

# ロールをモックせよ

## Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes ThoughtWorks UK (著)

和智 右桂 Growth xPartners Inc. (翻訳)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

個人的、もしくは教育の場での利用であれば、本稿の全部または一部の電子コピーまたはハードコピーを無料で作成することができます。ただし、そのコピーが金銭的利益を生むことを目的に作られ、配布されるものでないことが条件となります。また、そのコピーはこの条文を 1 ページ目に掲げなければなりません。それ以外の目的でのコピーや、出版、サーバーへのポスト、リストへの再配布は特別な許可か料金（あるいはその両方）が必要となります。

Conference'04, Month 1-2, 2004, City, State, Country.

Copyright 2004 ACM 1-58113-000-0/00/0004...\$5.00.

## 要約

モックオブジェクトはテスト駆動開発を拡張したものだ。優れたオブジェクト指向設計をサポートし、整合性のとれた型の体系をコードベース内に発見するよう導く。一般的に、モックオブジェクトはサードパーティライブラリからテストを分離するためのテクニックだと考えられているが、その側面にはそれほど主眼が置かれていないことがわかってきた。この論文では、モックオブジェクトを使うプロセスを応用例を使って説明し、そのプロセスを適用した経験から得られたベストプラクティスとワーストプラクティスを報告する。また、我々の経験を結集した Java フレームワークである jMock についても紹介する。

## 1 導入

モックオブジェクトという命名は失敗だった。モックオブジェクトは実は、オブジェクトの果たすロールに基づいて、システム内の型を識別するためのテクニックなのだ。

[10] において、我々はモックオブジェクトという考え方を、テスト駆動開発をサポートするテクニックだと紹介した。モックオブジェクトによってテストはより適切に構造化されるのであり、さらに重要なことにはドメインコードが改善されると論じたのだ。カプセル化が保たれ、依存対象が減り、クラス間のインタラクションが明確になるからである。本稿では、我々がその後の経験に基づき、このテクニックをどのように精緻化し、調整したかについて説明する。特に、現在我々は、

モックオブジェクトを使うことで得られる最大の恩恵が、元々我々が「インターフェイスの発見」と呼んでいたものにあると理解している。また我々はこの経験に基づき、自分たちのフレームワークを実装し直してモックオブジェクトの動的生成をサポートするようにしたのだ。

本セクションはこの後、我々がテスト駆動開発とオブジェクト指向プログラミングを行う際の優れたプラクティスについてどう理解しているかについて基礎を固める。その上で、モックオブジェクトの概念を説明しよう。本稿ではその後、モックオブジェクトを使ってニーズ駆動開発 (Need-Driven Development) について説明し、事例を示す。その上で、モックオブジェクトを開発した経験について議論し、それを我々のモックオブジェクトフレームワークである jMock にどう適用したかを説明する。

## 1.1 テスト駆動開発

テスト駆動開発 (TDD:Test-Driven Development) を行う場合、プログラマはあるコードのユニットに対して、実際のコードを書く前に、いわゆる**プログラマテスト**を書く [1]。テストを書く作業は、**設計**の活動だ。つまり、各要件を実行可能な具体例のかたちで定義して、動かしてみせるのである。コードベースが成長するにつれ、プログラマはコードをリファクタリングする [4]。重複を排除し、意図を明確にすることで設計を改善するのだ。こうしたリファクタリングは自信を持って行うことができる。テストファーストのアプローチをとっていれば、定義上、高いレベルでテストカバレッジが保証されるので、間違いがあれば検出できるからだ。

これにより設計というものの性質が変化した。設計はかつて、**創作**のプロセスだった。開発者はコードのユニットが何をすべきかを真剣に考え、その上で実装する。それが、**発見**のプロセスになったのだ。ここでは開発者は機能を小さくインクリメンタルに加えていき、動くコードから構造を抽出する。

TDD を行うことで受けられる恩恵は様々だが、最も重要なのはプログラマがコードの設計について考える際に、実装からではなく意図する使い方から考えられるようになるということだろう。また、TDD を行うとコードはシンプルになる傾向がある。これは、将来必要になるかもしれないものに集中するのではなく、今の要件に集中するからであり、リファクタリングを強調することで、開発者がドメインをよりよく理解するようになったときに設計上の弱点を修正できるようになるからである。

## 1.2 オブジェクト指向プログラミング

実行中のオブジェクト指向 (OO:Object-Oriented) プログラムは、協力しあうオブジェクトの網の目だ。オブジェクトは互いにメッセージを送りあうのである。これについて、バックとカニンガムはこう言っている [2]。「孤立しているオブジェクトなどない。あらゆるオブジェクトは、他のオブジェクトとの関係性の中に置かれており、他のオブジェクトが提供するサービスに依存したり、他のオブジェクトをコントロールしたりするのである。」各オブジェクトの目に見えるふるま

いは、メッセージをどのように送り、受け取ったメッセージに対してどう結果を返すかという観点から定義される。

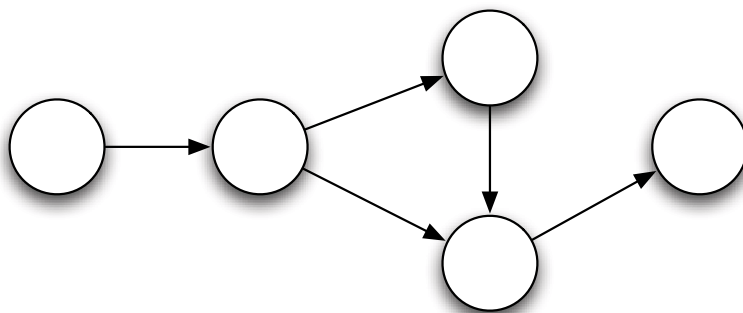


図1 協力しあうオブジェクトの網の目

オブジェクト指向を使うことで受けられる恩恵に、モジュールのユニットが定義されるということがある。モジュールは内部で凝集しつつ、システムの他の部分との結合は最小化されるのだ。こうしておく、オブジェクトの組み合わせ方を変更することで、ソフトウェアを簡単に修正できるようになる。実際にこのような柔軟性を実現するためには、うまく設計されたシステム内のオブジェクトは、直接隣接したオブジェクトにしかメッセージを送ってはいけない。このルールは、デメテルの掟 [15] としても知られている。

「あるオブジェクトに直接隣接したオブジェクト」と言うときには、他のオブジェクトを呼び出したときに参照が返されるオブジェクトが含まれていない点に注意しよう。プログラマはこういうコードを書かないようにしなければならない。

```
dog.getBody().getTail().wag();
```

これは俗に「列車事故」と呼ばれるコードだ。このコードがまずいのは、この一行が別々のオブジェクト 3 つのインターフェイスと内部構造に依存しているからだ。このようなスタイルをとると、コードベース全体に、関係のないオブジェクト間の構造的な依存関係が組み込まれてしまう。これに対する解決策は「命じよ、訊ねるな」 [7] という経験則によって説明される。コードを書き直そう。

```
dog.expressHappiness();
```

そして、このコードが何を意味するかは、dog の実装に決めさせるのだ。

あるオブジェクトが、直接隣接したオブジェクトとだけやり取りするようになっていれば、ある

オブジェクトについて、そのオブジェクトが提供するサービスと、そのオブジェクトが隣接オブジェクトに対して要求するサービスの観点だけから記述できる。このように必要とされるサービスのことを、我々は**外向性インターフェイス** (outgoing interface) と呼ぶ。あるオブジェクトが、他のオブジェクトに対して**要求するさま**を表しているからだ。

### 1.3 オブジェクト指向プログラムのテスト駆動開発

あるオブジェクトの外部とのインタラクションに集中すると、サービスを1つ呼び出し、その結果隣接オブジェクトとどうインタラクションしているかを追跡することでテストできるようになる。テストファーストでプログラミングしている場合、これらのテストを外向性インターフェイス (まだ存在していないかもしれない) の観点から定義できるようになる。それが、あるアクションが成功したかどうかを確かめる手段になるからだ。

たとえば、`dog.expressHappiness()` が成功したら、`dog` の実装が `body.wagTail()` を呼び出すと決めよう。これは `dog` オブジェクトを開発するときに行う設計上の判断で、サービスの実装方法に関するものだ (ここでも、`body` (胴体) に対して `tail` (尻尾) の実装を訊ねないようにすることで列車事故を防いでいることがわかるだろう)。

