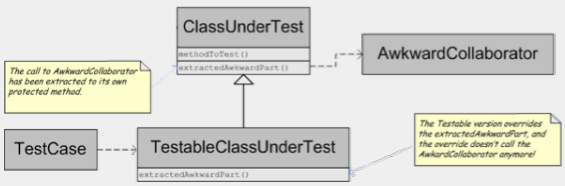
| | |
|---|---|
| **Fakes - Auto-Mock** | Auto-Mocks sind gleich wie normale Mocks, werden jedoch “automagically” vom Computer generiert. Es gibt 2 Arten: Statische Mock Klassen, diese werden mit unserem Code erstellt und kompiliert. Dynamische Mock Klassen hingegen werden während der runtime erstellt werden. Dynamische Mocks sind heute sehr populär. |
| **Fakes - Listener** | Ein Stub der unserer getesteten Klasse zuhört und sich erinnert was gesagt wurde. |
| **Fakes - Mock** | Ein Fake der aktive kontrolliert dass die getestete Klasse sich korrekt verhält. Asserts verifizieren das Verhalten der Klasse und nicht den Status.<br><br>**Vorteile von Mocks:**<br>Mit Mocks kann man sicherstellen dass ein Objekt korrekt mit anderen Objekten interagiert, und zwar so wie wir es erwarten. Mit normalen Test können wir jeweils nur den Status überprüfen mit Mocks hingegen sehen wir das Verhalten. Mocks können vom Computer generiert (z.B. mit Hilfe von JMock, Mockito) oder selbst geschrieben werden. |
| **Fakes - Simulator** | Ein Fake der schwer ist (viel Funktionalität beinhaltet) und so in komplexen Dialogen mit der getesteten Klasse engagiert werden kann. Siehe komplexer Flugsimulator mit Cockpit etc. Ein Simulator erscheint wie “the real thing”.<br><br>**Ein Simulator ist ein sehr ausführlicher Fake mit folgenden Eigenschaften:**<br>• **Lifecycle:** Man kann einen Simulator instanzieren und sehr viele Tests damit durchführen<br>• **Extended Collaboration:** Ein komplexes Protokoll das duzende von Simulator-Aufrufen involviert kann problemlos geschrieben werden<br>• **Once and for all:** Einmal geschrieben und debugged simuliert ein Simulator komplett ein reales Objekt. D.h. man hat das Problem für alle Collaborators die den Simulator verwenden gelöst. |
| **Fakes - Stub** | Einfachste Art des Fakes, sehr leicht-gewichtig. Bringt uns durch den Test aber tut sonst nichts. |
| **Fakes - Talker** | Ein Stub der unserer getesteten Klasse vor-definierte Antworten gibt. |
| **Slipping** | “Slipping” nennt man es wenn seinem Code gefälschte Objekte “unterjubelt” (übergibt). |
| **Slipping Patterns** | • **Inversion of Control:** Anstatt dass Klasse A Klasse B kontrolliert, kehren wir die Beziehung um, Klasse B kontrolliert jetzt Klasse A.<br>• **Dependency Injection:** Anstatt das wir provider = new CustomProvider(); machen, schreiben wir neu: provider = Provider.getProvider();. Dadurch ist der Provider in der Kontrolle der Provider Implementation.<br>• **Slipping a Fake:** Gleiche Technik wie Dependency Injection aber unterschiedliches Motiv. alt: provider = new CustomProvider(); neu: provider = getProvider(); Wir verwenden hier eine statische Methode so dass wir diese extrahieren und überschreiben können. |
| **Slipping Techniken** | • **Argument Slip:** Die getestete Method akzeptiert unser Fake Object als Argument.<br>• **Parameterize Method:** Wenn die getestete Methode den Collaborator instanziert oder eine globale Variable verwendet, extrahieren wir sie in eine parametrierte Methode.<br>• **Constructor Arguement Slip:** Wir übergeben dem Konstruktor der Klasse ein fake collaborator anstelle eines echten, awkward collaborators.<br>• **Alternate Constructor:** Wenn der Konstruktor für die Zusammenarbeit zuständig ist bauen wir einen alternativen Konstruktor der ein fake übergibt. Dies nennt man auch “Overloading”.<br>• **Passing a factory:** Wen die Klasse mehrere awkward collaborators hat sollten wir der Klasse eine fake factory übergeben die dann alle awkward collaborator fakes erstellt.<br>• **Introduce Adapter:** Manchmal darf oder kann man den Code des awkward collaborators nicht direkt berühren weil er eine third party library ist oder speziell alt. In solchen Fälle kann man einen “Introduce Adapter” verwenden. Dieser Adapter stecken wir zwischen unseren awkward Collaborator und den zu testenden Code. |
| **Slipping Techniken - Extract & Override Pattern** | Den Problematischen Teil in eine separate Methode auslagern. Dann diese Methode in einer abgeleiteten Klasse überschreiben, so dass wir ihr genau den Body geben können der gebraucht wird (kann leer sein). Zum Schluss testen wir gegen diese abgeleitete Klasse.<br>**Wichtig:** Man soll nur gerade soviel Code wegnehmen dass “awkward” wegfällt. Wenn man zuviel wegnimmt wird der Test unrealistisch!<br><br>**Was sind die Nachteile des E&O Pattern?**<br>• Die Tests sind mit dem Code gekoppelt.<br>• Die testbare Klasse kann nicht effektiv wiederverwendet werden.<br>• Wenn man dies mehrmals verwendet werden Tests schwer zu lesen.<br><br>**E&O Beispiel-Ablauf**<br>1. Pain-Quelle identifizieren<br>2. Pain-Quellen-Methode extrahieren<br>3. Klasse in Test umbenennen und generieren lassen<br>4. Extrahierte Methoden in Test-Klasse überschreiben mit z.B. statischen Rückgabewerten<br><br>Man kann damit 100% alle awkward MT Probleme in Java und den meisten Sprachen lösen. Dass heisst aber nicht das E&O es immer das beste Werkzeug ist<br>  |

