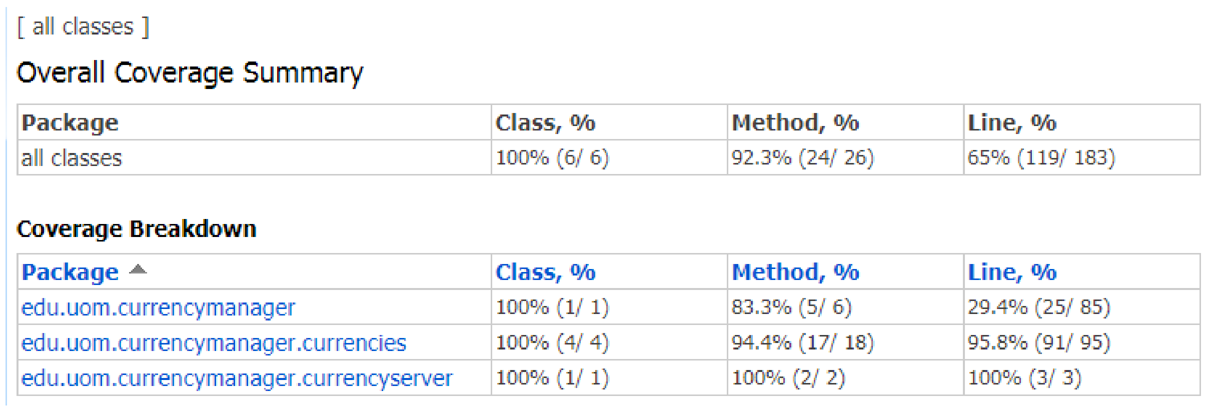
DefaultCurrencyServer has only one method that always returns double and is based on random so there is nothing to test.



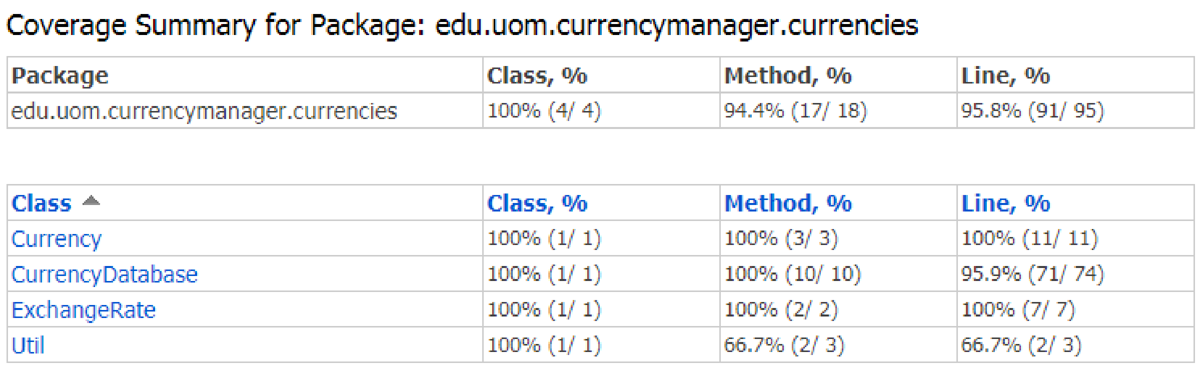
[all classes]

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (6/ 6)	92.3% (24/ 26)	65% (119/ 183)

Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
edu.uom.currencymanager	100% (1/ 1)	83.3% (5/ 6)	29.4% (25/ 85)
edu.uom.currencymanager.currencies	100% (4/ 4)	94.4% (17/ 18)	95.8% (91/ 95)
edu.uom.currencymanager.currencyserver	100% (1/ 1)	100% (2/ 2)	100% (3/ 3)