もし、`DogBody` オブジェクトがまだ `wagTail()` メソッドを実装していなければ、このテストによって満たすべき新しい要件が識別されたことになる。ここで手を止めて新しいフィーチャの実装を始めたくはない。それでは現在の作業から気が逸れてしまうし、`wagTail()` を実装しようとしてもその先の実装が長く続いていつ、いつ終わるかわからなくなってしまうと思われるからだ。その代わり、我々は `DogBody` オブジェクトの偽の実装を提供し、メソッドが実装されているフリをしてもらう。そうすれば、その偽のオブジェクトに仕掛けをして、`expressHappiness()` をテストしている間に `wagTail()` が実際に呼び出されたかどうかを確かめられるのだ。

要約すればこうなる。我々はオブジェクトをテストする際に隣接オブジェクトを偽物と置き換える。そしてその偽物が、期待通りに呼び出されたことを**検証**し、呼び出し元が求めるふるまいをすべて**スタブ化**するのだ。この偽物がモックオブジェクトと呼ばれる。そして、モックオブジェクトを使った TDD のテクニックも、**モックオブジェクト**と呼ばれるのだ。

## 2 モックオブジェクトとニーズ駆動開発

モックオブジェクトによって、TDD の焦点がオブジェクトの状態変化について考えることから、他のオブジェクトとのインタラクションについて考えることに移った。モックオブジェクトを使うことで、あたかもテスト対象のオブジェクトが周囲の環境に求めるものが**すべてそろっているか**のようにコードを書けるようになる。このプロセスにより、あるオブジェクトの周囲の環境が**どうあるべきか**が示される。そこで、我々はその環境を提供できるようになるのだ。

## 2.1 ニーズ駆動開発

リーン開発のコアとなっているプラクティスによれば、価値は必要に応じてプルすることで現実のものとするのであり、実装からプッシュするのではない。つまり、「『プル』」することの効果は、製品を予測で作らない点に現れる。要求が現実化し、顧客は本当は何をほしがっているのかということがわかるまで、約束は先延ばしにされる」のだ [16]。

モックオブジェクトを使ったプログラミングもこの流れで行われる。オブジェクトを個別にテストすることで、プログラマは、あるオブジェクトとそのコラボレーターとのインタラクションについて抽象的に考えざるを得なくなる。しかも、もしかしたら、そうしたコラボレーターがまだ存在しないうちに考えるよう強制されるのだ。モックオブジェクトを使った TDD によって、インターフェイス設計ができるようになる。それも、オブジェクトが**提供するサービス**だけでなく、オブジェクトが**要求するサービス**によって導かれるのだ。このプロセスによって、システムを構成するインターフェイスは狭くなり、それぞれがオブジェクト間のインタラクションを定義したものとなる。そのクラスが提供するフィーチャをすべて記述した広いインターフェイスではなくなるということだ。我々は、このアプローチのことを**ニーズ駆動開発**と呼んでいる。

たとえば、図 2 はオブジェクト A のテストを表現している。A のニーズを満たすためには、サービス S が必要であることがわかった。A をテストする際、我々は S の責務をモックし、具体的な実装は定義しない。

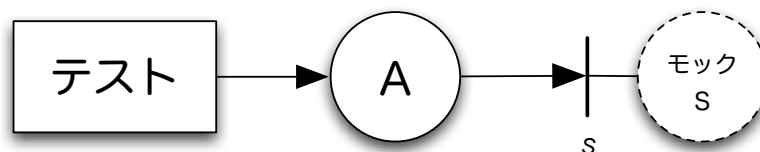


図 2 インターフェイスの発見

いったんこの要件を満たすために A を実装したら、焦点を切り替えて、S のロールを演じるオブジェクトを実装する。これを図 3 にオブジェクト B として示す。このプロセスでは B が必要とするサービスが発見されるが、そこでも再び、B の実装が終わるまではモック化される。

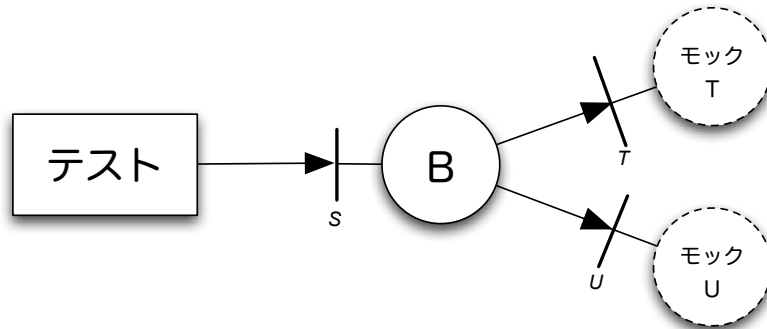


図3 イテレーティブなインターフェイスの発見

このプロセスを繰り返し、最終的にはシステムのランタイムや外部ライブラリという観点で実際の機能を実装しているレイヤにたどり着く。

こうしていると、我々のアプリケーションは、狭く定義されたロールインターフェイスを通じてコミュニケーションするオブジェクトの組み合わせとして構成されることになる。有名な言葉を借りれば「各人は能力に応じて働き、必要に応じて受け取る」ということだ [11]。

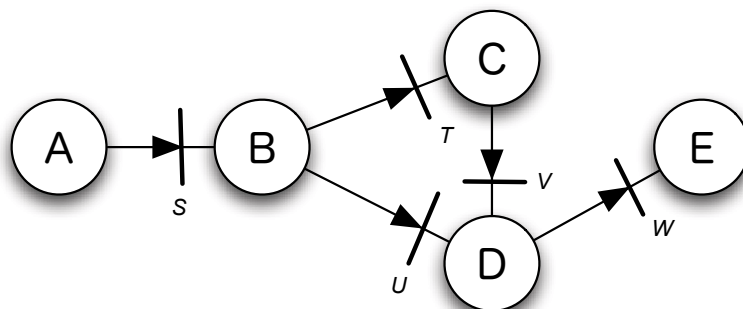


図4 ロールを通じて協力しあうオブジェクトの網の目

我々の経験上、このように作ったシステムのクラス階層はきわめて平坦になる。こうすることで、「脆弱なベースクラス」 [12] のようなよく知られた問題を避けられる。こうした問題があるとシステムは理解しにくくなり、修正も難しくなるのだ。

このプロセスは伝統的なトップダウン開発に似ている。この開発スタイルでは、プログラマは最も抽象度の高いところから始め、1レイヤずつ進んで詳細を埋めていく。ここでの意図は、コードの各レイヤが一貫した用語法に基づいて書かれるようになることだ。こうした用語法は一段抽象度が下がった観点から定義される。これを実際に実現するのは難しい。最も重要な決定を早期に行わなければならない、抽象度の低いレイヤのコンポーネントで重複を避けるのが難しいからだ。TDDによってこの難しさが緩和される。プロセスの中にリファクタリングが含まれるからだ。

一方、ボトムアップ型のプログラミングにも別のリスクがある。著者たちは皆、補助クラスの開発を大きな作業の一部として個別に行い、その結果、何かを誤解していたせいで結果が正しくならなかったという経験をしている。

ニーズ駆動開発を行うことで、目の前にある要件に集中し続け、一貫したオブジェクトを開発できるようになることがわかっている。

### 3 事例

このテクニックを説明するため、あるサンプルを見ていこう。オブジェクトをロードするフレームワークに対してキーを元にした検索を行い、その結果をキャッシュするコンポーネントを考えてほしい。ロードされてから一定時間が経つと、そのインスタンスは使えなくなる。そこで、時々リロードをしなければならなくなる。

モックオブジェクトを使う場合、[10] で説明したような、プログラマテストではよくある構造を用いる。

1. すべてのモックオブジェクトを含んだテストフィクスチャを生成する。
2. モックを使ってエクスペクテーションとスタブを定義する。
3. テストするメソッドを呼び出す。
4. エクスペクテーションを検証し、事後条件をすべてアサートする。

こうすればテストが読みやすくなる。

#### 3.1 オブジェクトローダー

最初を書くプログラマテストは、シンプルな正常ケースにするべきだ。そこで、キャッシュにないオブジェクトをロードし、呼び出し元に返すことにしよう。この場合、各キーに対してローダーが1回だけ呼ばれることを期待する。そして、キャッシュから正しい値が返されていることをチェックする必要がある。jMock フレームワーク（詳細は後述）を使うことで、このためのJUnit[9] テストを書き出すことができる（コードを簡潔にするためインスタンスの生成は省略している。KEY と VALUE はテストケースに定義された定数で、jMock フレームワークの一部ではない）。

```
public class TimedCacheTest {
    public void testLoadsObjectThatIsNotCached() {
        // load メソッドが、KEY を引数に一度だけ呼び出されると期待する
        // その結果、所定の値が返る
        mockLoader.expect(once())
            .method("load").with( eq(KEY) )
            .will(returnValue(VALUE));
    }
}
```

```

mockLoader.expect(once())
    .method("load").with( eq(KEY2) )
    .will(returnValue(VALUE2));

assertSame( "should be first object", VALUE, cache.lookup(KEY) );
assertSame( "should be second object", VALUE2, cache.lookup(KEY2) );
mockLoader.verify();
    }
}

