "1" + "2"`	`return argument`
5	statement => **statements**	`return argument`	`return arguments`
6	unconditional => **conditional**	`return arguments`	`if(condition)return arguments`
7	scalar => **array**	`dog`	`[dog, cat]`
8	array => **container**	`[dog, cat]`	`{dog = "DOG", cat = "CAT"}`
9	statement => **tail recursion**	`a + b`	`a + recursion`
10	conditional => **loop**	`if(condition)`	`while(condition)`
11	tail recursion => **full recursion**	`a + recursion`	`recursion`
12	expression => **function**	`today - birthday`	`CalculateAge()`
13	variable => **mutation**	`day`	`var day = 10; day = 11;`
14	**switch case**		

- Application of the Rules
  - Rule 1 is sufficient to satisfy the first failing test case
  - If rule n isn't adequate anymore, then apply rule n+1

Object Calisthenics (beauty & strength)

Rule	Why
Only one level of indentation per method	• Focus (single responsibility) • Size
Don’t use the ELSE keyword	• Single execution line • Handle complex cases by polymorphism (Stragegy-Pattern) • Use a Map
Wrap all primitives and strings	• Better readability • Bundle behaviour and data
First class collections (wrap all collections)	• Bundle behaviour and data • Encapsulation • Streaming
No getters/setters/properties	• Bundle behaviour and data
One dot per line	• Readability • Hiding implementation (by not passing attribut values of an Object) • Only talk to friends (Law of Demeter) • Tell, don’t ask moves behavoiur from the calling class to the called class
Don’t abbreviate	• Long names are indicators for missing concepts • Avoid confusion
Keep entities small	• Single responsibility • Complexity (no class over 50 lines, no package over 10 files)
No classes with more than two instance variables	• Low cohesion for actuator classes • High cohesion only for orchestrators classes

Some Code snippets for visualisation

Only one level of indentation per method

```
String board() {
    StringBuffer buf = new StringBuffer();
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++)
            buf.append(data[i][j]);
        buf.append("\n");
    }
    return buf.toString();
}
```

```
String board() {
    StringBuffer buf = new StringBuffer();
    collectRows(buf);
    return buf.toString();
}

void collectRows(StringBuffer buf) {
    for (int i = 0; i < 10; i++)
        collectRow(buf, i);
}

void collectRow(StringBuffer buf, int row) {
    for (int i = 0; i < 10; i++)
        buf.append(data[row][i]);
    buf.append("\n");
}
```

Wrap all primitives and strings

```
public class Discounter {

    private int discount;

    public int applyTo(int initialPrice) {
        return initialPrice - discount;
    }
}
```

```
public class Discounter {

    private Money discount;

    public Money applyTo(Money initialPrice) {
        return initialPrice.minus(discount);
    }

}

public class Money {

    private int discountInCents;
```

No getters/setters/properties

```
q.setQuality(q.getQuality() - 1);
```

```
q.decrease();
```

```
q.quality = 0;
```

```
q.dropToZero();
```

First class collections (wrap all collections)

```
public class Accounts {

    private Map<AccountId, Account> accounts = new HashMap<AccountId, Account>();
```

Source: https://bolcom.github.io/student-dojo/legacy-code/DevelopersAnonymous-ObjectCalisthenics.pdf