**TDD Sample**

**Display Customer List of items along with Item, No. of Items, price, currency & Total….**

**Convert all the prices & total into single currency… etc..**

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **No. of Items** | **Price** | **Total** |
| **A** | **2** | **5 USD** | **10 USD** |
| **B** | **10** | **10 CHF** | **100 CHF** |
|  |  | **Total** | **60 USD** |

**Exchange Rate**

|  |  |  |
| --- | --- | --- |
| **From** | **To** | **Rate** |
| **CHF** | **USD** | **2** |

**Red:** Write a test, thinking like the operation in your mind to appear in your code

**Green:** make test run. Quickly getting the bar to green dominates everything else. If a clean simple solution is obvious, then type it in, else add the items in to-do list and get back to main problem

**Refactor:** Make it right. Refactor the code, remove the duplications that you have introduced, and get to green quickly.

**TDD Patterns**: Test, Isolated Test,Test List, Test First,Assert First, Test Data, Evident Data

**Red Bar Patterns**: One Step Test, Starter Test, Explanation Test, Learning Test, Another Test, Regression Test, Break, Do over

**Green Bar Patterns:-**

Following are the 3 strategies to quickly getting to green: - (Green Patterns)

1. **Fake It** – Return a constant and gradually replace constants with variables until you have real code.
2. **Obvious Implementation** - Type in the real implementation (only if you are very much sure) otherwise shift to Fake implementation method and later refactor to the right code.
3. **Triangulation** – We only generalize code when we have two examples or more. We briefly ignore the duplication between test and model code. When the second example demands a more general solution, then and only then do we generalize. E.g. first $5 = $5 and second $5 != $6?

@Test  
**public void** testEquality(){  
 *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  
 *assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));  
}

1. Create Maven Project. E.g. tddjavasample or tddmoneysample
2. Understand the requirements and define the tests ( or to do list)
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. $5 \* 2 = $10 (Total amount for 2 items)
   3. Money rounding
   4. Test equality (e.g. USD = USD or USD = CHF)
3. Create test class with the name MoneyTest and add below test method
   1. @Test  
      **public void** testMultiplication(){  
       Dollar five = **new** Dollar(5);  
       five.times(2);  
       assertEquals(10, five.amount);  
      }
   2. Compile it (to make it compile we need to fix below 4 errors)
      1. No Class Dollar
      2. No Constructor
      3. No method times(int)
      4. NO field amount
4. Add Dollar Class
   1. **public class** Dollar {/   
       **int amount**;  
        
       Dollar(**int** amount){  
        
       }  
        
       **void** times(**int** multiplier){  
        
       }  
      }
   2. Bingo! Now we can run the test and watch it fail… (Red Stage)

Expected :10

Actual :0

**Our Programming problem has been transformed from “give me multi-currency” to “make this test work, and then make rest of the tests work.” Which is much simpler. Much smaller scope for fear.. lets make this test work.**

***\*\*\*This is TDD mindset moment. 😊***

1. Let’s do smallest change, which will cause our test to pass
2. Update TO-DO list
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. $5 \* 2 = $10 (Total amount for 2 items)
   3. Money rounding?
   4. Make “amount” field private
   5. Dollar issues/side effects
3. Refactor (need to generalize before we move on. Remove duplication)
   1. Generally, you see duplication between two pieces of code. Issue is dependency between the data in test & data in code class here. Can’t add another test without solving it.
   2. Dependency is the problem, duplication is the symptom (10 in assertEquals & 10 in code “amount field”). Same expression appearing multiple places in the code.