```

jMock はリフレクションを使って、名前とパラメータでメソッドを突き合わせている。エクスペクションを定義するための jMock のシンタックスはやや奇抜である。最初のエクスペクションはこれと同じだ。

```

expectation = mockLoader.expect(once());
expectation.method("load"); expectation.with( eq(KEY) );
expectation.will(returnValue(VALUE));

```

これらの呼び出しをつなぎ合わせて、テストをよりコンパクトに読みやすくしている。これについては、後で議論しよう。

このテストでは、Cache がオブジェクトローダーを表す何者かと関連していることを暗示している。

```

TimedCache cache = new TimedCache (
    (ObjectLoader)mockLoader;
);

```

ここでは、各キーに対して、値を得るためにオブジェクトローダーを一回だけ呼び出すべきだとしている。テストの最後にモックオブジェクトの verify() メソッドを呼び出し、エクスペクションが満たされていることをチェックしている。このテストに通る実装はこんな感じだろう。

```

public class TimedCache {
    private ObjectLoader loader;
    // コンストラクタ
    public Object lookup(Object key) {
        return loader.load(key);
    }
}

```

### 3.1.1 新しいインターフェイスの発見

テストが実際にチェックしているのは、Cache が ObjectLoader を実装したオブジェクトすべてと正しくやり取りしているかどうかということだ。実際のオブジェクトローダーに対するテストはどこか別の場所に書かれることになる。テストを書くために我々に必要なのは、ObjectLoader と名付けられた空のインターフェイスである。これがあればモックオブジェクトを構築できる。テストを通すためには、キーを1つ受け取る load() メソッドがあればよい。ある型に対するニーズを発見したのだ。

```
public interface ObjectLoader {  
    Object load(Object theKey);  
}
```

ここでは、自分たちの書くキャッシュと、最終的に使うであろうオブジェクトローディングフレームワークとの依存関係を最小限に抑えている。モックオブジェクトフレームワークには他にもメリットがあり、複雑なセットアップをしたり、テストのレベルで変更できるデータをそろえたりといった厄介事を避けられる。こうしておけば、オブジェクト間の関係について自由に考えられる。テスト基盤をどう動かすかということに煩わされずにすむのだ。

## 3.2 キャッシュの導入

次のテストケースでは同じ値を2回ルックアップし、2回目はロードされていないことを確認する。ローダーが1回だけ所定のキーで呼び出され、見つかった値が返されることを期待する。2つめのテストはこうなる。

```
void testCachedObjectsAreNotReloaded() {  
    mockLoader.expect(once())  
        .method("load").with( eq(KEY) )  
        .will( returnValue(VALUE) );  
  
    assertEquals( "loaded object", VALUE, cache.lookup(KEY) );  
    assertEquals( "cached object", VALUE, cache.lookup(KEY) );  
}
```

今度は verify() の呼び出しを省略した。実際には MockObjectTestCase クラスで自動的に制御されるのだ。このテストはもちろん失敗し、このようなメッセージが出力される。

```
DynamicMockError: mockObjectLoader: no match found
Invoked: load(<key>)
in:
expected once and has been invoked:
  load( eq(<key > ) ), returns <value>
```

ここから、load() が 2 回呼ばれたことがわかる。このキーに対して 2 回というのは期待と違う。”in:”に続く行では、テスト中にオブジェクトローダーとの間で期待されるやり取りを記述している。このテストを通すには、Cache に HashMap を追加すればよい。単なる値のフィールドではなく、HashMap が必要なのは、1 つめのテストが変わらず通るようにするためだ。

```
public Object lookup(Object key) {
    Object value = cachedValues.get(key);
    if (value == null) {
        value = loader.load(key);
        cachedValues.put(key, value);
    }
    return value;
}
```

もちろん、ここには null をロードできないということが含意されていて、そのことを要件として扱うことになる。その場合、テストを追加して、値がない場合に何が起きるかを示すことになるだろう。

### 3.2.1 インタラクションをテストする

オブジェクトの状態ではなく、オブジェクト間のインタラクションに集中することで、「いったんオブジェクトを検索した後はキャッシュがローダーに戻らなくてもよい」ということを示すことができる。lookup() を 2 回呼び出すが、load() の呼び出しは 1 回だけでなければ失敗するのだ。

また、テストの最後に失敗するのではなく、エラーが起きたときに適切なタイミングで失敗することからも恩恵を受けられる。スタックトレースを見れば、2 回目の lookup() の際の load() 呼び出しで失敗していることがわかり、失敗メッセージを見れば何が起きて、どうすべきだったのかわかるのだ。

## 3.3 時間の導入

要件の中には、時間に依存したふるまいがある。システム時刻を使ってプログラマテストを行うのは避けたい。そんなことをすれば、非決定的な失敗が発生することになるし、タイミングを合わせるために一時停止するせいでテストに時間がかかるようになってしまうからだ。そこで Clock オブジェクトを導入し、Timestamp オブジェクトを返すようにする。今のところは、値の

期限が切れるということが何を意味するかについてあまり深く考えたくない。そこでその判断は `ReloadPolicy` オブジェクトに委ねて、考えるのを先延ばしにする。

この要件のせいで、前のキャッシュヒットに対するテストの前提が変化する。そこで、それに対応して名前を変えよう。今度はテストである値をルックアップし、期限切れになる前にもう一度ルックアップする。タイムスタンプが2回取得されることを期待する。1回は最初のロード時であり、もう一回は2回目のルックアップ時である。ローダーへの呼び出しは1回だけ所定のキーで行われ、見つかった値が返されることを期待する。さらに、2つのタイムスタンプを比較して、キャッシュがまだ生きていることを期待する。

コードを簡潔にするため、タイムスタンプオブジェクトである `loadTime` と `fetchTime` のインスタンス化は省略しよう。テストはこうなる。

```
public void testReturnsCachedObjectWithinTimeout() {
    mockClock.expect(atLeastOnce())
        .method("currentTime").withNoArguments()
        .will( returnValues(loadTime, fetchTime) );

    mockLoader.expect(once())
        .method("load").with( eq(KEY) )
        .will( returnValue(VALUE) );

    mockReloadPolicy.expect(atLeastOnce())
        .method("shouldReload")
        .with( eq(loadTime), eq(fetchTime) )
        .will( returnValue(false) );

    assertEquals( "should be loaded object", VALUE, cache.lookup(KEY) );
    assertEquals( "should be cached object", VALUE, cache.lookup(KEY) );
}
```

ここでも期限に関する要件があるので、`Clock` と `ReloadPolicy` をコンストラクタに渡している。

```
TimedCache cache = new TimedCache(
    (ObjectLoader)mockLoader,
    (Clock)mockClock,
    (ReloadPolicy)mockReloadPolicy
);
```

このテストはもちろん失敗し、こんなメッセージが出力される。

```
AssertionFailedError:
mockClock: expected method was not invoked:
expected at least once:
  getCurrentTime(no arguments),
  returns <loadTime>, then returns <fetchTime>
```

この失敗は `verify()` のときに検出され、キャッシュにタイミングに関するふるまいを導入しなければならないことが示される。`TimedCache` に対する次の変更はもう少し大きくなる。シンプルな `TimestampedValue` クラスを追加してタイムスタンプ/値のペアを保持させ、`loadObject()` で求められたオブジェクトをロードし、現在の時刻を含めた `TimestampedValue` として `cachedValues` にインサートする。

```
private class TimestampedValue {
    public final Object value;
    public final Timestamp loadTime;
}

public Object lookup(Object theKey) {
    TimestampedValue found = (TimestampedValue) cachedValues.get(theKey);
    if( found == null || reloadPolicy.shouldReload( found.loadTime, clock.getCurrentTime() ) ) {
        found = loadObject(theKey);
    }
    return found.value;
}
```