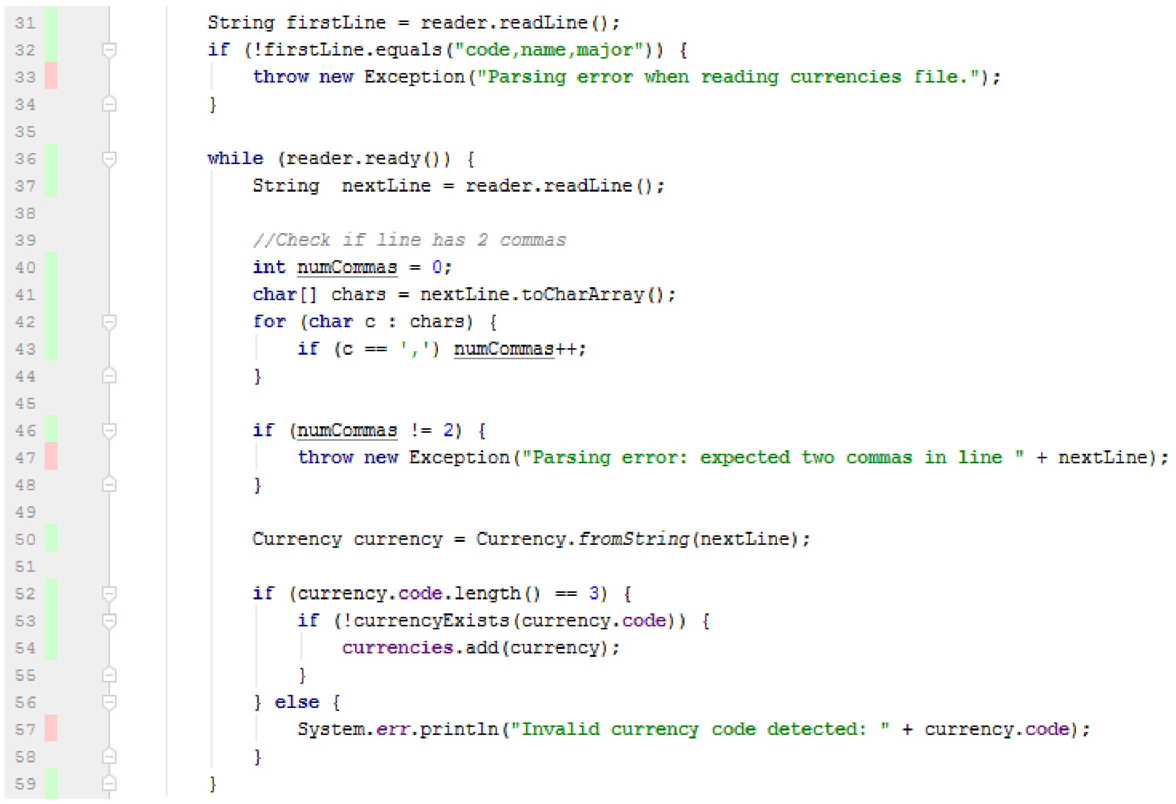
Coverage Summary for Package: edu.uom.currencymanager.currencies

Package	Class, %	Method, %	Line, %
edu.uom.currencymanager.currencies	100% (4/ 4)	94.4% (17/ 18)	95.8% (91/ 95)

Class ▲	Class, %	Method, %	Line, %
Currency	100% (1/ 1)	100% (3/ 3)	100% (11/ 11)
CurrencyDatabase	100% (1/ 1)	100% (10/ 10)	95.9% (71/ 74)
ExchangeRate	100% (1/ 1)	100% (2/ 2)	100% (7/ 7)
Util	100% (1/ 1)	66.7% (2/ 3)	66.7% (2/ 3)

Testing all the methods in *currencies* package leads to high percentage of coverage, but reveals the design vulnerabilities. By implementing OOP concepts in the project, test coverage will be increased to 100%