***Note: Objects are excellent for abstracting away the duplication of logic.***

* 1. *Let’s move the setting of the amount from object initialization to the times() method*

**void** times(**int** multiplier){  
 **amount** = 10;  
}

**\*\*\* small steps but in progressive manner as much as possible. 😊**

**Always check but is the right step and the right size of change appropriate depending upon the complexity.**

* 1. *Where can we get a 5?*
     1. *Value passed to Constructor. Lets save it in amount field*
     2. *Replace 5 in times() with amount*
     3. *Replace 2 in times() with multiplier*
  2. *Dollar class after refactoring.*

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

***Summary till now:-***

1. *Made a list of the tests*
2. *Made the test compile with Stubs*
3. *Gradually generalized the working code, replacing constants with variables*
4. *Adding items to to-do list rather than addressing them all at once*

***Phase 2: Degenerate Objects***

**Note:** We got one test to work, but in the process noticed something strange. When we perform an operation on a Dollar, the Dollar changes. If we want to write like below: **Mutation Testing or normal test will not catch this right away…. If you run Mutation Coverage now.. it will show 100% coverage… But after adding below highlighted second assert will make the test fail…**

1. Change the test to test it with different multiplier

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

But if we return Dollar new object from times() method, we can multiple our initial values with new multiplier. For that we need to change the test as well.

@Test  
**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**);  
}

1. Test won’t compile until we change the times()

**// this is Fake implementation. To make code compile quickly with stub implementation**

Dollar times(**int** multiplier){  
 *//amount = amount\* multiplier;* **amount** \*= multiplier;  
 **return null**;  
}

**Now run the test to make it fail….. Red**

1. Make little change to make it pass in times()

//real implantation.. seemed to be the right code

Dollar times(**int** multiplier){  
 **return new** Dollar(**amount** \* multiplier);  
}

1. Update TO-DO list
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
   3. Money rounding
   4. Make “amount” field private
   5. ~~Dollar issues/side effects~~

**Phase 3: Protect Integer member from change after initial value setting by constructor**

**Note: we can achieve4 it by using Value Object design pattern.**

**Value Object Rules:**

1. Instance variable never change once they have been set in the constructor
2. If anyone wants to change the value or perform any operation, they will have to make an entirely new object
3. Value Object should implement equals()
4. Update our to-do list
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
   3. Money rounding?
   4. Make “amount” field private
   5. ~~Dollar issues/side effects~~
   6. equals()
   7. hashCode()

if you use Dollars as the key to a hash table, then you have to implement hashCode()if you implement equals()… will implement this when it’s required

1. Add Test & Run to make it fail

@Test  
**public void** testEquality(){  
 *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));   
}

1. Add equals() fake implementation to make the test pass

@Override  
**public boolean** equals(Object obj) {  
 *//return super.equals(obj);* **return true**;  
}

Add second assert statement to Test & Run to make it fail (Triangulation implementation case because of second assert statement)

@Test  
**public void** testEquality(){  
 *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));

*assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));

}

**Now run the test to make it fail….. Red**

1. Now we need generalize equality with actual implementation of equals()

@Override  
**public boolean** equals(Object obj) {  
 *//return super.equals(obj);* Dollar dollar = (Dollar) obj;  
 **return amount** == dollar.**amount**;  
}

Note: So, equality is done for the moment. But what about comparing with null and comparing with other objects(currency). Let’s add them to our to-do list.

1. Update to-do list
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
   3. Money rounding?
   4. Make “amount” field private
   5. ~~Dollar issues/side effects~~
   6. ~~equals()~~
   7. hashCode()
   8. Equal null
   9. Equal other object/currency

**Summary**: Value object design pattern implied an operation and testing for that operation

**Phase 4: Make “amount” private**

Now as we implemented equals(), we can use it to make our tests more clear speaking.

1. Change the testMultiplication() as below

@Test  
**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);  
}

1. Now temporary “product” variable not helping much, so we can inline it. And with that our test speaks more clearly and we not performing sequence of operations in test now.