### 3.3.1 コンポジションによるプログラミング

`TimedCache` が必要とするものがすべて渡されていることにお気づきだろう。コンストラクタで渡していることもあれば、メソッド呼び出しのときに渡していることもある。これは、隣接オブジェクトをモック実装で置き換える必要があることから、多かれ少なかれプログラマがやらなければならないとなっている。我々はこのことを強みだと考えている。これによって設計の進む方向が、「機能の絞り込まれた小さいオブジェクトが、自分の知っているコラボレーターとだけインタラクションする」というものになるからだ。またこれによって、プログラマがシステムにおける抽象概念（たとえば、`ReloadPolicy`）を表す型を作るよう促される。すると、コード内で関心事の分離が明確に行われるようになるのだ。

### 3.3.2 抽象概念でのプログラミング

今度のテストでは、`Cache` が現在時刻を 2 回取得していることをチェックしている。1 回は各ルックアップのためであり、この値を正しく `ReloadPolicy` に送っているのだ。時間が何を意味

し、値がどのように期限切れになるかはまだ定義しなくてもよい。我々はメソッドの本質的なフローについて関心を寄せているだけだ。外部の時間を使ってやらなければならないことはすべてインターフェイスを使って抽象化されており、そのインターフェイスもまだ実装していない。これは、オブジェクトローディングの基盤を抽象化したのと同じやり方だ。このコードはタイムスタンプを「不透明な型」として扱っている。したがって、ダミー実装を使うことができる。こうすることで、コアとなるキャッシュのふるまいを正しく行うことに集中できるようになる。

### 3.4 順序の導入

あるオブジェクトのタイムスタンプが、キャッシュにロードされる前にはセットされていないことも関心の対象である。つまり、オブジェクトをロードした後で現在時刻を取得できることを期待しているのだ。テストを調整して、この順序性を強制できる。

```
public void testReturnsCachedObjectWithinTimeout() {
    mockLoader.expect(once())
        .method("load").with( eq(KEY) )
        .will( returnValue(VALUE) );

    mockClock.expect(atLeastOnce())
        .after(mockLoader, "load")
        .method("getCurrentTime").withNoArguments()
        .will( returnValues(loadTime, fetchTime) );

    mockReloadPolicy.expect(atLeastOnce())
        .method("shouldReload")
        .with( eq(loadTime), eq(fetchTime) )
        .will( returnValue(false) );

    assertEquals( "should be loaded object", VALUE, cache.lookup(KEY) );
    assertEquals( "should be cached object", VALUE, cache.lookup(KEY) );
}
```

`after()` 節はある呼び出しの ID を突き合わせる。この場合の相手は別のオブジェクトだ。この ID は `id()` 節によって設定される。この場合のようにデフォルトはメソッド名になる。このテストは失敗する。`loadObject()` の今の実装では、現在時刻を取得して変数に設定するのが、オブジェクトをロードする前だからだ。メッセージはこうなる。

```
DynamicMockError: mockClock: no match found
Invoked: getCurrentTime()
in:
  expected at least once:
    getCurrentTime(no arguments),
    after load on mockObjectLoader,
```

```
returns <loadTime>, then returns <fetchTime>
```

このメッセージを見ると、`getCurrentTime()` が呼び出されたことはわかる。しかし我々が実際に期待していたのは、`getCurrentTime()` の呼び出しが `load()` の呼び出しの後に行われることだ。この2つは同じことではない。実装を修正して、呼び出しを `clock` に移す。

### 3.4.1 正確さのレベルを変える

今度のテストでは、追加の関係性を定義し、オブジェクトがロードされるまでは `clock` への問い合わせを行ってはならないとしている。これが可能になるのは、最終的な状態ではなくオブジェクト間のインタラクションをテストしているからだ。したがって、イベントが発生したときにそれを検知できる。隣接オブジェクトすべてに対してモック実装を使っているので、こうしたアサーションを追加で紐づける場所があるのだ。

一方、`ReloadPolicy` は何度呼ばれても気にしない。パラメータが正しい限り、同じ結果が返されるのだ。これが意味するのは、要件を緩めて「1回だけ呼び出される」ではなく「少なくとも1回は呼び出される」にできるということだ。同じように、`jMock` ではモックオブジェクト呼び出し時のパラメータに対する要件を緩めるのに、制約 (Constraints) と呼ぶテクニックを使うことができる。これについては後で説明しよう。

## 3.5 タイムアウトを導入する

最後に、期限切れとなった値が実際にローダーから消されていることをチェックしたい。この場合、ローダーが同じキーで2回呼び出され、それぞれで別のオブジェクトを返すことを期待する。さらに、`ReloadPolicy` がリロードを要求し、追加でロードされたことに対して `clock` が新しくタイムスタンプを返すことも期待する。

```
public void testReloadsCachedObjectAfterTimeout() {
    mockClock.expect(times(3))
        .method("getCurrentTime").withNoArguments()
        .will( returnValues(loadTime, fetchTime, reloadTime) );

    mockLoader.expect(times(2))
        .method("load").with( eq(KEY) )
        .will( returnValues(VALUE, NEW_VALUE) );

    mockReloadPolicy.expect(atLeastOnce())
        .method("shouldReload")
        .with( eq(loadTime), eq(fetchTime) )
        .will( returnValue(true) );

    assertEquals( "should be loaded object", VALUE, cache.lookup(KEY) );
}
```

```
    assertSame( "should be reloaded object", NEW_VALUE, cache.lookup(KEY) );  
}
```

今の実装でテストは通る。この場合、テストが成功したのが、コードが正しいからであってテストが間違っているからではないことを確認するために、コードを壊して実験するのもよいだろう。

前と同じように、このテストは待ち時間なしでタイムアウトを実行できる。タイミングの側面を抽象化して Cache の外に追い出しているからだ。ReloadPolicy から別の値を返すことで、リロードを強制できる。

### 3.6 テストを後ろから書く

テストを書くとき、実際には最終的な記述と異なる順序で書いていることに気がついた。TDD を行いながら考えていることに従っているのだ。

1. テスト対象のオブジェクトを特定し、必要なパラメータをすべて渡してメソッド呼び出しを書く。
2. エクスペクテーションを書いて、オブジェクトがシステムの他の部分に対して要求するサービスを記述する。
3. それらのサービスをモックオブジェクトとして表現する。
4. テストが実行されるその他のコンテキストを生成する。
5. 事後条件を定義する。
6. モックを検証する。

「命じよ、訊ねるな」の原則に従った結果として、事後条件をアサートする必要がなくなることはよくある（ステップ5）。オブジェクトがどうコミュニケーションしているかを考えていれば、これは驚くようなことではない。

これらのステップを例で示す。

```
public void testReturnsNullIfLoaderNotReady() {  
    Mock mockLoader = mock(ObjectLoader.class); // 3  
    mockLoader.expect(never()) // 2  
        .method("load").with( eq(KEY) )  
  
    mockLoader.stub() // 4  
        .method("isReady").withNoArguments()  
        .will( returnValue(false) );  
    TimedCache cache = new TimedCache((ObjectLoader)mockLoader); // 4  
  
    Object result = cache.lookup(KEY); // 1
```

```
assertNull( "KEY があってはいけない", result ); // 5
mockLoader.verify(); // 6
}
```

テスト対象のオブジェクトとそのロールについて、自分がかつきりとわかっていることを確認するのは特に重要だ。これがはつきりしないせいで、テストで問題が起きたときに混乱が起きることがよくあるからだ。いったんこれが明確になれば、ステップ 2 以降を素直に進められる。

### 3.7 まとめ

この例を見ながら、オブジェクトの状態ではなくオブジェクト間のインタラクションにプログラマが集中することで、オブジェクトのロールの発見がどのように駆動されるかを示してきた。テストを書くことで、機能について考える際の枠組みが与えられる。また、モックオブジェクトによって、これらの関係についてアサーションを行い、応答をシミュレートするためのフレームワークが与えられる。

プログラマは目の前の仕事に集中して、必要な基盤はいずれ手に入ると想定することができる。基盤を後で作ることができるからだ。モックオブジェクトを対象コードに渡す必要があることから、継承ではなく委譲に基づいたオブジェクト指向スタイルにつながる。こうしたことがすべて、関心事を適切に分離したモジュール度の高い設計につながる。

また、モックオブジェクトがあればプログラマが自分たちのテストを必要な正確さで作れるようになる。この例ではアサーションをより正確にするところも、より緩めるところも示している。より正確にするところでは、ある呼び出しが別の呼び出しの後に続かなければならないとしたし、より緩めるところでは、呼び出しが行われるのが 1 回だけではないかもしれないとした。jMock の制約 (Constraint) フレームワークについては、後で議論する。

