Test-Driven Development

Fall 2017

Department of Computer Science Ben-Gurion University

Based on Presentation by
Nurit Gal-oz, Department of Computer Science Ben-Gurion University
Mira Balaban Department of Computer Science Ben-Gurion university
K. Beck: Test-Driven Development by Example
Slides of: Grenning
Motivation

Why do we need XP?

Common problems of software development:
- Schedule slips
- Business misunderstood
- Defect rate
- Management
- Motivation of developers

XP solutions:
- Short iterations, fast delivery
- Whole team
- Test driven development
- Shared understanding
- Humanity and productivity
Some eXtreme Practices

* Is testing good?
  * Then write the **unit test before the code**, **automate testing**, and **run all tests all the time**. (Coding)

* Are code inspections good?
  * Then **code in pairs**—all code is inspected all the time! (Team)

* Is customer contact good?
  * Then **locate a customer representative in the team**, so that you have access to them all the time. (Processes, Customer)

* If design is good
  * refactor all the time

* If integration testing is good,
  * integrate all the time

* If simplicity is good,
  * do the simplest thing that could possibly work

* If short iterations are good,
  * make them really, really short
XP Values, Principles & Practices

- Principles are bridge between Values, which is synthetic and abstract, and Practices, which tell how to actually develop software.

Values
- Communication
- Simplicity
- Feedback
- Courage

Principles
- Humanity
- Improvement
- Quality
- Accepted Responsibility
- ...

Practices
- Planning Game
- Short Release
- Continuous Integration
- Simple Design
- Pair Programming
- Test Driven Development
The 12 Key Practices

- The Planning Game
- Small Releases
- Metaphor
- Simple Design
- Test-Driven Development
- Refactoring
- Pair Programming
- Collective Ownership
- Continuous Integration
- 40-Hour Workweek
- On-site Customer
- Coding Standards
Automated Testing

- “Code that isn’t tested doesn’t work”
- “Code that isn’t regression tested suffers from code rot (breaks eventually)”
- “If it is not automated it is not done!”
- “A unit testing framework enables efficient and effective unit & regression testing
- Programmer Friendly

http://people.cs.aau.dk/~bnielsen/TOVfdp.ddt/renoitkel/08
Test-Driven Development

So, we're suppose to test this BEFORE we build it?
Test-Driven Development

So, we're suppose to test this BEFORE we build it?
K. Beck
Test-Driven Development by Example
Why is TDD important?

• Many projects fail because they lack a good testing methodology.

• The sense of continuous reliability and success gives you a feeling of confidence in your code, which makes programming more fun.
How does it work?

- Have a requirement. Let’s say “create a new random card, as for a card game”.
- Think about what could be a reasonable test to verify compliance with the requirement.
- Write the test *before writing the code*. Of course, the test will fail, and that’s ok.
- Keep things simple.
- Then, write *only* the *minimally necessary code* to make the test pass.
- This is a process of discovery; as you identify possible improvements, refactor your code relentlessly to implement them.
- Build, keep, and frequently run your cumulative collection of tests.
What is Test-Driven Development?

- An iterative technique for developing software.

- As much about **design** as **testing**.

  - Encourages design from a user’s point of view.

  - Encourages testing classes in isolation.

  - Produces loosely-coupled, highly-cohesive systems

  (Grenning)
What is TDD?

- Must be learned and practiced.
  - If it feels natural at first, you’re probably doing it wrong.
- More productive than debug-later programming.
- *It’s an addiction rather than a discipline* – Kent Beck
TDD Mantra

Red / Green / Refactor
TDD Mantra
Red / Green / Refactor

1. **Red** – Write a little test that doesn’t work, and perhaps doesn’t even compile at first.

2. **Green** – Make the test work quickly, committing whatever sins necessary in the process.

3. **Refactor** – Eliminate all of the duplication created in merely getting the test to work.
From Red to Green
TDD Development Cycle

1. Start
2. Write a test for a new capability
3. Run all tests to see them pass
4. Fix compile Errors
5. Run all tests to see them pass
6. Write the code
7. Run the tests and see it fail
8. Refractor as needed

Start
Do the Simplest Thing

- **Assume simplicity**
  - Consider the *simplest thing* that could possibly work.
  - Iterate to the needed solution.