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

1. Now with these changes to the test, Dollar is the only class using its “amount” instance variable, so we can make it private

**private int amount**;

1. Update to-do list
2. $5 + 10 CHF = $10 (if rate is 2:1)
3. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
4. Money rounding?
5. ~~Make “amount” field private~~
6. ~~Dollar issues/side effects~~
7. ~~equals()~~
8. hashCode()
9. Equal null
10. Equal other object/currency

**Phase5: Franc(CHK) Implementation:-**

1. We can copy and edit the Dollar test. (Simple & readable test helping us now 😊)

@Test  
**public void** testFrancMultiplication(){  
 Franc five = **new** Franc(5);  
 assertEquals(**new** Franc(10), five.times(2));  
 assertEquals(**new** Franc(15), five.times(3));  
}

**Run the test and make it fail… Red…**

1. Quick step to make test pass is ? yes copy Dollar class and replace Dollar with Franc

***Sins we are doing :*** ☹

* 1. Copy and paste reuse ☹
  2. The death of abstraction ☹
  3. Killing the clean design ☹

**Note**: But remember the TDD cycle & rules 😊.. quickly make it green take importance on anything else. We can make sins to complete first 2 stages(Red & Green)… refactor is the stage where we remove duplicate and write right code

**public class** Franc {  
 **private int amount**;  
  
 Franc(**int** amount){  
 **this**.**amount** = amount;  
 }  
  
 Franc times(**int** multiplier){  
 **return new** Franc(**amount** \* multiplier);  
 }  
  
 @Override  
 **public boolean** equals(Object obj) {  
 *//return super.equals(obj);* Franc franc = (Franc) obj;  
 **return amount** == franc.**amount**;  
 }  
}

**Compile the code and run the test to make it quick green 😊**

1. Update to-do list
2. $5 + 10 CHF = $10 (if rate is 2:1)
3. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
4. Money rounding?
5. ~~Make “amount” field private~~
6. ~~Dollar issues/side effects~~
7. ~~equals()~~
8. hashCode()
9. Equal null
10. Equal other object/currency
11. ~~5 CHF \* 2 = 10 CHF~~
12. Dollar/Franc Duplication
13. Common equals()
14. Common times()
15. **Time to Refactor & remove the duplication….**

**One way is one class extends the other, But that approach hardly saves any code at all.**

**Instead, we are going to find a common superclass for the two classes, which is Money**

**Now let’s work to make equals() common**

1. Create Money class and Dollar & Franc class extends Money
2. Now we can move “amount” field to Money as protected so that sub classes can see it
3. Remove “amount” field from Dollar & Franc classes
4. Now we can change the equals() code to make it ready to move up to Money by changing the declaration & cast of temporary variables. E.g. Dollar to Money
5. Now move equals() to Money and remove it from Dollar class.

**public boolean** equals(Object obj) {  
 *//return super.equals(obj);* Money money = (Money) obj;  
 **return amount** == money.**amount**;  
}

1. Same we need to do for Franc and before that lets add missing test for comparing Franc to Franc in testEquality()

@Test  
**public void** testEquality(){  
 *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  
 *assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));  
 *assertTrue*(**new** Franc(5).equals(**new** Franc(5)));  
 *assertFalse*(**new** Franc(5).equals(**new** Franc(6)));  
}

1. Update to-do list
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
   3. Money rounding?
   4. ~~Make “amount” field private~~
   5. ~~Dollar issues/side effects~~
   6. ~~equals()~~
   7. hashCode()
   8. Equal null
   9. Equal other object/currency
   10. ~~5 CHF \* 2 = 10 CHF~~
   11. Dollar/Franc Duplication
   12. ~~Common equals()~~
   13. Common times()
2. Time to add another test. Comparing Dollar to Franc

@Test  
**public void** testEquality(){  
 *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  
 *assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));  
 *assertTrue*(**new** Franc(5).equals(**new** Franc(5)));  
 *assertFalse*(**new** Franc(5).equals(**new** Franc(6)));  
 *assertFalse*(**new** Franc(5).equals(**new** Dollar(5)));  
}

1. **Compile it & Run the tests to make it fail… Red….**
2. **Lets make quick change to make it pass… green…**

**public boolean** equals(Object obj) {  
 *//return super.equals(obj);* Money money = (Money) obj;  
 **return amount** == money.**amount** && getClass().equals(money.getClass());  
}

**Note: Using classes in Model code is a bit smelly. We should use criteria that make sense in the domain of finance, not in the domain of Java objects. But we don’t currently have anything like a currency, and right now lets add into to-do list**

1. Update to-do list
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
   3. Money rounding?
   4. ~~Make “amount” field private~~
   5. ~~Dollar issues/side effects~~
   6. ~~equals()~~
   7. hashCode()
   8. Equal null
   9. Equal other object/currency
   10. ~~5 CHF \* 2 = 10 CHF~~
   11. Dollar/Franc Duplication
   12. ~~Common equals()~~
   13. Common times()
   14. Currency ?