CurrencyDatabase.java



```
31  String firstLine = reader.readLine();
32  if (!firstLine.equals("code,name,major")) {
33      throw new Exception("Parsing error when reading currencies file.");
34  }
35
36  while (reader.ready()) {
37      String  nextLine = reader.readLine();
38
39      //Check if line has 2 commas
40      int numCommas = 0;
41      char[] chars = nextLine.toCharArray();
42      for (char c : chars) {
43          if (c == ',') numCommas++;
44      }
45
46      if (numCommas != 2) {
47          throw new Exception("Parsing error: expected two commas in line " + nextLine);
48      }
49
50      Currency currency = Currency.fromString(nextLine);
51
52      if (currency.code.length() == 3) {
53          if (!currencyExists(currency.code)) {
54              currencies.add(currency);
55          }
56      } else {
57          System.err.println("Invalid currency code detected: " + currency.code);
58      }
59  }
```

Above 3 lines (#33 #47 #57) couldn't be covered by the tests because a path to currencies.txt is hardcoded within the class and it contains only valid rows.

The solution is to apply the encapsulation – make all the elements that could be private and expose them through the getters and setters.

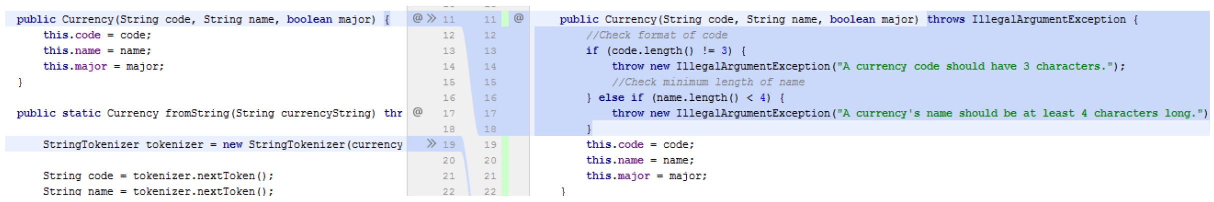
Exception-driven development should be avoided.

CurrencyManager is left with no methods to be tested so the test class is removed.

Refactoring

CurrencyManager as the main user interaction controller shouldn’t be “overloaded” with data processing rules (e.g. if Currency arguments are correct or ExchangeRate already exists in the db …). So the first step is to delegate the control to the components. This leads to cleaner code that is also easier to test. Another benefit is that if the new controller has to be created (e.g. gui or webservice), it doesn’t have to do the checks as they are passed to the lower level. Code duplication is avoided that way too.

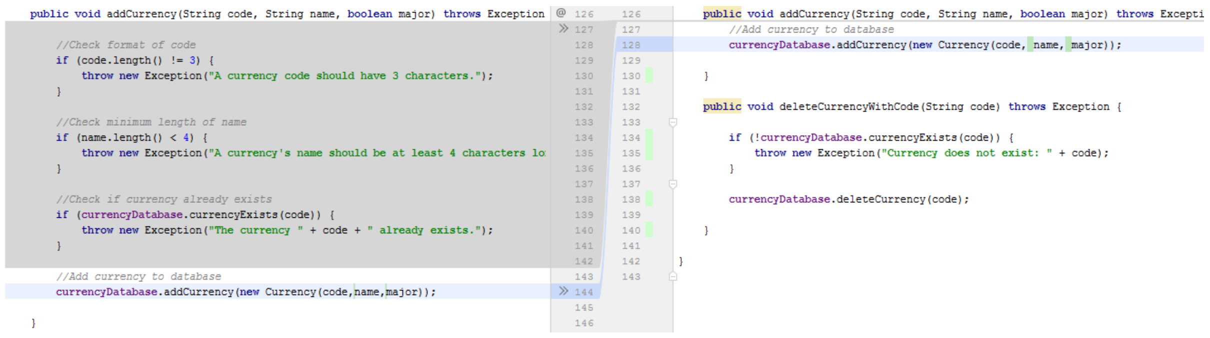
Example of arguments inspection in the Currency constructor – before and after



```java
public Currency(String code, String name, boolean major) {
    this.code = code;
    this.name = name;
    this.major = major;
}

public static Currency fromString(String currencyString) thr

    StringTokenizer tokenizer = new StringTokenizer(currency

    String code = tokenizer.nextToken();
    String name = tokenizer.nextToken();
```

```java
public Currency(String code, String name, boolean major) throws IllegalArgumentException {
    //Check format of code
    if (code.length() != 3) {
        throw new IllegalArgumentException("A currency code should have 3 characters.");
        //Check minimum length of name
    } else if (name.length() < 4) {
        throw new IllegalArgumentException("A currency's name should be at least 4 characters long.")
    }
    this.code = code;
    this.name = name;
    this.major = major;
}
```