- **When Coding:**
  - Build the simplest possible code that will pass the tests.
  - Refactor the code to have the simplest design possible.
  - Eliminate duplication.
TDD Process

- Once a test passes, it is **re-run** with every change.
- **Broken tests are not tolerated.**
- **Side-effects** defects are detected immediately.
- **Assumptions** are continually checked.
- Automated tests provide a **safety net** that gives you the courage to refactor.

Beck
TDD Process

- What do you test? …

  *everything that could possibly break* - Ron Jefferies

- Don’t test anything that could not possibly break (always a judgment call)

  Example: *Simple* accessors and mutators
TDD Process

1. **Start small or not at all**
   select one small piece of functionality that you know is **needed** and you **understand**.

2. Ask “what set of tests, when passed, will demonstrate the presence of code we are confident fulfills this functionality correctly?”

3. Make a **to-do list**, keep it next to the computer
   1. Lists tests that need to be written
   2. Reminds you of what needs to be done
   3. Keeps you focused
   4. When you finish an item, cross it off
   5. When you think of another test to write, add it to the list
Test Selection

• Which test should you pick from the list?

• Pick a test that will teach you something and that you are confident you can implement.

• Each test should represent one step toward your overall goal.

• How should the running of tests affect one another?

• Not at all – isolated tests.
Assert First

• Where should you start building a system?
  • With the stories you want to be able to tell about the finished system.

• Where should you start writing a bit of functionality?
  • With the tests you want to pass on the finished code.

• Where should you start writing a test?
  • With the asserts that will pass when it is done.
Clean Code that Works

The goal is *clean code that works*: 
Divide and conquer: First make it work then, clean

Test-Driven Development:
1. “that works”, 2. “clean”.

Model Driven Development:
1. “clean”, 2. “that works”.

24
TDD Techniques
TDD Development Cycle

- Start
- Write a test for a new capability
- Write the code
- Run all tests to see them pass
- Fix compile Errors
- Run the tests and see it fail
- Refractor as needed
- Compile
TDD techniques

• The three approaches to making a test work cleanly:

  • Fake It (‘Til You Make It)
    • Return a constant and gradually replace constants with variables until you have the real code.

  • Triangulate
    • A generalization that is motivated by 2 examples or more.

  • Obvious Implementation
TDD techniques: Fake it

- Fake It (‘Til You Make It)
  - Why would you do something that you know you will have to rip out?
  - Because having something running is better than not having something running, especially if you have the tests to prove it.

- Motivation:
  - Psychological
    - Having a green bar feels completely different from having a red bar.
    - When the bar is green, you know where you stand. You can refactor from there with confidence.
  - Scope control
    - Starting with one concrete example and generalizing from there prevents from prematurely confusing yourself with extraneous concerns.
    - You can do a better job of solving the immediate problem because you are focused.
    - When you go to implement the next test case, you can focus on that one, too, knowing that the previous test is guaranteed to work.
TDD techniques: Fake it

- Test:
  
  ```java
  public void test_fib_of_0_is_0 () {
    assertEquals(0, fib(0));
  }
  ```

  TODO List: fib(0) == 0

- Fake Implementation

```java
public int fib(int n) {
    return 0;
}
```
TDD techniques: Triangulate

- Triangulate
- I only use Triangulation when I'm really, really unsure about the correct abstraction for the calculation.
- **Use a sequence of test examples** and **generalize** the solution until you cover just enough test cases to produce the general solution.
TDD techniques: Triangulate

```java
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 1 + 1;
}
...
```

```java
public void testFibonacci() {
    assertEquals(0, fib(0));
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
}
```

TODO List:
- fib(0) == 0
- fib(1) == 1
- fib(2) == 1
TDD techniques: Triangulate

```java
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 1 + 1;
}
```

```java
public void testFibonacci() {
    assertEquals(0, fib(0));
    assertEquals(1, fib(1));
    assertEquals(1, fib(2));
}
```

```java
public void testFibonacci() {
    int cases[][] = {{0,0},{1,1},{2,1}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
```
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 1 + 1;
}

...
TDD techniques: Triangulate

int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return 1 + 1;
}

...  
int fib(int n) {
    if (n == 0) return 0;
    if (n <= 2) return 1;
    return fib(n-1) + 1;
}

int fib(int n) { 
    if (n == 0) return 0; 
    if (n <= 2) return 1; 
    return fib(n-1)+ fib(n-2); 
}