**Summary:**

1. Made the test run reasonable comparing apples to oranges, but not perfect way – getClass()
2. Decided not to introduce more design until we had a better need or motivation around currency

**Phase 8: Time to get rid of the common times()**

1. It’s basically we need to introduce abstraction here as we did in case of “amount” field by introducing object for Dollar class and removed the reference and need of “amount field in tests.
2. Similarly, Now we would be one step closer to eliminating the subclasses if there were fewer references to the subclasses directly in other places like in test class etc.
3. We can introduce a factory method in Money that returns Dollar

**static** Dollar dollar(**int** amount){  
 **return new** Dollar(amount);  
}

1. Now remove declarations of Dollar class in test

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

1. Our compiler tells us that times() is not defined for Money.
   1. What we can do now to quickly fix it ?
      1. Make Money abstract class
      2. Add times as abstract method
      3. And now we can change the declaration of the factory method in Money class

**static** Money dollar(**int** amount){  
 **return new** Dollar(amount);  
}

* + 1. Change the times() method to return Money in both Dollar & Franc classes

Dollar class:

Money times(**int** multiplier){  
 **return new** Dollar(**amount** \* multiplier);  
}

Franc Class:

Money times(**int** multiplier){  
 **return new** Franc(**amount** \* multiplier);  
}

Compile and run the test to see Tests are in green and everything is working.. 😊

Now no client code knows that there is a subclass called Dollar.

Same Changes we need to make for Franc and test it again.

**Note: if you notice now testFrancMultiplication() not doing anything separate then testMultiplication().. are we confident to delete this test? For now lets leave it until we get good confidence and it to to-do list**…. 😊

//Money Test class after changes….

**import** org.junit.jupiter.api.Test;  
  
**import static** org.junit.jupiter.api.Assertions.*assertEquals*;  
**import static** org.junit.jupiter.api.Assertions.*assertFalse*;  
**import static** org.junit.jupiter.api.Assertions.*assertTrue*;  
  
**public class** MoneyTest {  
  
 @Test  
 **public void** testMultiplication(){  
 Money five = Money.*dollar*(5);  
 *assertEquals*(Money.*dollar*(10), five.times(2));  
 *assertEquals*(Money.*dollar*(15), five.times(3));  
 }  
  
 @Test  
 **public void** testEquality(){  
 *assertTrue*(Money.*dollar*(5).equals(Money.*dollar*(5)));  
 *assertFalse*(Money.*dollar*(5).equals(Money.*dollar*(6)));  
 *assertTrue*(Money.*franc*(5).equals(Money.*franc*(5)));  
 *assertFalse*(Money.*franc*(5).equals(Money.*franc*(6)));  
 *assertFalse*(Money.*franc*(5).equals(Money.*dollar*(5)));  
 }  
  
 @Test  
 **public void** testFrancMultiplication(){  
 Money five = Money.*franc*(5);  
 *assertEquals*(Money.*franc*(10), five.times(2));  
 *assertEquals*(Money.*franc*(15), five.times(3));  
 }  
  
}

// Money Class after changes

**abstract public class** Money {  
 **protected int amount**;  
  
 **abstract** Money times(**int** multiplier);  
  
 **public boolean** equals(Object obj) {  
 *//return super.equals(obj);* Money money = (Money) obj;  
 **return amount** == money.**amount** && getClass().equals(money.getClass());  
 }  
  