この例には 1 つ欠点がある。TimedCache の要件自体が、顧客というより上位の存在によって駆動されていないのだ (普通、要件は顧客から来るものだ)。

## 4 モックオブジェクトの実践

著者たちは幅広いプロジェクトで、5 年以上もの間モックオブジェクトを使って仕事をしてきている。また、このテクニックを使っている他の開発者とも会ってきた。最長のプロジェクトで 4 年、最大のチームでは開発者が 15 人いた。言語としては、Java、C#、Ruby、Python それに Javascript を使ってきたし、アプリケーションの規模としては、エンタープライズレベルからハンドヘルドコンピュータまで様々だ。

モックオブジェクトは設計を助けてくれるが、スキルのある開発者の代わりになるものではない。我々の経験によれば、システムの設計が貧弱だと、モックをベースとしたテストがすぐに入り組んだものになってしまうのだ。モックオブジェクトを使うと、結合が密であったり、責務が間

違った場所に置かれていたりといった問題が拡大する。こうした困難に直面した場合の対応の1つに、モックオブジェクトの利用をやめてしまうというものがある。しかし、設計を改善するエンジンとしてモックオブジェクトを使った方がよいと我々は信じている。このセクションでは、役に立つことがわかっているヒューリスティクスをいくつか説明しよう。

#### 4.1 モックするのは、自分の所有する型だけにする

モックオブジェクトは設計テクニックである。したがって、プログラマは自分たちが変更できる型だけをモックとして書くべきだ。そうでなければ、テストの過程で上がってきた要件に対応するために設計を変更することができない。プログラマは固定された型に対してモックを書くべきではない。固定された型とはたとえば、ランタイムや外部ライブラリによって定義されるようなものだ。その代わり、薄いラッパーを書いて、根底にある基盤に対してアプリケーションの抽象を実装するべきなのだ。こうしたラッパーは、ニーズ駆動テストの一部として定義されることになる。

このやり方はプログラマがこのテクニックを理解できるようにする上で強力な洞察であることがわかっている。これにより、モックオブジェクトを利用する際の設計の優位性が取り戻される。モックオブジェクトをサードパーティライブラリとのインタラクションのテストに使ってしまうせいで、この側面が覆い隠されることが多かったのだ。

#### 4.2 ゲッターを使わない

我々がこのテクニックを発見する元々のきっかけとなったのは、ジョン・ノランがゲッターを使わずにコードを書こうという課題を設定したときのことだった。ゲッターは実装を露呈させる。そのためオブジェクト間の結合度が高まり、責務が正しくないモジュールに放置される。ゲッターを使わないようにすることで、オブジェクトの状態ではなくふるまいが強調される。これは責務駆動設計の特徴の1つだ。

#### 4.3 何が起こればいけないかを明記する

テストは求められるふるまいの仕様であり、最初にプログラマがテストを書いた後、かなり時間が経ってから読まれることも多い。ある条件をテストから単純に省略してしまうと、それが不明確になってしまう。あるメソッドが呼ばれてはならないという仕様は、そのメソッドに一切言及しない仕様とは異なる。後者の場合、後からテストを読む人にとってメソッドの呼び出されたことがエラーなのかどうかはつきりしない。我々は必要のない場合であっても、メソッドが呼ばれるべきではないという定義をテストに書く。これは、自分たちの意図を明確にするためだ。

## 4.4 テストでの定義は必要最小限にする

モックオブジェクトを使ってテストをする際には、そのユニットに求められるふるまいの正確な定義と、コードベースが容易に進化できるようにするための柔軟性との間で適切なバランスを見出すことが重要である。TDD を行う場合のリスクの 1 つに、テストが「もろく」になってしまうことがある。つまり、プログラマがアプリケーションコードの関係ない場所を変更したときに、テストが失敗してしまうのだ。これは定義をやりすぎて、オブジェクトに対する要件を表現するのではなく、実装の産物であるフィーチャをチェックしてしまっているせいだ。もろいテストを大量に含むテストスイートは開発速度を落とし、リファクタリングを阻害することになる。

これを解決するにはコードを見直して、仕様を弱めるべきか、あるいはオブジェクトの構造が間違っていて変更するべきなのかを見定めなければならない。アインシュタインによれば、仕様はできる限り正確でなければならないが、必要以上に正確であってはいけないのだ。

## 4.5 境界オブジェクトをテストするのにモックを使わない

あるオブジェクトがシステム内の他のオブジェクトと一切関係を持っていなかったら、モックオブジェクトを使ってテストする必要はない。このようなオブジェクトに対するテストは、メソッドから返される値をアサーションするだけでよい。典型的には、こうしたオブジェクトはデータを格納し、独立した演算を行うか、アトミックな値を表現する。これは当たり前のことかもしれないが、実際には必要ないところでモックオブジェクトを使おうとする人に何度も会ったことがある。

## 4.6 ふるまいを追加しない

モックオブジェクトはスタブにすぎない。テスト環境を複雑にしてはならないし、ふるまいは明らかでなければならない [10]。モックオブジェクトに実際のふるまいを追加し始めるよう迫られる場合、たいていは責務が正しくない場所に置かれていることの兆候であることがわかっている。

これについてのよくある例が、あるモックが入力を解釈して別のモックを返さなければならないというものだ。おそらく、イベントメッセージの解析のようなことを行っているのだろう。これは、テスト対象コードではなくテスト基盤をテストしているというリスクを持ち込んでしまう。

jMock ではこの問題を避けられる。呼び出しのマッチングを行う基盤が期待されるふるまいを定義できるようになっているからだ。一例を示す。

```
mock.expect(once())
    .method("retrieve").with(eq(KEY1))
    .willReturn(VALUE1);

mock.expect(once())
    .method("retrieve").with(eq(KEY2))
```

```
.willReturn(VALUE2);
```

## 4.7 モックするのは直近の隣接オブジェクトだけにする

あるオブジェクトが実装で他のオブジェクトのネットワークを辿らなければならないとすると、それはもろいと考えられる。依存関係があまりに多いからだ。この兆候の1つに、テストのセットアップが複雑で読みにくくなることが挙げられる。モックオブジェクトで似たようなネットワークを構築しなければならないからだ。ユニットテストが最もうまくいくのは、一度に1つのことをテストすることに集中し、直近の隣接オブジェクトに対するエクスペクションだけを設定している場合だ。

この問題を解決するには、自分が正しいオブジェクトをテストしているかを確認するか、オブジェクトとその周囲とをブリッジするロールを導入するかである。

## 4.8 モックが多すぎる

同じような問題は、テスト対象コードに対してあまりに多くのモックオブジェクトを渡さなければならない場合にも起きる。それが直近の隣接オブジェクトであってもだ。ここでも、テストのセットアップは複雑になり、読みにくくなる。これを解決するには、やはり整っていない責務を変更するか、仲介ロールを導入するかどうかだろう。あるいは、テスト対象のオブジェクトが巨大すぎるので、細かく分解して、オブジェクトがより責務に集中し簡単にテストできるようにしなければならないのかもしれない。

## 4.9 新しいオブジェクトのインスタンス化

テスト対象コード内で生成されるオブジェクトとのインタラクションをテストするのは、コンストラクタとのインタラクションも含めて不可能だ。これを解決するには、オブジェクトの生成に介入するしかない。インスタンスを渡すか、new の呼び出しをラップするかである。

この問題にアプローチする上では役立つ方法がいくつかあることがわかっている。インスタンスを渡すためには、コンストラクタかテスト対象オブジェクトの対応するメソッドにパラメータを追加すればよい。どちらになるかは2つのオブジェクト間の関係に依存する。インスタンス生成をラップするには、テストではファクトリーオブジェクトを渡すか、テスト対象オブジェクトにファクトリーメソッドを追加するかである。

ファクトリーオブジェクトのメリットは、新しいインスタンスを生成するのに使われる引数に対してエクスペクションを設定できるということだ。デメリットは新しく型が必要になることである。ファクトリーオブジェクトは、通常ドメインにおける有用な概念を表している。前述の例で

例えば、Clock などだ。

ファクトリーメソッドは単純にその型の新しいインスタンスを返すだけだ。しかし、テスト対象オブジェクトのサブクラスでオーバーライドして、モック実装を返すようにできるかもしれない。これはファクトリー型を作るよりも軽量の現実的解決策であり、暫定的な実装としては効果的だ。

アスペクト指向プログラミングやクラスローダーを操作して実際のオブジェクトを置き換えるといったテクニックを使おうと提案する開発者もいる。これらのテクニックは外部の依存関係を取り除く上では役に立つが、コードベースの設計を改善する助けにはならない。