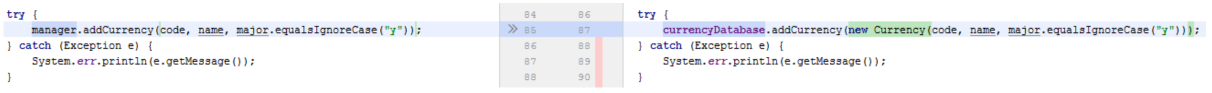
CurrencyManager then becomes lighter for the same checks



```java
public void addCurrency(String code, String name, boolean major) throws Exception

    //Check format of code
    if (code.length() != 3) {
        throw new Exception("A currency code should have 3 characters.");
    }

    //Check minimum length of name
    if (name.length() < 4) {
        throw new Exception("A currency's name should be at least 4 characters lo
    }

    //Check if currency already exists
    if (currencyDatabase.currencyExists(code)) {
        throw new Exception("The currency " + code + " already exists.");
    }

    //Add currency to database
    currencyDatabase.addCurrency(new Currency(code,name,major));

}
```

```java
public void addCurrency(String code, String name, boolean major) throws Excepti
    //Add currency to database
    currencyDatabase.addCurrency(new Currency(code, name, major));

}

public void deleteCurrencyWithCode(String code) throws Exception {

    if (!currencyDatabase.currencyExists(code)) {
        throw new Exception("Currency does not exist: " + code);
    }

    currencyDatabase.deleteCurrency(code);

}

}
```

And since addCurrency method is left with the single line to execute, than it can be removed – instead of calling the method, that line can be executed



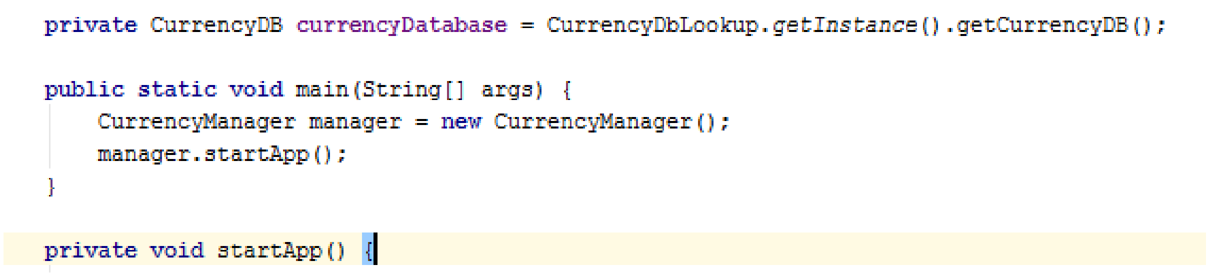
```java
try {
    manager.addCurrency(code, name, major.equalsIgnoreCase("y"));
} catch (Exception e) {
    System.err.println(e.getMessage());
}
```

```java
try {
    currencyDatabase.addCurrency(new Currency(code, name, major.equalsIgnoreCase("y")));
} catch (Exception e) {
    System.err.println(e.getMessage());
}
```

Dependency Injection

Simplest example of Dependency Injection is shown on the example of CurrencyManager. It was responsible to initialize the CurrencyDatabase and then use it. This leads to a hard-coded dependency. If there is a need to switch to some other database access in the future, it would require code changes in CurrencyManager class. This made the application hard to extend and if database was accessed in multiple classes then that would be even harder.

