

6 Unabhängigkeit durch Dummy- und Mock-Objekte

In einer durchschnittlich komplizierten Anwendung kommt kaum ein Objekt ohne die Mitwirkung zahlreicher anderer Objekte – der gleichen oder einer anderen Klasse – aus. Wie also testet man dieses Objekt, das von so vielen anderen abhängt? Der pragmatischste und (vermutlich) intuitivste Weg aus diesem Dilemma ist der *Bottom-up-Ansatz*: Man beginnt die Entwicklung und das Testen mit den Klassen, die selbst nur auf systemeigenen Klassen aufbauen. Danach benutzt man diese getesteten Komponenten zum Aufbau abhängiger Klassen (vgl. auch Seite 50 ff.).

Dieses Vorgehen nennt sich *bottom-up*, da man »ganz unten«, also bei den konkretesten Objekten beginnt und sich nach oben zu den abstrakteren Systemkomponenten hocharbeitet. Dass man beim Test-First-Ansatz einfacher *top-down* (bzw. *outside-in*) vorgeht, haben wir bereits in Kapitel 3.2 gesehen. Dort sind wir auch auf Probleme gestoßen, die durch die Abhängigkeit verschiedener Klassen untereinander und zu externen Quellen hervorgerufen wurden. Wie man diese Abhängigkeiten in vielen Fällen vermeiden oder auch nachträglich eliminieren kann, zeigt dieses Kapitel.

6.1 Kleine Attrappe

Eine wichtige Regel des Unit-Testens taucht immer wieder auf: Der einzelne Testfall soll so *lokal* wie möglich sein, d.h., er soll nur das Objekt testen, das wir gerade unter der Lupe haben, und nicht all die anderen, die es bei seiner Arbeit benötigt, mit denen es zusammenarbeitet und an die es Aufgaben delegiert. Ein Weg, um diesem Ziel der größtmöglichen Unabhängigkeit eines Tests näher zu kommen, sind *Dummy-Objekte*. Das englische Wort *Dummy* bedeutet *Attrappe*, d.h., wir ersetzen einen Teil unserer Objekte durch andere, die nur so tun als ob. Ein einfaches Beispiel veranschaulicht diese Idee:

*Lokalität und
Unabhängigkeit*

Wir möchten einen Euro-Rechner programmieren, der für einen gegebenen Betrag und Währung den entsprechenden Betrag in Euro zurückgibt¹. Zunächst natürlich die Tests:

```
public class EuroCalculatorTest extends TestCase {
    public EuroCalculatorTest(String name) {
        super(name);
    }
    public void testEUR2EUR() {
        double result =
            new EuroCalculator().valueInEuro(1.0, "EUR");
        assertEquals(1.0, result, 0.00001);
    }
    public void testUSD2EUR() {
        double result =
            new EuroCalculator().valueInEuro(1.0, "USD");
        assertEquals(1.1324, result, 0.00001);
    }
}
```

Als fertige Komponente steht uns die Klasse `ExchangeRateProvider` zur Verfügung, die über eine Netzwerkverbindung den Wechselkurs für alle gängigen Währungen bereitstellt:

```
public class ExchangeRateProvider {
    public double getRateFromTo(String fromCurrency,
                               String to) {
        double retrievedRate = ... // Netzzugriff auf Server
        return retrievedRate;
    }
}
```

Probleme

Die Implementierung der `EuroCalculator`-Klasse ist denkbar einfach; dennoch haben unsere Tests ein paar Probleme: Zum einen funktioniert der Testfall `testUSD2EUR()` vermutlich nur für kurze Zeit, nämlich genau so lange, bis sich der Wechselkurs zwischen Dollar und Euro ändert. Zum anderen ist der Zugriff auf den Wechselkursserver eine sehr unsichere Sache, da es sich um einen Netzzugriff handelt. Der Server kann lange Antwortzeiten haben oder wegen Überlastung für unabsehbare Zeit gar nicht verfügbar sein. Durch diese Abhängigkeit von einem externen Dienst geht unser Test möglicherweise schief, ohne dass unser Programm einen Fehler aufweist.

