

JUnit中級

テストビリティとリファクタリング

技術評論社 JUnit実践入門 -体系的に学ぶユニットテストの技法-

第11章 テストダブル

発表者 吉富 楓雅

テストビリティを高める リファクタリング

テスタビリティとは？

- テスタビリティ (testability) : テストのしやすさのこと
- 以下のコードに対するテストは成功したり失敗したりする (Flaky Test)
 - * JavaのDateオブジェクトの精度がミリ秒であるため、テストの実行が十分に早く、new Date() が実行されるタイミングでシステム時間のミリ秒が変わっていなければテストは成功する
- このようなテストは避けるべき

```
4 public class DateDependencyExample {  
5     ^  
6     ^  
7     ^   Date date = new Date();  
8     ^  
9     ^   public void doSomething() {  
10    ^       ^   this.date = new Date();  
11    ^       ^   // 何らかの処理(省略)  
12    ^       ^  
13    ^   }  
14 }
```

ソースコード

```
7 class DateDependencyExampleTest {  
8     ^  
9     ^   @Test  
10    ^   public void doSomethingDateSet() throws Exception {  
11    ^       ^   DateDependencyExample sut = new DateDependencyExample();  
12    ^       ^   sut.doSomething();  
13    ^       ^   // このアサーションは実行タイミングによって成功にも失敗にもなる  
14    ^       ^   assertThat(sut.date, is(new Date()));  
15    ^       ^  
16    ^  
17 }
```

テストコード

不安定なユニットテスト

- 不安定な結果をもたらすユニットテストは避けるべき
 - * テストがしにくいため、テストしやすい設計にする必要がある
- 質の高いユニットテストを書くには、テストビリティの高いクラス設計が必要

Column ユニットテストを前提としたリファクタリング

- 本来はリファクタリングを行う前にユニットテストを作成
 - * プログラムの外部的振る舞いが変わらないという保証を先に作り，内部構造を変更
 - ▶ 大規模なリファクタリングを行う場合にはユニットテストが必要不可欠
- **テストファースト**：プロダクションコードよりも先にテストコードを先に記述
 - * テスト駆動開発を導入すれば，常にテストコードが先にある状態となる
 - * リファクタリングを行い，プロダクションコードを積極的に改善できるようになる

リファクタリング

- Dateオブジェクトの生成が問題
 - * システム時間を取得する部分を1つの独立した処理とみなし、その部分をテストコードから制御できるようにする
- 方法1：テストが難しい処理をメソッドとして抽出し、抽出したメソッドをオーバーライドしてテストする
- 方法2：処理をオブジェクトとして抽出して委譲する

方法1:メソッドの抽出

- 処理をメソッドに抽出する
 - * new Date()をメソッドとして抽出
 - ▶ 抽出したメソッドの名称をnewDateとする
 - ▶ テストコードでオーバーライドできるように, 可視性をデフォルト(パッケージプライベート)
 - ▶ eclipseであればnew Date()を選択してCtrl+1を押すか, 右クリック→リファクタリング→「メソッドの抽出」によって作成できる

抽出メソッドをテストでオーバーライドする

- テストコードではテスト対象のサブクラスを定義し、テストを実行
 - * テスト対象クラスのサブクラスを匿名クラスとして定義し、newDateメソッドをオーバーライド
- オーバーライドされたnewDateメソッドでは、Dateクラスのコンストラクタを利用しない
 - * テストコード内で定義された、予測可能なDateオブジェクト(current)を返す

リファクタリングの結果

```
19 public class MethodExtractExample {  
20     ^   Date date = new Date();  
21     ↓  
22     ^   Date newDate() {  
23     ^       ^   return new Date();  
24     ^   }  
25     ^   ↓  
26     ^   public void doSomething() {  
27     ^       ^   this.date = new Date();  
28     ^       ^   // 何らかの処理(省略)  
29     ^   }  
30 }  
↓
```

メソッドを抽出

```
19 public class MethodExtractExampleTest {  
20     ^   ↓  
21     ^   @Test  
22     ^   public void doSomethingSetCurrentdate() throws Exception {  
23     ^       ^   final Date current = new Date();  
24     ^       ^   MethodExtractExample sut = new MethodExtractExample() {  
25     ^           ^   @Override  
26     ^           ^       Date newDate() {  
27     ^               ^       ^   return current;  
28     ^               ^       }  
29     ^           ^   };  
30     ^           ^   sut.doSomething();  
31     ^           ^   assertThat(sut.date, is(sameInstance(current)));  
32     ^       }  
33 }  
↓
```