## 5 モックに関する誤解

我々が「モックオブジェクト」と言うときに意味しているものは、他の人にとってのそれとは異なるし、かつて我々が使っていたものとも異なる。顕著な誤解を挙げよう。

### 5.1 モックはただのスタブである

スタブとは本番コードのダミー実装で、お決まりの結果を返すものだ。モックオブジェクトはスタブのようにふるまうが、テスト対象オブジェクトと隣接オブジェクトとのインタラクションを判定するアサーションも含まれている。

### 5.2 モックオブジェクトはシステムの境界でのみ使われる

我々の考え方はこれと正反対だ。モックオブジェクトが最も役に立つのは、テスト対象コードの設計を駆動するために用いる場合だ。これは、モックオブジェクトが最も役に立つのはインターフェイスを変更できるシステムで使われた場合だということを示唆している。モックとスタブはサードパーティーコードとのインタラクションをテストするのももちろん使える。テストでの依存関係を避ける場合には特にそうだ。しかしそれは、このテクニックにとって二次的な側面にすぎない。

### 5.3 テスト中に状態を収集し、それに対して後でアサートを行う

実装によっては、モックオブジェクト上のメソッドが呼び出されたときに値を設定し、テストの最後でチェックする。この特殊な例が、セルフシャントパターン [3] だ。このパターンではテストクラスがモック自体を実装する。

```
public class TimedClassTest implements ObjectLoader {
    final Object RESULT = new Object();
    final Object KEY = new Object();
    int loadCallCount = 0;
    Object lookupKey;
```

```

// ObjectLoader のメソッド
public Object lookup(Object key) {
    loadCallCount++;
    lookupKey = key;
    return LOOKUP_RESULT;
}

public testReturnsCachedObjectWithinTimeout() {
    // テストの続きをセットアップする...
    assertSame( "loaded object", RESULT, cache.lookup(KEY) );
    assertSame( "cached object", RESULT, cache.lookup(KEY) );

    assertEquals( "lookup key", KEY, lookupKey);
    assertEquals( "load call count", 1, loadCallCount);
}
}

```

これは簡単で自己完結もしているが、明らかな欠点が2つある。第一に、失敗するのは常に、エラーが発生したタイミングではなく事が起きた後だ。アサーションをモックに書くということは、load() に対する余分な呼び出しが行われたタイミングだということである。我々の経験によれば、問題があつてすぐに失敗した方が、**事後**にアサーションを行うよりも理解しやすいし修正も容易だ。第二に、このアプローチではアサーションの実装がテストコード内で分裂してしまう。これでは脳にかかる負荷が上がってしまう。だが、我々の最大の反論は、このアプローチがテスト対象オブジェクトとその隣接オブジェクトとの間のインタラクションに注目していない点に向けられる。このインタラクションこそが、直交していて組み合わせることのできるコードを書く上での鍵だと我々は信じているのだ。Feathers 氏も言っているとおり、セルフシャントはプレースホルダーの実装で、うまくスケールしないようなのだ。

## 5.4 モックオブジェクトを使ってテストをするとコードが重複する

モックオブジェクトの使い方によっては、確かにテスト対象コードを覆い隠してしまい、そのせいでテストがもろくなる。これはサードパーティーライブラリをモックするテストでは実によく見られることだ。ここでの問題は、モックオブジェクトが設計を駆動させるために使われていないことにある。そうではなく、誰かの設計につきあうために使われているのだ。あるレベルでは、モックオブジェクトは対象コードに向けたシナリオを覆い隠すべきだ。だがそれは、コードの設計をテストによって駆動しなければならないからだ。テストのために複雑なモックをセットアップしているというのは、実は、設計に欠けているオブジェクトがあることを示すヒントなのだ。

## 5.5 モックオブジェクトを使うと、多くのテストと一緒に壊れるため、リファクタリングの妨げになる

プログラマの中にはオブジェクトのクラスをテストするのが好む人もいる。テストを変更せずにクラス内のコードをリファクタリングできるからだ。だがこのアプローチにはデメリットもある。各テストがモックオブジェクトを使ったテストよりも多くのオブジェクトに依存してしまうのだ。第一に、新しい要件のせいでコアとなるクラスに変更が入ると、複数のテストを変更しなければならない。テストデータはコードほどリファクタリングしやすすくないため、特にそう言える。第二に、テストが失敗したときにエラーを見つけ出すのがより複雑になる。テストと失敗したコードとのつながりがあまり直接的ではないからだ。さらに悪いことに、もしかしたらデバッグさえも必要になるかもしれない。我々の経験によれば、モックオブジェクトを使ったテストの失敗はより焦点が絞られていて、問題も一目でわかる。そのため、コードを変更する際の所要時間が減るのだ。

## 5.6 メソッド名に文字列を使うと脆弱になる

我々の動的モックフレームワークは文字列を使ってメソッドをルックアップしている<sup>\*1</sup>。したがって、モックされるメソッド名が変更されたとき、開発環境のリファクタリング機能では認識されることもないし変更されることもない。したがって、関連するテストは壊れてしまう。定期的にテストを修正するはめになると、リファクタリングをするのに時間がかかるようになってしまうと信じているプログラマもいる。だが実際には、モックオブジェクト駆動のコードベースでは型の利用は局所的になりがちで、思ったほどには多くのテストは壊れない。またそれらのテストはきれいに壊れるので、やるべき変更が明らかになるのだ。余分なオーバーヘッドはいくらかあるが、我々がエクスペクトーションを指定するやり方の持つ柔軟性を大幅に増加させるという意味で、これは価値があるのだ。

## 6 jMock：ニーズ駆動開発用ツール

jMock はオープンソースのフレームワークで、インターフェイスをモックするための表現力豊かで便利な API を提供している。「モックする」とは期待される呼び出しを定義し、呼び出されたふるまいをスタブ化することである。ここ数年、我々はテスト駆動開発でモックオブジェクトを使ってきたのであり、jMock にはその教訓が集約されている。

テスト駆動開発のプロセスには、フィードバックを与えてモチベーションを維持するためのリズムがある。ペアプログラミング [18] を行った場合には特にそうだ。プログラマがテストを書く手を止めてサポートコードを書かなければならないとすると、このリズムが壊れてしまう。

---

\*1 訳注：jMock2 でこの仕様は改善され、メソッド名のリファクタリングが効くようになっている。

最初のモックオブジェクトライブラリにはこの問題があった。テストを書いている最中にインターフェイスを発見したプログラマは、手を止めてモック実装を書かなければならなかったのだ。jMockAPIは動的コード生成を行って、ランタイムにオンザフライでモックの実装を生成し、プログラマがエクスペクテーションを書いたり、後になって読んだりすることをサポートするためにあらゆること（Java言語ができる範囲で）を行う。

jMockAPIのメインの入り口はMockObjectTestCaseだ。このクラスはJUnitのTestCaseクラスを継承し、モックオブジェクトの使用をサポートしている。MockObjectTestCaseでは、エクスペクテーションを簡単に読めるようにするためのメソッドが提供される。そして、プログラマが間違えることのないように、テストの最後で自動的にモックオブジェクトの結果を検証する。

モックオブジェクトはmock(...)メソッドを呼び出すことで生成される。このメソッドは、インターフェイスの型を表すClassオブジェクトを引数に取り、そのインターフェイスを実装したモックオブジェクトを返す。モックオブジェクトはその後で、モックされた型にキャストし、テスト対象のドメインコードに渡すことができる。

```
class TimedCacheTest extends MockObjectTestCase {
    Mock mockLoader = mock(ObjectLoader.class);
    TimedCache cache = new TimedCache ( (ObjectLoader)mockLoader );
    ...
}
```

mock(...)メソッドから返されるモックオブジェクトには、エクスペクテーションをセットアップするためのメソッドがある。

## 6.1 エクスペクテーションを定義する

jMockの設計が特に目指しているのは、実行することも文章形式で読むことも同時にできるテストを書けるようにすることだ。jMockAPIはほとんどが、エクスペクテーションを読みやすく定義するための構文糖衣を定めている。この目的に従った結果、できあがったAPIは典型的なJavaの設計と比べてかなり型破りなものになった。Javaにホストされたドメイン特化組み込み言語[6]を実装しようとしているからだ。特に、このAPIはデメテルの掟に意図的に違反しており、メソッド名を命令法の動詞にしていない。