-
1. Zur Vereinfachung benutzen wir hier den primitiven Typ `double` zur Repräsentation von Geldbeträgen, der wegen seiner Rundungsproblematik im »richtigen« Leben nur selten zum Einsatz kommt.

Eine Möglichkeit, dieses Problem zu lösen, ist die Verwendung eines »falschen« Wechselkurservers, dem wir den erwarteten Wechselkurs gleich mitgeben können:

Erste Attrappe

```
public class DummyProvider extends ExchangeRateProvider {
    private double dummyRate;
    public DummyProvider(double dummyRate) {
        this.dummyRate = dummyRate;
    }
    public double getRateFromTo(String from, String to) {
        return dummyRate;
    }
}
```

Jetzt bleibt nur noch die Frage, wie wir unsere Attrappe dem EuroCalculator unterschmuggeln. Eine Möglichkeit ist, die Signatur der `valueInEuro()` Methode um einen Parameter vom Typ `ExchangeRateProvider` zu erweitern. Bei dieser Gelegenheit wird auch gleich noch die erwartete Genauigkeit in eine Konstante `ACCURACY` hinausgezogen. Der geänderte Test sieht damit folgendermaßen aus:

```
public class EuroCalculatorTest extends TestCase {
    private final static double ACCURACY = 0.00001;
    public EuroCalculatorTest(String name) {
        super(name);
    }
    public void testEUR2EUR() {
        ExchangeRateProvider provider =
            new DummyProvider(1.0);
        double result = new EuroCalculator().
            valueInEuro(2.0, "EUR", provider);
        assertEquals(2.0, result, ACCURACY);
    }
    public void testUSD2EUR() {
        ExchangeRateProvider provider =
            new DummyProvider(1.1324);
        double result = new EuroCalculator().
            valueInEuro(1.5, "USD", provider);
        assertEquals(1.6986, result, ACCURACY);
    }
}
```

Und siehe da, unser Test läuft schnell, stabil und ist von den sich ändernden Kursen unabhängig. Uns muss jedoch klar sein, dass wir nun jedoch einzig die Klasse `EuroCalculator` testen, nicht jedoch den Währungskursserver. Aber dieser wird (hoffentlich) bereits vom

Anbieter der Komponente getestet worden sein. Sind wir selbst dieser Anbieter, dann testen wir auch den Server, aber in einer anderen Test-suite.

6.2 Begriffswirrwarr

Bevor wir uns anspruchsvolleren Fälschungen widmen, zunächst ein Blick auf die verwendeten Begriffe: Im angloamerikanischen Sprachgebrauch existieren zahlreiche Wörter für das, was wir hier Dummy-Objekte nennen, u.a. *Dummy*, *Stub*, *Mock* und *Shunt*. Unsere Verwendungsweise ist die folgende:

Stub Ein *Stub* (dt. Stummel) ist ein bislang nur rudimentär implementierter Teil der Software, der später durch die richtige Implementierung ersetzt werden soll. Die Aufgabe eines Stub-Objekts ist die eines Platzhalters für geplante, aber noch nicht umgesetzte Funktionalität.

Dummy Ein *Dummy* (dt. Attrappe, Schaufensterpuppe) dagegen kann die echte Implementierung für Testzwecke ersetzen. Ob das echte oder ein Dummy-Objekt verwendet wird, entscheidet sich durch codeinterne oder externe Konfiguration.

Mock Ein *Mock* (dt. Nachahmung) unterscheidet sich vom *Dummy* durch zusätzliche Funktionalität: Ein Mock-Objekt erlaubt, falls nötig, die Einstellung der von ihm gewünschten Reaktionen und das Verifizieren des korrekten Verhaltens seines »Klienten«. Mock-Objekte werden ausführlich in Kapitel 6.5 besprochen.

In der Literatur und im Web ist die Verwendung der Begriffe jedoch alles andere als konsistent und man muss damit rechnen, dass jeder Begriff als Synonym eines beliebig anderen benutzt wird – und umgekehrt.