So the solution was to introduce an interface for the database access with all the needed methods.

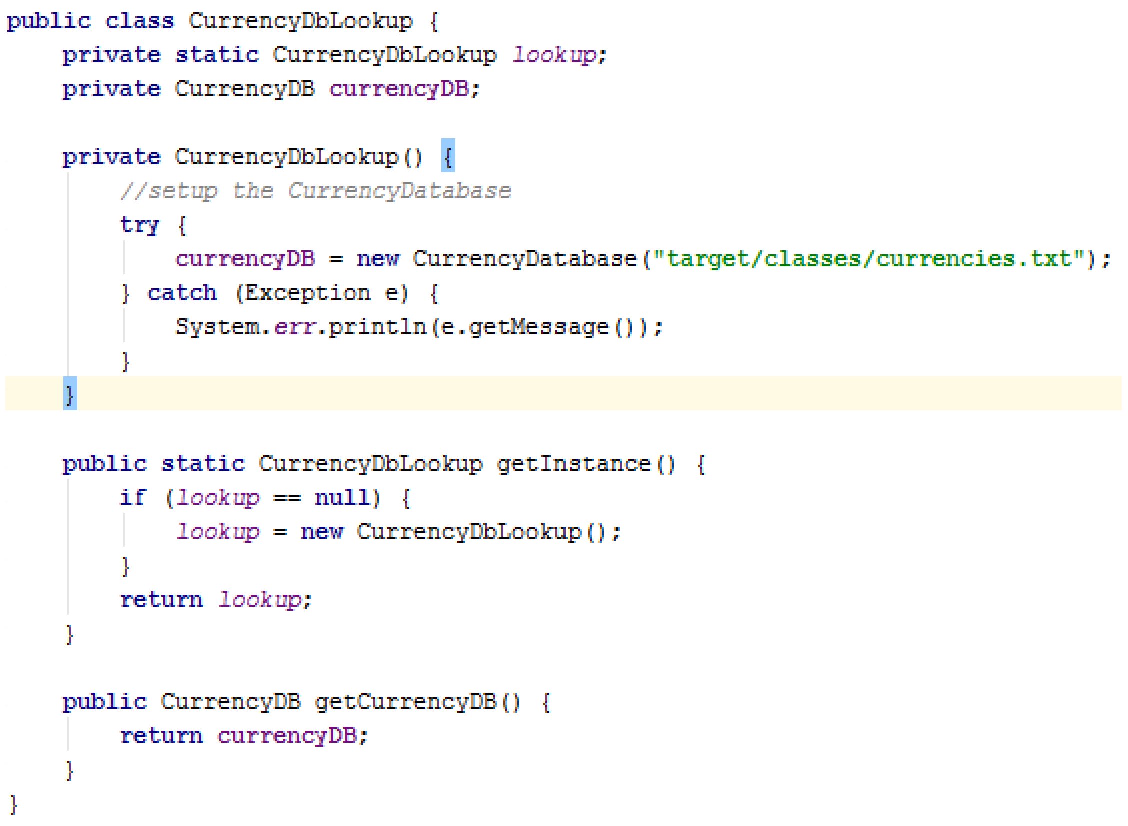


```java
private CurrencyDB currencyDatabase = CurrencyDbLookup.getInstance().getCurrencyDB();

public static void main(String[] args) {
    CurrencyManager manager = new CurrencyManager();
    manager.startApp();
}

private void startApp() {
```

Dependency Lookup

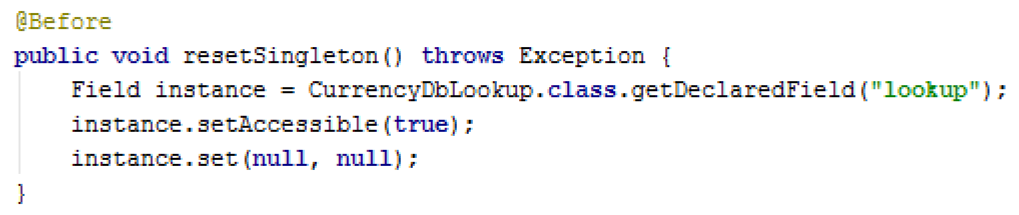
A path to the currencies definition is provided to the constructor so that different implementations/instances could use different data.