エクスペクテーションはいくつかの節を使って定義される。最初の節では、呼び出しを「期待」するのか「スタブ化」するのかを宣言する。jMockはスタブのことをエクスペクテーションの一形態で、実際には起きる必要のないものとして扱う。しかし、スタブとエクスペクテーションの区別はプログラマにとって非常に重要なので、jMockはテストコードでその区別を明確にしているのだ。

その後続く節では、そのエクスペクテーションでテストされるのがモックに対するどのメソッドの呼び出しなのか（マッチングルール）ということや、マッチしたメソッドのスタブ化されたふ

るまいが定義される。場合によってはエクスペクテーションを定義し、後に続くエクスペクテーションで参照できるようにする。1つのエクスペクテーションには複数のマッチングルールが含まれていて、そのルールに合った呼び出しをすべてマッチングする。

エクスペクテーションの各節は、API インターフェイスに対するメソッド呼び出しとしてテストコードの中に表現される。各メソッドはインターフェイスに対する参照を返すので、プログラマはそれを使って次の節を定義できる。さらにその節はその後に続く節で使うためのインターフェイスを返しという具合に続いていくのだ。エクスペクテーション全体を定義する呼び出しのチェーンは、モック自体の `expect()` や `stub()` を呼び出すことで開始できる。

```
mock.expect(エクスペクテーション)
    .method(メソッド名)
    .with(引数の制約)
    .after(1つ前の呼び出しの ID)
    .match(その他のマッチングルール)
    .will(スタブ化されたふるまい)
    .id(この呼び出しの ID);

mock.stub().method(メソッド名)...
```

エクスペクテーションをセットアップするステートメントはメソッドのチェーンで構成されるが、そのメソッド名のおかげでエクスペクテーションを容易に理解できる。デ이지ーチェーン型 API 方式のおかげで、すべてのエクスペクテーションが一貫した順序で定義できるようになるのだ。エクスペクテーションかスタブ、メソッド名、引数、順序やその他のマッチングルール、スタブ化されたふるまい、識別子。このおかげで、他の人が書いたテストコードで作業するのも容易になる。

開発ツールの自動補完を使うと、API は「ウィザード」のようになる。1ステップずつプログラマを導き、エクスペクテーションを定義するというタスクを遂行させるのだ。

## 6.2 柔軟で正確な定義

前述したような「定義のやりすぎ」を避けるため、jMock では、期待されたメソッド呼び出しを実際の値ではなく満たすべき制約としてプログラマが定義できるようにしている。制約は引数の値をテストするのに使われ、場合によってはメソッド名をテストするのにも使われる。制約のおかげで、プログラマはテストされる機能と関係のないオブジェクトのインタラクションという側面を無視できるようになる。

制約は通常、許可される引数の値を定義するのに用いられる。たとえば、ある文字列が期待される文字列を含んでいることをテストしつつ、フォーマットや正確にはどういう文字列かという不要な詳細を無視できる。最もよくあるのが引数と期待値との比較であるが、制約を使えば、その比較が等価性によるのか (`equals` メソッド)、同一性によるのか (`==` 演算子) を明示できる。パラ

メータをすべて無視することもよくあるが、それは `IS_ANYTHING` 制約を使って定義できる。

制約は `MockObjectTestCase` の「糖衣」メソッドで生成される。エクスペクションビルダーインターフェイスの `with` メソッドで引数の制約を定義できるのだ。次に示すエクスペクションでは、`pipeFile` メソッドが引数を 2 つ渡されて一度呼ばれなければならないと定義している。メソッドの一方は期待された `fileName` と等価であり、もう一方は `mockPipeline` オブジェクトそのものでなければならない。

```
mock.expect(once())
    .method("pipeFile")
    .with(eq(fileName), same(mockPipeline))
    .will( returnValue(fileContent) );
```

パラメータの値だけでなく、他にも様々なものとマッチできることは役に立つ。たとえば、`JavaBean` のプロパティゲッターのように、あるオブジェクトのメソッドのサブセットに対してマッチできれば便利なことが多い。この場合、`jMock` を使えばプログラマは複数のメソッドに対して制約を定義できる。デフォルトの結果を作成する仕組みと一緒に使えば、オブジェクトのインターフェイスの中でテストと関係のない側面を無視し、関心対象の側面だけに集中できる。

```
mock.stub().method(startingWith("get"))
    .withNoArguments()
    .will(returnADefaultValue);
```

`jMock` を使えば、モックに対する呼び出しの順序に関する制約や別々のモックに対する呼び出しの順序などといった、より複雑なマッチングルールも定義できるようになる。普通は順序制約を使わなくてもよいので、使う場合には注意する必要がある。使うと、テストがもろくなりすぎてしまうことがあるのだ。`jMock` は個々の呼び出しの局所的な順序を定義してしまうというリスクを最小化している。順序については、シーケンスの例を紹介するときに使い方を示す。

`jMock` を使うと、ユーザーは引数の制約を定義し、テストが読みやすくなるようにしなければならない。多少余分なタイピングをしなければならなくなったとしても、結果として明確になるならばよいとユーザーは考えるようだ。ちょっとしたエラーを防げるようになるからである。

### 6.3 拡張性

`jMock` では制約やマッチングルールに関する大量のライブラリが提供されているが、プログラマが必要とするであろうシナリオをすべて網羅することはできない。実際、問題領域に特化した制約を作ることで、テストは一層明確になる。そのため、マッチングルールと制約は拡張できるように

なっている。プログラマは、jMock のシンタックスをシームレスに拡張して自分たちの独自ルールや制約を定義できる。

たとえば、複数のイベントを引き起こすオブジェクトは、イベントが発生するたびに新しいオブジェクトを生成する。特定のオブジェクトから発生するイベントとマッチさせるためには、カスタム制約を書いて、期待された発生元と実際の発生元を比較するようにすればよい。

```
mock.expect(once())
    .method("actionPerformed")
    .with(anActionEventFrom(quitButton));
```

jMock は何よりもまずニーズ駆動開発をサポートするように設計されている。そのため、それ以外のシナリオには jMock の API がうまく当てはまらないかもしれない。jMock を修正してほしいと頼んでくるユーザーは何人もいる。たとえば、こんな要望がある。インテグレーションテストを行う際にも使えるようにしてほしい。手続き型プログラミングでも使えるようにしてほしい。貧弱な設計のコードをリファクタリングしなくてもすむようにしてほしい。物理クラスのモックができるようにしてほしい。しかし、こうした要望を我々は丁寧に断っている。すべてができる万人向け API などというものは存在しない。しかし、jMock には一般的なテストを書くための便利な構成要素が数多くある。必ずしもニーズ駆動開発をやらなくてもいいのだ。そのため、jMock ではレイヤ化設計を行っている。jMock API は「構文糖衣」であり、それを実装しているコアのオブジェクト指向フレームワークはその他のテスト API を作るのにも使える。これらのコア API を説明することは、本稿のスコープを越える。興味があれば、jMock の Web サイトを見てほしい。

## 6.4 失敗したときのために

jMock は情報量の多いメッセージを生成しているので、テストの失敗が何に起因するのかを容易に判断できる。モックオブジェクトには名前をつけられるので、失敗メッセージが出たときに、テストコードやテスト対象コードの実装と簡単に関連づけられるようになっている。エクスペクテーションを定義するために構成されたコアオブジェクトには、組み合わせることで明確な失敗メッセージとなる説明文が書かれている。

デフォルトでは、モックオブジェクトの名前はモックする型を元につけられる。しかし、テスト内でモックが果たすロールを記述した名前を使った方が役に立つことも多い。その場合、モックのコンストラクタにモック名を明示的に渡すことで名前をつけられる。

```
namedMock = mock(MockedType.class, "namedMock");
```

適切なエラーメッセージを生成するのに役立つテストテクニックは他にも数多くあることがわ

かっている。自己記述的な値 (Self Describing Values) やダミーオブジェクト (Dummy Object) などだ。自己記述的な値は、エラーメッセージの一部として出力された場合に、自身のロールを説明する。たとえば、ファイル名として使われる文字列は「開いたファイルの名前」のような値であるべきで、「invoice.xml」のような実際のファイル名ではない。ダミーオブジェクトは、オブジェクト間で受け渡しされるが、テスト中に呼び出されることがないオブジェクトだ。テストでは、ダミーオブジェクトを使ってエクスペクションやアサーションを定義し、テスト対象オブジェクトが隣接オブジェクトと正しくコラボレーションしていることを確認する。jMockAPI には便利な機能が含まれていて、自己記述的なダミーオブジェクトを生成できる。

```
Timestamp loadTime = (Timestamp) newDummy(Timestamp.class, "loadTime");
```