6.3 Große Attrappe

Das obige Euro-Rechner-Beispiel besticht durch seine Einfachheit, da es nichts anderes tut, als eine in der Realität komplexe Funktion durch festverdrahtete Werte zu ersetzen, die genau auf die Tests abgestimmt sind. Betrachten wir ein komplexeres Problem:

In den meisten Applikationen benötigen wir eine Möglichkeit, verschiedenste Ereignisse während des Programmablaufs an zentraler Stelle festzuhalten. Um diese *Logging-Funktionalität* überall im Programm auf konsistente Weise durchführen zu können, definieren wir uns ein standardisiertes Interface:

```
public interface Logging {
    public final static int DEFAULT_LOGLEVEL = 2;
    public void log(int logLevel, String message);
    public void log(String message);
}
```

Das Interface erlaubt das Loggen einer Nachricht `message` unter Angabe einer Log-Stufe `loglevel`, um etwa zwischen Fehler- und Debug-Meldungen unterscheiden zu können. Zudem soll die Möglichkeit bestehen, auch ohne expliziten Loglevel, d.h. mit einem Standardwert `DEFAULT_LOGLEVEL`, zu arbeiten.

Eine erste Implementierung von `Logging` soll die Klasse `LogServer` sein, die unter Angabe eines Dateinamens erzeugt wird und alle Log-Einträge in diese Datei schreibt. Wie immer beginnen wir mit einem Test:

```
public class LogServerTest extends TestCase {
    public void testSimpleLogging() {
        Logging logServer = new LogServer("log.test");
        logServer.log(0, "Zeile eins");
        logServer.log(1, "Zeile zwei");
        logServer.log("Zeile drei");
        // assertTrue(??) 0ops, und jetzt?
    }
}
```

Während uns die ersten vier Zeilen des Tests geradezu aus den Fingern fließen, befinden wir uns jetzt in einem Dilemma: Wie gelangt man an die Innereien der Datei `log.test`, um zu überprüfen, ob der `LogServer` seine Arbeit auch wirklich ordentlich verrichtet? Eine Möglichkeit wäre es, unseren Log-Server um eine Funktion `getLoggingFile()` zu erweitern. Allerdings hätten wir uns dann auf Datei-Logging festgelegt und ein Implementierungsdetail nur für Testzwecke offengelegt. Zudem kann das Öffnen und Lesen einer Datei Schwierigkeiten mit sich bringen, die eine kontrollierte und wiederholbare Testdurchführung erschweren:

*Wie testet man
Dateizugriffe?*

- Wie finde ich einen Pfad, der für Testzwecke les- und schreibbar ist?
- Wie stelle ich sicher, dass die Zugriffsrechte in diesem Pfad stimmen?
- Wie gehe ich sicher, dass die Datei nicht bereits existiert bzw. vor dem Test gelöscht wird?

All dies sind Probleme, die von uns im Produktivbetrieb zwar bedacht werden müssen, für den gegenwärtigen Stand der Entwicklung aber unbedeutend sind bzw. sein sollten.

PrintWriter statt *Datei*

Unsere Kenntnisse der Java-IO-Klassen helfen hier weiter: Wie wäre es, wenn wir unserem `LogServer` anstatt einem Dateinamen einfach eine Instanz vom Typ `java.io.PrintWriter` im Konstruktor übergeben. Dieser bietet die Möglichkeit, unsere Log-Nachricht per `println()` auszugeben, und ist nicht nur für Dateien, sondern auch für jede Art von `OutputStream` zu gebrauchen. Unser Test ändert sich damit folgendermaßen:

```
public void testSimpleLogging() {
    PrintWriter writer = new PrintWriter(
        new FileOutputStream("log.test"));
    Logging logServer = new LogServer(writer);
    logServer.log(0, "Zeile eins");
    logServer.log(1, "Zeile zwei");
    logServer.log("Zeile drei");
    // assertTrue(??) Oops, und jetzt?
}
```