 **static** Money dollar(**int** amount){  
 **return new** Dollar(amount);  
 }  
  
 **static** Money franc(**int** amount){  
 **return new** Franc(amount);  
 }  
}

// Dollar Class after changes

**public class** Dollar **extends** Money{  
  
 Dollar(**int** amount){  
 **this**.**amount** = amount;  
 }  
  
 Money times(**int** multiplier){  
 **return new** Dollar(**amount** \* multiplier);  
 }  
}

//Franc class after changes

**public class** Franc **extends** Money{  
  
 Franc(**int** amount){  
 **this**.**amount** = amount;  
 }  
  
 Money times(**int** multiplier){  
 **return new** Franc(**amount** \* multiplier);  
 }  
}

1. **Update to-do list**
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~
   3. Money rounding?
   4. ~~Make “amount” field private~~
   5. ~~Dollar issues/side effects~~
   6. ~~equals()~~
   7. hashCode()
   8. Equal null
   9. Equal other object/currency
   10. ~~5 CHF \* 2 = 10 CHF~~
   11. Dollar/Franc Duplication
   12. ~~Common equals()~~
   13. Common times()
   14. Currency ?
   15. Delete testFrancMultiplication()?

Summary:

1. Took a step toward eliminating duplication by reconciling the signature of two variants of the same method times()
2. Moved at least a declaration of the method to the common superclass
3. Decoupled test code from the existence of concrete subclasses by introducing factory methods
4. We also noticed that when subclasses disappear some tests will be redundant, but for now we are not taking any action on that..

**Phase 9: Introducing Currency:-**

1. Add Currency test

@Test  
**public void** testCurrency(){  
 assertEquals(**"USD"**, Money.*dollar*(1).Currency());  
 assertEquals(**"CHF"**, Money.*franc*(1).Currency());  
}

**Ping-Pong Pairing Steps (TDD Manner)**

Create Maven Project. E.g. tddjavasample or tddmoneysample

1. Understand the requirements and define the tests ( or to do list)
   1. $5 + 10 CHF = $10 (if rate is 2:1)
   2. $5 \* 2 = $10 (Total amount for 2 items)
   3. Money rounding
   4. Test equality (e.g. USD = USD or USD = CHF)