**Singleton (Faking & Mocking)**

**Möglichkeiten mit Singletons umzugehen**
- **Extract & Override:** referenziert zum awkward collaborator
- **Alternate Construction:** des Problem-Objekts als fake object.
- **Set Fake Singleton:** anstelle des awkward singleton
- **Redirection:** der Aufrufe ins real oder fake object

**Awkward Testing - Gründe**

- Code der von Zeit abhängig ist.
- Code der von zufälligen Zahlen abhängig ist.
- Code der von kaputten Collaborators abhängig ist.
- Awkward Inclusion Pyramid - Ein Modul ist abhängig von vielen andern, d.h. in Wirklichkeit werden Integrations-Tests durchgeführt und gar nicht micro-tests.
- Awkward Access - z.B. Drucker/Netzwerk/FileSystem können nicht vollständig kontrolliert werden.

**Awkward Collaborations**

- **Awkward Setup**
  Ugly tests, too much setup, no overview
- **Awkward Results**
  Indeterminacy due to time, random or other, uncontrollable module dependency
- **Awkward Runtime**
  Slow Tests = No Tests
- **Awkward Inclusion**
  collaborator has a high 'fan-out', lots of its own collaborators
- **Awkward Access**
  Awkward access to input/output (filesystem, a global variable, the network, or the printer)

**Mocks**

```
class Another…
   public ResultCode doSomething(Parameter p) {
      // blah-blah-blah, including using 'p' and setting 'result'.
      return result;
   }

class FakeAnother extends Another…
   ArrayList doSomethingParameters = new ArrayList();
   ArrayList doSomethingResults = new ArrayList();
   int doSomethingIndex = 0;

   public ResultCode doSomething(Parameter p) {
      doSomethingParameters.add(p);
      return doSomethingResults.get(doSomethingIndex++);
   }
```

```
public class OneThingTest…
   FakeAnother another = new FakeAnother();

   @Test
   public void doSomethingDisconnected() {
      another.doSomethingResults.add(new ResultCode("DISCONNECTED"));
      oneThing.use(another);
      assertEquals(1, another.doSomethingParameters.size());
      assertEquals(new Parameter("The Right One"), another.doSomethingParameters.get(0));
   }
```

**Fake Factory**

```
public interface BorderFactory {
   public RemoteSource getRemoteSource();
   public RemoteLog getRemoteLog();
   public CardProcessor getCardProcessor();
   public Database getDatabase();
   public Firmware getFirmware();
}
```

The production version (not shown here) creates the awkward collaborators. Our fake implements this interface :

```
public class BorderFactoryFake implements BorderFactory…
   public CardProcessor cardProcessor;

   public CardProcessor getCardProcessor() {
      return cardProcessor;
   }
```

As usual, we use the fake in the test:

```
public class SwipeStepTest …
   @Test
   public void expiry() {
      BorderFactoryFake factory = new BorderFactoryFake();
      factory.cardProcessor = new CardProcessorFake();
      SwipeStep step = new SwipeStep(factory);
      step.activate();
      assertEquals("BadSwipe", step.next);
   }
```