方法2:委譲オブジェクトの抽出

- はじめに, 委譲したい処理をnewDateメソッドに抽出
- 次に, この処理(メソッド)をオブジェクトに変換し, テストコードで置き換えられるようにする
 - * DateFactory1クラスを作成し, 抽出できるように処理(メソッド)を移動する
 - * DateFactoryオブジェクトはフィールドに保持されているので, テストコードでDateFactoryクラスのサブクラスを作成し, 差し替えることができる

新しいクラスとDateFactoryクラス

```
1 package junitpractice;↓
2 ↓
3 public class DateFactory1 implements DateFactory{↓
4   ^ public Date newDate() {↓
5   ^   ^   return new Date();↓
6   ^ }↓
7 }↓
```

```
1 package junitpractice;↓
2 ↓
3 public class DelegateObjectExample {↓
4   ^ ↓
5   ^ DateFactory dateFactory = new DateFactory1();↓
6   ^ Date date = new Date();↓
7   ^ ↓
8   ^ public void doSomething() {↓
9   ^   ^   this.date = dateFactory.newDate();↓
10  ^   ^   // 何らかの処理(省略)↓
11  ^ }↓
12 }↓
```

インタフェースを使って処理を差し替えたユニットテスト

```
1 package junitpractice;
2 ↓
3 import org.junit.Test;
4 ↓
5 public class DelegateObjectExampleTest {
6 ↓
7     @Test
8     public void doSomethingSetCurrentdate() throws Exception {
9         ↓
10        final Date current = new Date();
11        DelegateObjectExample sut = new DelegateObjectExample();
12        sut.dateFactory = new DateFactory() {
13            @Override
14            public Date newDate() {
15                return current;
16            }
17        };
18    }
19 ↓
20 }
```

委譲オブジェクトをテストで置き換える

- DateFactoryクラスからインタフェースを抽出し、ポリモフィズムを利用した設計にすることもできる
 - * ポリモフィズム：多態性. Javaでは、同一インタフェースを異なるクラスがインプリメントすることで、同じ型の同じ名前のメソッドに、それらの異なるクラスごとに異なる振る舞いを実装できること
- DateFactoryインタフェースを定義、デフォルトの実装を用意
 - * DateFactoryインタフェースとデフォルト実装を使い書き換え
- テストコードではテスト対象となるオブジェクトのDateFactoryフィールドを差し替えてテストする

DateFactoryクラスのインタフェース化

```
1  package junitpractice;↓
2  ↓
3  public interface DateFactory {↓
4  ^    Date newDate();↓
5  }↓
```

DateFactoryクラスのデフォルト実装

```
1  package junitpractice;↓
2  ↓
3  public class DateFactoryImpl implements DateFactory{↓
4  ^    @Override↓
5  ^    public Date newDate() {↓
6  ^        ^    return new Date();↓
7  ^    }↓
8  }↓
```

インタフェースを使ってシステム時間の処理を委譲

```
1  package junitpractice;↓
2  ↓
3  public class DelegateObjectExample {↓
4  ^    ↓
5  ^    DateFactory dateFactory = new DateFactoryImpl();
6  ^    Date date = new Date();↓
7  ^    ↓
8  ^    public void doSomething() {↓
9  ^    ^    this.date = dateFactory.newDate();↓
10 ^    ^    // 何らかの処理(省略)↓
11 ^    }↓
12 }↓
```

委譲オブジェクトへ抽出することのメリット・デメリット



- 処理を委譲オブジェクトへ抽出することで、テストバリエーションの高い設計に改善できる
 - * クラスに抽出することで、そのクラスは独立した責務を持つため、個別にテストを行うこともできる
 - ▶ インタフェースを利用した設計とするならば、実装クラスが提供されていなくても仮の実装クラスでユニットテストを書くこともできる



- インタフェースとして抽出することは、過剰設計となる可能性がある
 - * 過剰設計もまたテストバリエーションを低下させる
 - * テストバリエーションを高くする目的であれば、多くの場合で、オーバーライド可能なメソッドとして抽出したり、処理を委譲オブジェクトに抽出するだけで十分な効果がある