ダミーオブジェクトを使えば、プログラマは型の定義や、型をどうインスタンス化するかといった設計上の意思決定を先延ばしにできる。

## 7 関連した研究

責務駆動設計 [19] はオブジェクト指向ソフトウェアを設計するのに役立つアプローチであるとして知られている。ニーズ駆動開発は、テストファーストで責務駆動開発を行うためのテクニックだ。モックオブジェクトを使うことで、テストを書くという行為を通じて、ロールや責務を発見できるようになる。

元々の `mockobjects.com` ライブラリ [10] では、モックを手で書く場合にエクスペクションを定義し、検証するためのローレベルなライブラリが提供される。インターフェイスのモック実装を書くために時間をとらなければいけないというのは、テスト駆動開発サイクルのリズムを妨げるし、インターフェイスが変更になった場合に余分な作業が必要になる。これを低減するため、このプロジェクトでは JDK や J2EE の共通インターフェイスの多くに対してモック実装を提供している。だが、すべてのライブラリに対してこれを行うのは非現実的だし、モックオブジェクトの使い道を設計ツールとしてではなくテスト目的に集中してしまっている。

MockMaker[14] は、ユーザーが定義したインターフェイスを元に、ビルド時にモックオブジェクトのソースコードを自動生成する。こうすればモックオブジェクトを使って設計を行うようになるし、プログラミングのリズムが妨げられることもない。欠点は、この処理のせいでビルドプロセスが複雑になり、生成されたモックオブジェクトのカスタマイズが難しくなってしまうことだ。

EasyMock[5] は動的コード生成を通じて実行時にモックオブジェクトを生成する。この特徴は「記録と再生」型の API だ。テストコードでは、モックオブジェクトを「記録モード」にしてモックオブジェクトに期待される呼び出しを行い、「再生モード」にしてからテスト対象のオブジェクトを呼び出す。その後、モックオブジェクトは記録されたのと同じ引数で同じ呼び出しが行われたことを検証する。このため、初めて使うユーザーにとっても API は使いやすく、リファクタリン

グツールを使う場合でもうまくいく。しかし、エクスペクテーションの定義がシンプルだと、結果として定義をやりすぎたり、テストがもろくなったりしてしまう。

DynaMock[13] も実行時にモックオブジェクトを自動生成する。この API は、要求されたふるまいの仕様として読めるように設計されている。しかし、この API は柔軟性が低く、ユーザーが拡張するのは難しい。

アスペクト指向プログラミング [8] やバイトコード操作を使って、テスト中に行われたアプリケーションオブジェクトの呼び出しをモックオブジェクトの呼び出しにリダイレクトするプロジェクトもある。柔軟性の低いサードパーティー API に対してテストコードを実行するのであれば、このアプローチも便利だろう。しかし、このアプローチは単なるテストテクニックであり、設計プロセスに対する有用なフィードバックは得られない。

## 8 課題

今後 jMock に関して我々は、API を改善して自動リファクタリングツールやコード補完がうまく働くようにしようと考えている。一方で、現在の API が持っている柔軟性や表現力は保ちたい。

jMock を他の言語に移植することも考えている。C# もそうであるし、Ruby や Python のような動的言語もそうだ。jMock を開発する労力のほとんどは、便利なドメイン特化言語を Java で定義する方法を探究することに費やされた。移植する際の設計上の目標は、現在の API の表現力を保ちつつ、移植先言語のイディオムをサポートすることである。

モックオブジェクトのテクニックの課題は、別々のテスト間での整合性を維持し、確かめることがうまくできない点にある。モックオブジェクトを使ってテストをすることで、テスト対象のオブジェクトが行うと期待される一連の呼び出しが検証される。しかし、これでは、同じインターフェイスを使っているオブジェクトがすべて一貫したやり方を使っていることはテストできない。また、そのインターフェイスの実装に整合性があるかどうかもわからない。現在我々はこの課題を解決するため、インテグレーションテストを行ったり、エンドツーエンドで受け入れテストを実行したりしている。こうすることで、統合のエラーは検知できるが、エラーの原因を特定するのは厄介だ。現在は、クライアントコードやインターフェイス実装の整合性をテストするための API の開発に取り組んでいる。この API ではオブジェクト間のプロトコルを明示的に記述できるようにする予定だ。

## 9 結論

モックオブジェクトに関する以前の論文を書いた後も、我々の基本的な考え方が日々の使用に耐えることは複数のプロジェクトを通じてわかっている。このテクニックに関する我々の理解は深まっており、特に今では、モックオブジェクトを単にテストだけでなく設計に使おうと考えるようになってきている。現在では、要件から優れた設計を導き出す上でモックオブジェクトが果たす役割や、技術的な限界もわかっている。我々は自分たちの経験を jMock に実装した。これは新世代

のモックオブジェクトフレームワークであり、これを使えばニーズ駆動開発をサポートするのに必要な表現力が得られると信じている。

## 10 謝辞

Martin Fowler 氏、John Fuller 氏、Nick Hines 氏、Dan North 氏、Rebecca Parsons 氏、Imperial College の諸兄、ThoughtWorks の同僚たちと eXtreme Tuesday Club の同僚たちに感謝する。

## 11 REFERENCES 参考文献

- [1] Astels, D. Test-Driven Development: A Practical Guide, Prentice-Hall, 2003.
- [2] Beck, K. and Cunningham, W. A Laboratory For Teaching Object-Oriented Thinking. In SIGPLAN Notices (OOPLSA'89), 24, 10, October 1989.
- [3] Feathers, M. The "Self-Shunt" Unit Testing Pattern, May 2001. Available at: <http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf>
- [4] Fowler, M. et al. Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999.
- [5] Freese, T. EasyMock 2003. Available at: <http://www.easymock.org>
- [6] Hudak, P. Building domain-specific embedded languages. ACM Computing Surveys, 28(4es), December 1996.
- [7] Hunt, A. and Thomas, D. Tell, Don't Ask, May 1998. Available at: [http://www.pragmaticprogrammer.com/ppllc/papers/1998\\_05.html](http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.html)
- [8] Kiczales, G., et al. Aspect-Oriented Programming, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
- [9] JUnit. 2004. Available at <http://www.junit.org>
- [10] Mackinnon, T., Freeman, S., Craig, P. Endo-testing: unit testing with mock objects. In Extreme Programming Examined, Addison-Wesley, Boston, MA. 2001. 287-301.
- [11] Marx, K. Critique of the Gotha Program, 1874.
- [12] Mikhajlov, L. and Sekerinski, E. A Study of the Fragile Base Class Problem. In E. Jul (Ed.), ECOOP'98 - Object-Oriented Programming 12th European Conference, Brussels, Belgium, July 1998, pp 355-382, Lecture Notes in Computer Science 1445, Springer-Verlag, 1998.
- [13] Massol, V. and Husted, T. JUnit in Action, Manning, 2003.
- [14] Moore, I. and Cooke, M. MockMaker, 2004. Available at: <http://www.mockmaker.org>
- [15] Lieberherr, K. and Holland, I. Assuring Good Style for Object-Oriented Programs IEEE Software, September 1989, 38-48.

- [16] Poppendieck, M. Principles of Lean Thinking, In OOPSLA Onward!, November 2002.
- [17] Sun Microsystems, Java Messaging Service, Available at:  
<http://java.sun.com/products/jms>
- [18] Williams, L. and Kessler, R. Pair Programming Illuminated. Addison-Wesley, Reading, MA, 2002. ISBN 0201745763.
- [19] Wirfs-Brock, R. and McKean, A. Object Design: Roles, Responsibilities, and Collaborations, Addison-Wesley, Reading, MA, 2002.

## 12 日本語版あとがき

本稿は 2004 年に発表された論文「Mock Roles, Not Objects」(<http://www.jmock.org/oops1a2004.pdf>) の全文を、著者である Steve Freeman 氏、Nat Pryce 氏の許可を得て翻訳したものです。jMock については、現在では最新版の jMock2 がリリースされており、Java1.5 のジェネリクスを受けてメソッド名の定義を文字列ではなく型付けできるようになったことをはじめ、API には変化が見られます。しかし、本稿で説明されているようなモックに関する根本的な考え方は現在にも引き継がれており、jMock を利用する上でおいに参考になります。

なお、本稿の日本語訳にあたっては、角谷信太郎さん、高木正弘さんにお世話になりました。深く感謝します。(和智)