Generalize for n=2

public void testFibonacci() {
    int cases[][] = {{0,0},{1,1},{2,1},{3,2}};
    for (int i= 0; i < cases.length; i++)
        assertEquals(cases[i][1], fib(cases[i][0]));
}
After each cycle we have

1. **Made a list of** tests we knew we needed to have working.
2. **Told a story** with a snippet of code about how we wanted to view one operation.
3. **Made the test compile** with stubs.
4. **Made the test run** by committing horrible sins.
5. **Gradually generalized** the working code, replacing **constants** with variables.
6. **Added items to our to-do list** rather than addressing them all at once.
Motivating story

- **WyCash** – A company that sold bond portfolio management systems in US dollars

- **Requirement**: support multi-currencies.

- **Method used**:
  - Replace the former basic Dollar objects by Create Multi-Currency Money objects.
  - Instead of – revise all existing services.
Requirement – Create a report:

Like this:

<table>
<thead>
<tr>
<th>Instrument</th>
<th>Shares</th>
<th>Price</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>1000</td>
<td>25 USD</td>
<td>25000 USD</td>
</tr>
<tr>
<td>Novartis</td>
<td>400</td>
<td>150 CHF</td>
<td>60000 CHF</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td></td>
<td><strong>65000 USD</strong></td>
</tr>
</tbody>
</table>

Exchange rates:
- **From** CHF  **To** USD  **Rate** 1.5

- $5 + 10 CHF = $10 if rate is 2:1
  - $5 * 2 = $10
The 1st to-do list:

- Add amounts in two different currencies and convert the result given a set of exchange rates.

  \[5 + 10 \text{ CHF} = 10 \text{ if rate is 2:1}\]

- Multiply an amount (price per share) by a number (number of shares) and receive an amount.

  \[5 \times 2 = 10\]
The First test:

- We work on multiplication first, because it looks simpler.
- What object we need
The First test:

- We work on multiplication first, because it looks simpler.
- What object we need – **wrong question**!
- What test we need?

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```
2\textsuperscript{nd} to-do list:

\begin{itemize}
\item $5 + 10$ CHF $= 10$ if rate is 2:1
\item $5 \times 2 = 10$
\item Make "amount" private
\item Dollar side-effects?
\item Money rounding?
\end{itemize}

Test does not compile:

1. No class Dollar.
2. No constructor.
3. No method times(int).
4. No field amount.
Make the test compile:

class Dollar{
    public int amount;
    public Dollar(int amount) {} 
    public void times(int multiplier) {} 
}

Test does not compile:
    1. No class Dollar.
    2. No constructor.
    3. No method times(int).
    4. No field amount.
Test fails.

- Is it bad?
- It’s great, we have something to work on!
- Our programming problem has been transformed from
  
  "give me multi-currency"
  
  to
  
  "make this test work, and then make the rest of the tests work."

- Much simpler.
- Much smaller scope for fear.
- We can make this test work.
Make it run -- Green bar:

The test:

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

The code:

```java
class Dollar{
    public int amount = 10;
    public Dollar(int amount){}
    public void times(int multiplier){}
}
```
The refactor (generalize) step:

- Quickly add a test.
- Run all tests and see the new one fail.
- Make a little change.
- Run all tests and see them all succeed.
- **Refactor** to remove duplication between code and test –
  Duplication $\Rightarrow$ Dependencies between code and test
  Dependencies $\Rightarrow$ test cannot reasonably change independently of the code!
Dependency and Duplication

- **Dependency between the code and the test** occurs when you can't change one without changing the other.
- Goal: write another test that "makes sense" to us, without having to change the code
  - impossible with the current implementation of our example
- If **dependency** is the **problem**, **duplication** is the **symptom**.
- **Duplication** most often takes the form of duplicate logic—the same expression appearing in multiple places in the code.
- **Eliminating duplication in programs** → eliminates dependency.
- **TDD**: by eliminating duplication before we go on to the next test, we maximize our chance of being able to get the next test running with one and only one change.
Remove Duplication

