

JUnit中級

テストダブル

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

クラスやメソッド同士の依存

- テスト対象となるクラスやメソッドが、ほかのクラスに依存していないケースはほとんどない
 - * 依存しているクラスもまた他のクラスに依存している
 - ▶ ユニットテストでは、テスト対象クラスに加え、依存しているクラスも含めてテストしている
-  • ソフトウェアの実行時に近い状態でテストが実行される
 - * クラス単体の動作を保証するよりも価値が高い

依存関係を含めてテストすることのデメリット



- テスト対象クラス以外の問題を原因として、ユニットテストが失敗する可能性がある
 - * テストが失敗した時には、最初にテスト対象クラスに問題があると考えするため、原因の分析に余計な時間がかかる
 - * 依存する機能が外部サービスや乱数、システム時間など、期待する結果を予測できない場合には検証の精度を落とさざるを得ない

テストダブル

- 依存する機能がユニットテストで扱いづらい場合や、ユニットテストの独立性を高めたい場合には、依存するオブジェクトを「**代役**」で置き換えるのが有効
 - * 代役のオブジェクト：**スタブ・モック**
 - ▶ 総称して「**テストダブル(test double)**」と呼ぶ
- **JUnitではモックやスタブの機能を提供していない**
 - * 通常は外部のライブラリを利用(例：Mockito)

スタブ

依存オブジェクトに予測可能な振る舞いをさせる

スタブ

- 依存するクラスやモジュールの代用として使用する仮のクラスやモジュール
 - * 依存オブジェクトに予測可能な振る舞いをさせることが目的
- 使用される状況
 - * 依存オブジェクトが予測できない振る舞いをする
 - * 依存オブジェクトのクラスがまだ存在しない
 - * 依存オブジェクトの実行コストが高く、簡単に利用できない
 - * 依存オブジェクトが実行環境に強く依存している

固定値を返すスタブ

- 例：ランダムな整数を返す機能
- `java.util.Random`クラスの`nextInt`メソッドは、ランダムな整数を生成するJavaの標準的なメソッド
 - * しかし`nextInt`メソッドはユニットテストでは扱いにくい
- 今回は同様に機能を持つインタフェースを定義し、`Random`クラスを利用するデフォルト実装を定義する

乱数生成のインタフェースとデフォルト実装

```
1 package junitpractice2;↓
2 ↓
3 public interface RandomNumberGenerator {↓
4 ^   int nextInt();↓
5 }↓
```

乱数生成のインタフェース

```
1 package junitpractice2;↓
2 ↓
3 import java.util.Random;↓
4 ↓
5 public class RandomNumberGeneratorImpl implements RandomNumberGenerator{
6 ^   private final Random rand = new Random();↓
7 ^   ↓
8 ^   @Override↓
9 ^   public int nextInt() {↓
10 ^   ^   return rand.nextInt();↓
11 ^   }↓
12 }↓
```

乱数生成のデフォルト実装

スタブ実装

- 固定値を返すRandomNumberGeneratorインターフェースのスタブ実装
 - * インタフェースを定義することで、ユニットテストではスタブ実装を作成して利用できる
 - ▶ スタブ実装であれば不確実性を排除できる

```
1 package junitpractice2;↓
2 ↓
3 public class RandomNumberGeneratorFixedResultSub implements RandomNumberGenerator{
4     ^ @Override↓
5     ^ public int nextInt() {↓
6     ^     ^ return 1;↓
7     ^ }↓
8 }↓
```

テスト対象クラス

```
1 package junitpractice2;↓
2 ↓
3 import java.util.List;↓
4 ↓
5 public class Randoms {↓
6 ^ RandomNumberGenerator generator = new RandomNumberGeneratorImpl();
7 ^ ↓
8 ^ public <T> T choice(List<T> options) {↓
9 ^ ^ if (options.size() == 0) return null;↓
10 ^ ^ int idx = generator.nextInt() % options.size();↓
11 ^ ^ return options.get(idx);↓
12 ^ }↓
13 }↓
```