|  |  |  |  |
| --- | --- | --- | --- |
| **Iteration (Stage)** | **Developer 1** | **Developer 2** | **Comments** |
| Red | Create test class with the name MoneyTest and add below test method  @Test **public void** testMultiplication(){  Dollar five = **new** Dollar(5);  five.times(2);  assertEquals(10, five.amount); } |  |  |
| Green |  | Add Dollar Class   1. **public class** Dollar {   **int amount**;   Dollar(**int** amount){   }   **void** times(**int** multiplier){ **amount** = 5\*2;  } } | Compile it (to make it compile we need to fix below 4 errors)   1. No Class Dollar 2. No Constructor 3. No method times(int) 4. NO field amount   Update TODO List with below 2 items:   1. Make “amount” field private 2. Dollar issues/side effects |
| Refactor |  | **public class** Dollar {  **int amount**;   Dollar(**int** amount){  **this**.**amount** = amount;  }   **void** times(**int** multiplier){  *//amount = amount\* multiplier;* **amount** \*= multiplier;  } } | *Where can we get a 5?*   1. *Value passed to Constructor. Lets save it in amount field* 2. *Replace 5 in times() with amount*   *Replace 2 in times() with multiplier* |
| Red |  | Change the test to test it with different multiplier  @Test  public void testMultiplication(){  Dollar five = new Dollar(5);  five.times(2);  assertEquals(10, five.amount);  five.times(3);  assertEquals(15, five.amount);  }  But if we return Dollar new object from times() method, we can multiple our initial values with new multiplier. For that we need to change the test as well.  @Test **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**); } |  |
| Green | Make little change to make it pass in times()  //real implantation.. seemed to be the right code  Dollar times(**int** multiplier){  **return new** Dollar(**amount** \* multiplier); } |  | Update TO-DO list   1. $5 + 10 CHF = $10 (if rate is 2:1) 2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~ 3. Money rounding 4. Make “amount” field private 5. ~~Dollar issues/side effects~~ |
| Refactor |  |  |  |
| Red | Add Test & Run to make it fail  @Test **public void** testEquality(){  *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  } |  | **Phase 3: Protect Integer member from change after initial value setting by constructor**  **Note: we can achieve4 it by using Value Object design pattern.**  **Value Object Rules:**   1. Instance variable never change once they have been set in the constructor 2. If anyone wants to change the value or perform any operation, they will have to make an entirely new object 3. Value Object should implement equals()   Update our to-do list   1. $5 + 10 CHF = $10 (if rate is 2:1) 2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~ 3. Money rounding? 4. Make “amount” field private 5. ~~Dollar issues/side effects~~ 6. equals() 7. hashCode() |
| Green |  | Add equals() fake implementation to make the test pass  @Override **public boolean** equals(Object obj) {  *//return super.equals(obj);* **return true**; } |  |
| Refactor |  |  |  |
| Red |  | Add second assert statement to Test & Run to make it fail (Triangulation implementation case because of second assert statement)  @Test **public void** testEquality(){  *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  *assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));  }  **Now run the test to make it fail….. Red** |  |
| Green | Now we need generalize equality with actual implementation of equals()  @Override **public boolean** equals(Object obj) {  *//return super.equals(obj);* Dollar dollar = (Dollar) obj;  **return amount** == dollar.**amount**; }  Note: So, equality is done for the moment. But what about comparing with null and comparing with other objects(currency). Let’s add them to our to-do list. |  |  |
| Refactor | Now as we implemented equals(), we can use it to make our tests more clear speaking.  Change the testMultiplication() as below  @Test **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); }  Now temporary “product” variable not helping much, so we can inline it. And with that our test speaks more clearly and we not performing sequence of operations in test now.  @Test **public void** testMultiplication(){  Dollar five = **new** Dollar(5);   *assertEquals*(**new** Dollar(10), five.times(2));   *assertEquals*(**new** Dollar(15), five.times(3)); }  Now with these changes to the test, Dollar is the only class using its “amount” instance variable, so we can make it private  **private int amount**; |  | Update to-do list   * 1. $5 + 10 CHF = $10 (if rate is 2:1)   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~   3. Money rounding?   4. ~~Make “amount” field private~~   5. ~~Dollar issues/side effects~~   6. ~~equals()~~   7. hashCode()   8. Equal null   9. Equal other object/currency   **Summary**: Value object design pattern implied an operation and testing for that operation |