The test:
```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

The code:
```java
class Dollar{
    public int amount = 10;
    public Dollar(int amount){}
    public void times(int multiplier){}
}
```

the duplication is between the data in the test and the data in the code

int amount = \textcolor{red}{5*2}; we did the calculation in our head…
Remove Duplication:

class Dollar{
    public int amount;
    public Dollar(int amount){}
    public void times(int multiplier){
        amount = 5 * 2;
    }
}

Where did the 5 come from? The argument to the constructor:

class Dollar{
    public int amount;
    public Dollar(int amount){
        this.amount = amount
    }
    public void times(int multiplier){
        amount = amount * 2;
    }
}
The test:

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```

The 2 is the value of the multiplier argument passed to times.

class Dollar{
    public int amount;
    public Dollar(int amount){this.amount = amount}
    public void times(int multiplier){
        amount *= multiplier;
    }
}

The test is still green.
2\textsuperscript{nd} to-do list:

$5 + 10 \text{ CHF} = \$10 \text{ if rate is 2:1}$

$5 \times 2 = \$10$

Make "amount" private

Dollar side-effects?

Money rounding?
Our method

• **Write a test:**
  • Think about the operation as a story.
  • **Invent the interface** you wish you had.
  • Include all elements necessary for the **right answers**.

• **Make it run:**
  • Quickly getting that bar to go to **green** dominates everything else.
  • If the clean, simple solution is obvious but it will take you a minute, then make a note of it and
  • get back to the main problem -- getting the bar green in seconds.
  • **Quick green excuses all sins**. But only for a moment.

• **Make it right:**
  • Remove the **duplication** that you have introduced, and
  • get to green quickly.
2\textsuperscript{nd} to-do list:

$5 + 10 \text{ CHF} = $10 \text{ if rate is 2:1}$

$5 \times 2 = $10$

Make "amount" private

Dollar side-effects?

Money rounding?

Next we’ll try to get rid of the side effects.
Degenerate objects

- Write a new test for successive multiplications – that will test side effects:

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
}
```
Degenerate objects

- Write a new test for successive multiplications – that will test side effects:

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

The test fails – red bar!

Decision: Change the interface of Dollar.  
→ Times() will return a new object.
Degenerate objects

- Write another test for successive multiplications:

```java
public void testMultiplication() {
    Dollar five = new Dollar(5);
    Dollar product = five.times(2);
    assertEquals(10, product.amount);
    product = five.times(3);
    assertEquals(15, product.amount);
}
```

The test does not compile!
Degenerate objects:

- Change `Dollar.times`:

  ```java
  public Dollar times(int multiplier) {
      amount *= multiplier;
      return null;
  }
  ```

  The test compiles but does not run!

  ```java
  public Dollar times(int multiplier) {
      return new Dollar(amount * multiplier);
  }
  ```

  The test passes!
Achieved item in the to-do list:

$5 + 10 \text{ CHF} = $10 \text{ if rate is 2:1}$

$5 \times 2 = $10$

Make "amount" private

**Dollar-side-effects?**

Money rounding?

• The “side-effects” item is achieved by turning `Dollar.times()` into a **non-destructive (non-mutator) operation**.

• This is the characterisation of **Value Objects**.
Our method in the last step:

- We have translated a feeling --"disgust at side effects"-- into a test -- multiply a Dollar object twice.

- This is a common theme of TDD: 
  Translate aesthetic judgments into to-do items and into tests.

- Translated a design objection (side effects) into a test case that failed because of the objection.

- Got the code to compile quickly with a stub implementation.

- Made the test work by typing in what seemed to be the right code.
Equality for all – Value objects:

Dollar is now used as a **value object**.

The **Value Object pattern** constraints:

– they are not changed by operations that are applied on them.

– Values of instance variables are not changed.

– All operations must return a new object.

– Value objects must implement `equals()`
  (all $5 objects are the same).
3rd to-do list:

- If Dollar is the key to a hash table, then implementing `equals()` requires also an implementation of `hashCode()` (the equals – hash tables contract).

\[
\begin{align*}
\$5 + 10 \text{ CHF} &= \$10 \quad \text{if rate is } 2:1 \\
\$5 \times 2 &= \$10
\end{align*}
\]

Make "amount" private

Money rounding?

`equals()`

`hashCode()`
Handling `equals()`

Implementing `equals()`  no no!
Handling equals()

Implementing equals()? – no no!

Writing a test for equality!

```java
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

The test fails (bar turns red).

Fake implementation in class Dollar: Just return true.