CurrencyDbLookup is a singleton that returns the CurrencyDB implementation used by the app.



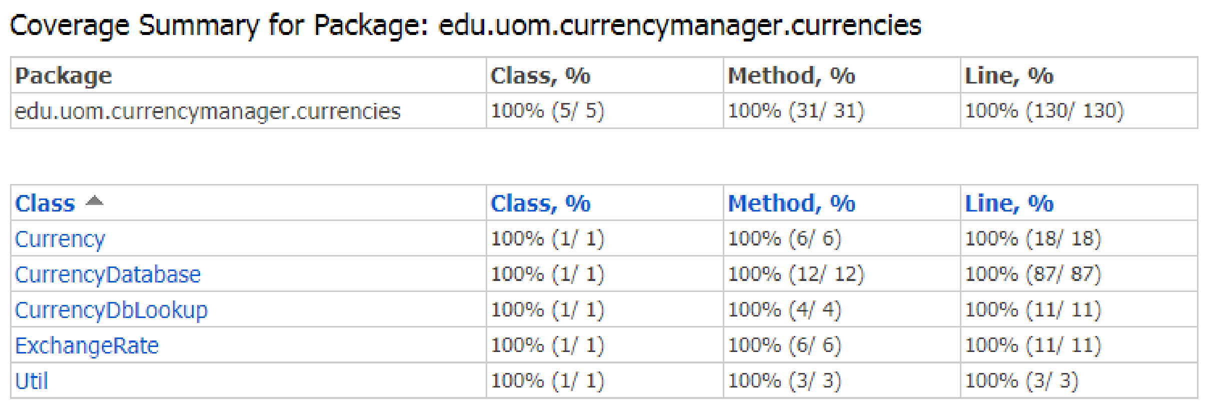
```java
public class CurrencyDbLookup {
    private static CurrencyDbLookup lookup;
    private CurrencyDB currencyDB;

    private CurrencyDbLookup() {
        //setup the CurrencyDatabase
        try {
            currencyDB = new CurrencyDatabase("target/classes/currencies.txt");
        } catch (Exception e) {
            System.err.println(e.getMessage());
        }
    }

    public static CurrencyDbLookup getInstance() {
        if (lookup == null) {
            lookup = new CurrencyDbLookup();
        }
        return lookup;
    }

    public CurrencyDB getCurrencyDB() {
        return currencyDB;
    }
}
```

Reflection is used to reset lookup before each test to prevent tests affecting each other.



```
@Before
public void resetSingleton() throws Exception {
    Field instance = CurrencyDbLookup.class.getDeclaredField("lookup");
    instance.setAccessible(true);
    instance.set(null, null);
}
```

Final coverage



Coverage Summary for Package: edu.uom.currencymanager.currencies

Package	Class, %	Method, %	Line, %
edu.uom.currencymanager.currencies	100% (5/ 5)	100% (31/ 31)	100% (130/ 130)

Class ▲	Class, %	Method, %	Line, %
Currency	100% (1/ 1)	100% (6/ 6)	100% (18/ 18)
CurrencyDatabase	100% (1/ 1)	100% (12/ 12)	100% (87/ 87)
CurrencyDbLookup	100% (1/ 1)	100% (4/ 4)	100% (11/ 11)
ExchangeRate	100% (1/ 1)	100% (6/ 6)	100% (11/ 11)
Util	100% (1/ 1)	100% (3/ 3)	100% (3/ 3)