Trotzdem stehen wir weiterhin vor dem Problem, zunächst an die Datei herankommen zu müssen, um die nötigen Überprüfungen durchführen zu können. Das Wissen jedoch, dass unser `LogServer` nichts weiter tun soll, als mittels der Methode `println(..)` eine Kombination aus `LogLevel` und Log-Nachricht auszugeben, führt zu einer weiteren Intuition: Warum nicht, wie im obigen Euro-Rechner-Beispiel, unsere eigene Unterklasse von `PrintWriter` implementieren, deren Instanzen all das, was ihnen per `println()` übergeben wird, aufzeichnen und für spätere Überprüfungen zur Verfügung stellen? Diese Überlegung lässt folgende *Dummy-Klasse* entstehen:

```
import java.io.PrintWriter;
import java.util.*;
public class DummyPrintWriter extends PrintWriter {
    private List logs = new ArrayList();
    DummyPrintWriter() {
        super((OutputStream) null);
    }
    public void println(String logString) {
        logs.addElement(logString);
    }
    public String getLogString(int pos) {
        return (String) logs.get(pos);
    }
}
```

Mit Unterstützung dieser `DummyPrintWriter`-Klasse lässt sich unser Test jetzt einfacher und klarer formulieren:

```
public void testSimpleLogging(){
    DummyPrintWriter writer = new DummyPrintWriter();
    Logging logServer = new LogServer(writer);
    logServer.log(0, "Erste Zeile");
    logServer.log(1, "Zweite Zeile");
    logServer.log("Dritte Zeile");
    assertEquals("0: Erste Zeile", writer.getLogString(0));
    assertEquals("1: Zweite Zeile",
        writer.getLogString(1));
    assertEquals("2: Dritte Zeile",
        writer.getLogString(2));
}
```

Geschafft! Oder etwa doch nicht? Sieht man genauer hin, dann weist unser Test noch einige Unschönheiten auf: Um `DummyPrintWriter` als Unterklasse von `PrintWriter` einsatzfähig zu machen, haben wir tricksen müssen: Zum einen führt unser Konstruktor einen *Cast* auf das null-Objekt durch; dies ist unästhetisch, aber nötig, um dem Java-Compiler die statische Bestimmung des richtigen Super-Konstruktors zu erlauben. Zum anderen machen wir die gefährliche Annahme, dass unser Log-Server ausschließlich die Methode `println(..)` der `PrintWriter`-Instanz aufruft. Warum nicht einfach diese implizite Annahme durch die Einführung eines Interfaces explizit machen? Gesagt – getan ...

Mangelnde Ästhetik

```
public interface Logger {
    public void logLine(String logString);
}
```

Natürlich ändert sich damit auch der Konstruktor unseres Log-Servers und folglich auch der Test:

```
public void testSimpleLogging() {
    DummyLogger logger = new DummyLogger();
    Logging logServer = new LogServer(logger);
    logServer.log(0, "Erste Zeile");
    logServer.log(1, "Zweite Zeile");
    logServer.log("Dritte Zeile");
    assertEquals("0: Erste Zeile", logger.getLogString(0));
    assertEquals("1: Zweite Zeile",
        logger.getLogString(1));
    assertEquals("2: Dritte Zeile",
        logger.getLogString(2));
}
```

und unser `DummyPrintWriter` wird zu einem `DummyLogger`:

```
import java.util.*;
public class DummyLogger implements Logger {
    private List logs = new ArrayList();
    public void logLine(String logString) {
        logs.add(logString);
    }
    public String getLogString(int pos) {
        return (String) logs.get(pos);
    }
}
```

Im Vergleich zu den Mühen, die uns das »Testbar-Machen« gekostet hat, erscheint die eigentliche Implementierung von `LogServer` trivial:

```
public class LogServer implements Logging {
    private Logger logger;
    public LogServer(Logger logger) {
        this.logger = logger;
    }
    public void log(int logLevel, String message) {
        String logString = logLevel + ": " + message;
        logger.logLine(logString);
    }
    public void log(String message) {
        this.log(DEFAULT_LOGLEVEL, message);
    }
}
```

Lohnt sich die Mühe?