| Red | **Phase5: Franc(CHK) Implementation:-**  We can copy and edit the Dollar test. (Simple & readable test helping us now 😊)  @Test **public void** testFrancMultiplication(){  Franc five = **new** Franc(5);  assertEquals(**new** Franc(10), five.times(2));  assertEquals(**new** Franc(15), five.times(3)); }  **Run the test and make it fail… Red…** |  |  |
| Green |  | **public class** Franc {  **private int amount**;   Franc(**int** amount){  **this**.**amount** = amount;  }   Franc times(**int** multiplier){  **return new** Franc(**amount** \* multiplier);  }   @Override  **public boolean** equals(Object obj) {  *//return super.equals(obj);* Franc franc = (Franc) obj;  **return amount** == franc.**amount**;  } }  **Compile the code and run the test to make it quick green** 😊 | Quick step to make test pass is ? yes copy Dollar class and replace Dollar with Franc  ***Sins we are doing :*** ☹   * 1. Copy and paste reuse ☹   2. The death of abstraction ☹   3. Killing the clean design ☹   **Note**: But remember the TDD cycle & rules 😊.. quickly make it green take importance on anything else. We can make sins to complete first 2 stages(Red & Green)… refactor is the stage where we remove duplicate and write right code |
| Refactor |  | **Time to Refactor & remove the duplication….**  **One way is one class extends the other, But that approach hardly saves any code at all.**  **Instead, we are going to find a common superclass for the two classes, which is Money**  **Now let’s work to make equals() common**   1. Create Money class and Dollar & Franc class extends Money 2. Now we can move “amount” field to Money as protected so that sub classes can see it 3. Remove “amount” field from Dollar & Franc classes 4. Now we can change the equals() code to make it ready to move up to Money by changing the declaration & cast of temporary variables. E.g. Dollar to Money 5. Now move equals() to Money and remove it from Dollar class.   **public boolean** equals(Object obj) {  *//return super.equals(obj);* Money money = (Money) obj;  **return amount** == money.**amount**; }   1. Same we need to do for Franc and before that lets add missing test for comparing Franc to Franc in testEquality()   @Test **public void** testEquality(){  *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  *assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));  *assertTrue*(**new** Franc(5).equals(**new** Franc(5)));  *assertFalse*(**new** Franc(5).equals(**new** Franc(6))); } | Update to-do list   * 1. $5 + 10 CHF = $10 (if rate is 2:1)   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~   3. Money rounding?   4. ~~Make “amount” field private~~   5. ~~Dollar issues/side effects~~   6. ~~equals()~~   7. hashCode()   8. Equal null   9. Equal other object/currency   10. ~~5 CHF \* 2 = 10 CHF~~   11. Dollar/Franc Duplication   12. ~~Common equals()~~   13. Common times() |
| Red |  | 1. Time to add another test. Comparing Dollar to Franc   @Test **public void** testEquality(){  *assertTrue*(**new** Dollar(5).equals(**new** Dollar(5)));  *assertFalse*(**new** Dollar(5).equals(**new** Dollar(6)));  *assertTrue*(**new** Franc(5).equals(**new** Franc(5)));  *assertFalse*(**new** Franc(5).equals(**new** Franc(6)));  *assertFalse*(**new** Franc(5).equals(**new** Dollar(5))); }  **Compile it & Run the tests to make it fail… Red….** |  |
| Green | **Lets make quick change to make it pass… green…**  **public boolean** equals(Object obj) {  *//return super.equals(obj);* Money money = (Money) obj;  **return amount** == money.**amount** && getClass().equals(money.getClass()); }  **Note: Using classes in Model code is a bit smelly. We should use criteria that make sense in the domain of finance, not in the domain of Java objects. But we don’t currently have anything like a currency, and right now lets add into to-do list** |  |  |
| Refactor |  |  | Update to-do list   * 1. $5 + 10 CHF = $10 (if rate is 2:1)   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~   3. Money rounding?   4. ~~Make “amount” field private~~   5. ~~Dollar issues/side effects~~   6. ~~equals()~~   7. hashCode()   8. Equal null   9. Equal other object/currency   10. ~~5 CHF \* 2 = 10 CHF~~   11. Dollar/Franc Duplication   12. ~~Common equals()~~   13. Common times()   Currency ?  **Summary:**   1. Made the test run reasonable comparing apples to oranges, but not perfect way – getClass() 2. Decided not to introduce more design until we had a better need or motivation around currency |