ユニットテスト

```
13 @Test
14 public void choiceAReturn() throws Exception {
15     List<String> options = new ArrayList<String>();
16     options.add("A");
17     options.add("B");
18     Randoms sut = new Randoms();
19     // スタブの設定
20     sut.generator = new RandomNumberGenerator() {
21         @Override
22         public int nextInt() {
23             return 0;
24         }
25     };
26     assertThat(sut.choice(options), is("A"));
27 }
28
29 @Test
30 public void choiceBReturn() throws Exception {
31     List<String> options = new ArrayList<String>();
32     options.add("A");
33     options.add("B");
34     Randoms sut = new Randoms();
35     // スタブの設定
36     sut.generator = new RandomNumberGenerator() {
37         @Override
38         public int nextInt() {
39             return 1;
40         }
41     };
42     assertThat(sut.choice(options), is("B"));
43 }
44
45 }
```

汎用性のあるスタブ実装



- スタブ実装はテストに応じて自由に作ることができる

- * 各テストケースで無名クラスのオブジェクトを作成することも、汎用的に作成して再利用できるクラスを作成することもできる



- **ただし汎用的すぎるクラスはテストコードの可読性を落とす原因になる**

再利用可能なスタブ実装

```
1  package junitpractice2;↓
2  ↓
3  public class RandomNumberGeneratorStub implements RandomNumberGenerator{↓
4  ^   private final int num;↓
5  ^   ↓
6  ^   public RandomNumberGeneratorStub(int num) {↓
7  ^   ^   this.num = num;↓
8  ^   }↓
9  ^   ↓
10 ^   @Override↓
11 ^   public int nextInt() {↓
12 ^   ^   return num;↓
13 ^   }↓
14 }↓
```

例外を送出するスタブ



- オブジェクトによっては，例外が発生する条件が非常に複雑であったり， 外的要因に依存したりする

* 例外ハンドリング(処理)に関するユニットテストは簡単にはできない



- スタブオブジェクトで無条件に例外を送出するように実装すれば，簡単に例外ハンドリングのテストを行うことができる

例外ハンドリングのテスト

```
1 package junitpractice2;↓
2 ↓
3 public interface UserDao {↓
4 ^   User find(String userId) throws UserNotFoundException;↓
5 }↓
```

RDBからユーザ情報を取得するインタフェース

```
1 package junitpractice2;↓
2 ↓
3 public class UserDaoStub implements UserDao{↓
4 ^   ↓
5 ^   @Override↓
6 ^   public User find(String userId) throws UserNotFoundException {
7 ^       throw new UserNotFoundException();↓
8 ^   }↓
9 }↓
```

UserDaoの例外を送出するスタブ

モック

依存オブジェクトの呼び出しを検証する

モック

- 依存クラスやモジュールが正しく利用されているかを検証する
 - * ユニットテストでモックを使う場合、テストコードで依存クラスのメソッドが正しく利用されているかを検証できるようにモックオブジェクトを用意する

モックの定義

```
12 public class RandomsMockTest {  
13     ^  
14     ^ @Test  
15     ^ public void choiceReturnA() throws Exception {  
16     ^     ^ List<String> options = new ArrayList<String>();  
17     ^     ^ options.add("A");  
18     ^     ^ options.add("B");  
19     ^     ^ Randoms sut = new Randoms();  
20     ^     ^ // モックの設定  
21     ^     ^ final AtomicBoolean isCallNextIntMethod = new AtomicBoolean(false);  
22     ^     ^ sut.generator = new RandomNumberGenerator() {  
23     ^     ^     ^ @Override  
24     ^     ^     ^ public int nextInt() {  
25     ^     ^     ^     ^ isCallNextIntMethod.set(true);  
26     ^     ^     ^     ^ return 0;  
27     ^     ^     ^     ^ }  
28     ^     ^     ^ };  
29     ^     ^ assertThat(sut.choice(options), is("A"));  
30     ^     ^ assertThat(isCallNextIntMethod.get(), is(true));  
31     ^     ^ }  
32 }  
    ^
```

サブクラスを定義, 検証対象となる
nextIntメソッドをオーバーライドする

呼び出されたかどうかを保持する変数を利用
実行された場合にtrueを設定

nextIntメソッドが呼び出されたことを検証

モックとスタブの違い

- モックとスタブはよく似たテクニック
 - * スタブ：仮の実装で予測可能な値を返すことで、予測が難しいテストケースを検証可能なテストケースにする
 - * モック：目的のメソッドがテストの実行中に呼び出されたかを検証する
- テストケースによっては両方の目的で利用されることもある
 - * モックなのかスタブなのか曖昧なことも…
- モックは基本的にテストでしか利用しないが、スタブはテストコード以外でも仮実装として利用されることがある
 - * モックオブジェクトの作成は手作業で行うと非常に面倒であるため、通常はMockitoなどのライブラリを使用する