```java
public boolean equals(Object object) {
    return true;
}
```

Note: true is 5 == 5 which is amount == 5.
Triangulation generalization technique:

- Triangulation is a generalization that is motivated by 2 examples or more.
- Triangulation ignores duplication between test and model code.
- **Technique**: invent another example and extend the test:

**We add $5 \neq $6?**

```java
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
    assertFalse(new Dollar(5).equals(new Dollar(6)));
}
```

**Generalize Dollar.equality:**

```java
public boolean equals(Object object) {
    Dollar dollar = (Dollar) object;
    return amount == dollar.amount;
}
```
Triangulation considerations:

- The difference between "Triangulate" and "Fake It" is shown in the example:
  - we had 2 different values checked in the test, there is no way we can hardcode a single return value to get to green.
  - We need to implement it for real.
- Triangulation means trying some variation in a dimension of the design.
- Leads to abstraction

- Beck:

  *Use triangulation only if you do not know how to refactor using duplication removal.*
Review of handling the 4\textsuperscript{th} to-do list:

- **Noticed** that our design pattern (Value Object) implied an operation.
- **Tested** for that operation.
- **Implemented** it simply.
- **Didn't refactor immediately**, but instead **tested** further.
- **Refactored** to capture the two cases at once.
4th to-do list:

Equality requires also:
- Comparison with null.
- Comparison with other, non Dollar objects.

$5 + 10 \text{ CHF} = $10 \text{ if rate is 2:1}$
$5 \times 2 = $10$

Make "amount" private

**Dollar side-effects?**

Money rounding?

`equals()`

`hashCode()`

Equal null

Equal object
Privacy: 5\textsuperscript{th} to-do list:

Equality of \textit{Dollar} objects enables making the instance variable \textit{amount} private:

\begin{itemize}
  \item $5 + 10 \text{ CHF} = \$10$ if rate is 2:1
  \item $5 \times 2 = \$10$
  \item Make "amount" private
  \item \textit{Dollar side-effects}?
  \item Money rounding?
  \item \textit{equals}()
  \item \textit{hashCode}()
  \item Equal null
  \item Equal object
\end{itemize}
Improve the multiplication test:

- **Turn the integers equality test into Dollar equality test:**

Replace:

```java
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(10, product.amount);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

with

```java
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(new Dollar(10), product);
    product= five.times(3);
    assertEquals(new Dollar(15), product);
}
```
Improve the multiplication test:

- Get rid of the product variable:

  ```java
  public void testMultiplication() {
    Dollar five = new Dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
  }
  ```

  This is a truly **declarative test**.

Now Dollar is the only class using the amount instance variable.

Therefore in the Dollar class:

*Private int amount;*
Achieved the 5th to-do list:

$5 + 10 \text{ CHF} = $10 \text{ if rate is } 2:1$
$5 \times 2 = $10$
Make "amount" private
Dollar side-effects?
Money rounding?
equals()
hashcode()
Equal null
Equal object
Achieved the 5\textsuperscript{th} to-do list:

Note: The multiplication test is now based on equality!

→ dependency among tests.

If the equality test fails, then also multiplication fails!

The operations in the 5\textsuperscript{th} to-do list step:

• Used functionality just developed to improve a test.
• Noticed that if two tests fail at once we're sunk.
• Proceeded in spite of the risk.
• Used new functionality in the object under test to reduce coupling between the tests and the code (removal of the amount instance variable from the test).
The 13th to-do list:

- $5 + 10 \text{ CHF} = $10 if rate is 2:1
- $5 \times 2 = $10
- Delete testFranMultiplication?
- Make "amount" private
- Dollar side-effects?
- Money rounding?
- equals()
  - hashCode()
  - Equal null
  - Equal object
- 5 \text{ CHF} \times 2 = 10 \text{ CHF}
- Dollar/Franc duplication
- Common equals
- Common times
- Compare Frans with Dollars
- Currency?
Current Tests:

public void testMultiplication() {
    Money five = Money.dollar(5);
    assertEquals(Money.dollar(10), five.times(2));
    assertEquals(Money.dollar(15), five.times(3));
}

public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}

public void testCurrency() {
    assertEquals("USD", Money.dollar(1).currency());
    assertEquals("CHF", Money.franc(1).currency());
}
public class Money {
    protected int amount;
    protected String currency;
    public Money(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }
    public Money times(int multiplier) {
        return new Money(amount * multiplier, currency);
    }
    public static Money dollar(int amount) {
        return new Money(amount, "USD");
    }
    public static Money franc(int amount) {
        return new Money(amount, "CHF");
    }
    public boolean equals(Object object) {
        Money money = (Money) object;
        return amount == money.amount && currency().equals(money.currency());
    }
    public String currency() {
        return currency;
    }
    public String toString() {
        return amount + " " + currency;
    }
}
14. Addition – at last: A new to-do list:

- Remove achieved items.
- Handle simple items.