| Red | **Phase 8: Time to get rid of the common times()**   1. It’s basically we need to introduce abstraction here as we did in case of “amount” field by introducing object for Dollar class and removed the reference and need of “amount field in tests. 2. Similarly, Now we would be one step closer to eliminating the subclasses if there were fewer references to the subclasses directly in other places like in test class etc. 3. We can introduce a factory method in Money that returns Dollar   **static** Dollar dollar(**int** amount){  **return new** Dollar(amount); }   1. Now remove declarations of Dollar class in test   @Test **public void** testMultiplication(){  Money five = Money.dollar(5);  *assertEquals*(**new** Dollar(10), five.times(2));  *assertEquals*(**new** Dollar(15), five.times(3)); } |  |  |
| Green |  | 1. Our compiler tells us that times() is not defined for Money.    1. What we can do now to quickly fix it ?       1. Make Money abstract class       2. Add times as abstract method       3. And now we can change the declaration of the factory method in Money class   **static** Money dollar(**int** amount){  **return new** Dollar(amount); }   * + 1. Change the times() method to return Money in both Dollar & Franc classes   Dollar class:  Money times(**int** multiplier){  **return new** Dollar(**amount** \* multiplier); }  Franc Class:  Money times(**int** multiplier){  **return new** Franc(**amount** \* multiplier); }  Compile and run the test to see Tests are in green and everything is working.. 😊  Now no client code knows that there is a subclass called Dollar.  Same Changes we need to make for Franc and test it again.  **Note: if you notice now testFrancMultiplication() not doing anything separate then testMultiplication().. are we confident to delete this test? For now lets leave it until we get good confidence and it to to-do list**…. 😊 |  |
| Refactor |  | //Money Test class after changes….  **import** org.junit.jupiter.api.Test;  **import static** org.junit.jupiter.api.Assertions.*assertEquals*; **import static** org.junit.jupiter.api.Assertions.*assertFalse*; **import static** org.junit.jupiter.api.Assertions.*assertTrue*;  **public class** MoneyTest {   @Test  **public void** testMultiplication(){  Money five = Money.*dollar*(5);  *assertEquals*(Money.*dollar*(10), five.times(2));  *assertEquals*(Money.*dollar*(15), five.times(3));  }   @Test  **public void** testEquality(){  *assertTrue*(Money.*dollar*(5).equals(Money.*dollar*(5)));  *assertFalse*(Money.*dollar*(5).equals(Money.*dollar*(6)));  *assertTrue*(Money.*franc*(5).equals(Money.*franc*(5)));  *assertFalse*(Money.*franc*(5).equals(Money.*franc*(6)));  *assertFalse*(Money.*franc*(5).equals(Money.*dollar*(5)));  }   @Test  **public void** testFrancMultiplication(){  Money five = Money.*franc*(5);  *assertEquals*(Money.*franc*(10), five.times(2));  *assertEquals*(Money.*franc*(15), five.times(3));  }  }  // Money Class after changes  **abstract public class** Money {  **protected int amount**;   **abstract** Money times(**int** multiplier);   **public boolean** equals(Object obj) {  *//return super.equals(obj);* Money money = (Money) obj;  **return amount** == money.**amount** && getClass().equals(money.getClass());  }   **static** Money dollar(**int** amount){  **return new** Dollar(amount);  }   **static** Money franc(**int** amount){  **return new** Franc(amount);  } }  // Dollar Class after changes  **public class** Dollar **extends** Money{   Dollar(**int** amount){  **this**.**amount** = amount;  }   Money times(**int** multiplier){  **return new** Dollar(**amount** \* multiplier);  } }  //Franc class after changes  **public class** Franc **extends** Money{   Franc(**int** amount){  **this**.**amount** = amount;  }   Money times(**int** multiplier){  **return new** Franc(**amount** \* multiplier);  } } | **Update to-do list**   * 1. $5 + 10 CHF = $10 (if rate is 2:1)   2. ~~$5 \* 2 = $10 (Total amount for 2 items)~~   3. Money rounding?   4. ~~Make “amount” field private~~   5. ~~Dollar issues/side effects~~   6. ~~equals()~~   7. hashCode()   8. Equal null   9. Equal other object/currency   10. ~~5 CHF \* 2 = 10 CHF~~   11. Dollar/Franc Duplication   12. ~~Common equals()~~   13. Common times()   14. Currency ?   15. Delete testFrancMultiplication()?   Summary:   1. Took a step toward eliminating duplication by reconciling the signature of two variants of the same method times() 2. Moved at least a declaration of the method to the common superclass 3. Decoupled test code from the existence of concrete subclasses by introducing factory methods 4. We also noticed that when subclasses disappear some tests will be redundant, but for now we are not taking any action on that.. |
| Red |  | **Phase 9: Introducing Currency:-**   1. Add Currency test   @Test **public void** testCurrency(){  assertEquals(**"USD"**, Money.*dollar*(1).Currency());  assertEquals(**"CHF"**, Money.*franc*(1).Currency()); } |  |