The new list:

$5 + 10 \text{ CHF} = $10 \text{ if rate is } 2:1$

Start with a simpler addition:

$5 + 10 \text{ CHF} = $10 \text{ if rate is } 2:1$

$5 + $5 = $10$
Test for simple addition:

Start with........... A Test:

```java
public void testSimpleAddition() {
    Money sum = Money.dollar(5).plus(Money.dollar(5));
    assertEquals(Money.dollar(10), sum);
}
```

Make the test pass:

1. Fake the implementation: Return "Money.dollar(10)".
2. Obvious implementation.

```java
Money
Money plus(Money addend) {
    return new Money(amount + addend.amount, currency);
}
```

All tests are passed!
Think Again About the Test:

1. We know that we’ll need multi-currency arithmetic.
2. Constraint: System must be unaware that it is dealing with multiple currencies.
3. Possible solution: Convert into a single currency.
   - Rejected. Too rigid.

We’ll try to rewrite the test so that it reflects the context of multi-currency arithmetic:

- Question:
  What is a multi-currency object?
Multi-currency objects:

Example: $(2 + 3\text{CHF}) \times 5$

$5 + 10 \text{CHF} = 10 \text{ if rate is 2:1}$

$5 + 5 = 10$

→ A multi-currency object can be **reduced** to a single currency, if an exchange rate is given.

→ A multi-currency object can be **added**.

→ **Require a multi-currency object Expression.**

  *think about an expression as a… Wallet!*

  *Money* is an atomic expression.

  *Sum* is an expression.

Expressions can be **reduced**, WRT an exchange rate.
Rewrite the test to reflect multi-currencies context:

Replace:

```java
public void testSimpleAddition() {
    Money sum = Money.dollar(5).plus(Money.dollar(5));
    assertEquals(Money.dollar(10), sum);
}
```

With:

```java
public void testSimpleAddition() {
    Money reduced = ... addition of $5 expressions and reduction of the sum ...;
    assertEquals(Money.dollar(10), reduced);
}
```
Who is responsible for exchange of Moneys?

→ The Bank, of course!

Need a reduced Money:

```java
public void testSimpleAddition() {
    ...
    Money reduced = bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

→ Need a bank:

```java
public void testSimpleAddition() {
    ...
    Bank bank = new Bank();
    Money reduced = bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```
A Revised Addition Test:

➔ Need a *sum of Moneys which is an Expression*:

```java
public void testSimpleAddition() {
    ...
    Expression sum = five.plus(five);
    Bank bank = new Bank();
    Money reduced = bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

➔ Need 5 dollars *Money*:

```java
public void testSimpleAddition() {
    Money five = Money.dollar(5);
    Expression sum = five.plus(five);
    Bank bank = new Bank();
    Money reduced = bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```
public void testSimpleAddition() {
    Money five = Money.dollar(5);
    Expression sum = five.plus(five);
    Bank bank = new Bank();
    Money reduced = bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}

Needed:

10 interface Expression {
10   Money:
10      Expression plus(Money addend) {
10          return new Money(amount + addend.amount, currency);
10      }
10  }
10  class Money implements Expression
10  class Bank {
10      Money reduce(Expression source, String to) {
10          return null;
10      }
10  }
public void testSimpleAddition() {
    Money five = Money.dollar(5);
    Expression sum = five.plus(five);
    Bank bank = new Bank();
    Money reduced = bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}

Bank

Money reduce(Expression source, String to) {
    return Money.dollar(10);
}

All tests are passed!
TDD – a break for discussion

- What did we learn so far...?
Summary – 3 basics of TDD:

- The three approaches to making a test work cleanly—fake it, triangulation, and obvious implementation.

- Removing duplication between test and code as a way to drive the design.

- The ability to control the gap between tests to increase traction when the road gets slippery and cruise faster when conditions are clear.