War das jetzt wirklich die ganze Mühe wert? Sieht man nicht auf einen Blick, dass die Klasse genau das tut, was sie tun soll? Bevor wir an die Beantwortung dieser Frage gehen, lohnt es sich anzuschauen, was wir alles mit der Einführung unseres »Dummys« erreicht haben und was nicht: Durch die Einführung des `Logger`-Interfaces haben wir einen `Log-Server`, dessen Implementierung von systemnahen `IO`-Klassen unabhängig ist. Dieses Interface erlaubt es uns auch, unterschiedliche `Logger` zu implementieren, die unser `Server` ohne Modifikationen verwenden kann.

*Dependency Inversion
Principle*

Wir haben damit eine wichtige Heuristik objektorientierten Designs befolgt, nämlich das so genannte *Dependency Inversion Principle* (siehe [Martin96b] und [Meade00]). Dieses *Prinzip der umgekehrten Abhängigkeit* besagt:

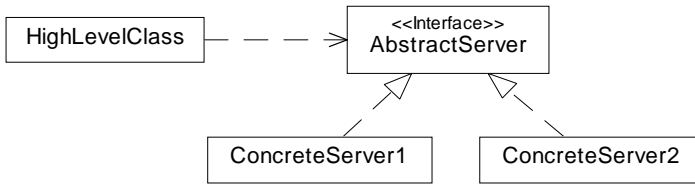


Abb. 6-1
 Dependency Inversion
 Principle

- High-Level-Module sollen nicht von Low-Level-Modulen abhängen – der Log-Server nicht vom Datei-Logger. Abhängigkeiten sollen ausschließlich zu *Abstrahierungen* (Interfaces) bestehen.
- Abstrahierungen sollen ihrerseits nicht von Details abhängen, sondern die Details von den Abstrahierungen.

Abbildung 6-1 zeigt dieses einfache Prinzip anhand eines Abhängigkeitsdiagramms der Klasse `HighLevelClass` von einem abstrakten Interface `AbstractServer`. Die beiden Implementierungen `ConcreteServer1` und `ConcreteServer2` hängen wiederum nur von diesem Interface ab.

Eine weitere Errungenschaft ist, dass wir *die korrekte Interaktion des Log-Servers mit seinem Logger* auf einfache Art und Weise testen können. Durch Verfolgung des einen Ziels – nämlich unseren Log-Server testbar zu machen – gab es noch ein zweites gratis dazu: ein verändertes Design und zwar eines, das die Abhängigkeiten verringert und die Erweiterbarkeit vergrößert. Wir haben jedoch (noch) keinen Log-Server, der wirklich in eine Datei schreibt. Doch das ist bei unserem jetzigen Wissensstand ein Klacks! Zunächst der Test:

Designverbesserung

```

import java.io.*;
public class FileLoggerTest extends TestCase {
    private final String TEMPFILE = "C:\\temp\\test.txt";
    public FileLoggerTest(String name) {
        super(name);
    }
    public void testLogLine() throws IOException {
        FileLogger logger = new FileLogger(TEMPFILE);
        logger.logLine("Zeile 1");
        logger.logLine("Zeile 2");
        logger.close();
        BufferedReader reader = new BufferedReader(
            new FileReader(TEMPFILE));
        assertEquals("Zeile 1", reader.readLine());
        assertEquals("Zeile 2", reader.readLine());
        assertNull("Dateiende erreicht",
            reader.readLine());
        reader.close();
    }
}
  
```

```
    }
}
```

Der Test besitzt noch eine kleine Unschönheit, nämlich die Abhängigkeit von einem absoluten Dateipfad; aber darauf kommen wir später zurück (siehe Kapitel 6.10). Die Implementierung der Klasse `FileLogger` ist mit Hilfe des Tests jetzt auch kein Hexenwerk mehr:

```
import java.io.*;
public class FileLogger implements Logger {
    private PrintWriter writer;
    public FileLogger(String filename) throws IOException {
        writer = new PrintWriter(
            new FileOutputStream(filename));
    }
    public void close() {
        writer.close();
    }
    public void logLine(String logMessage) {
        writer.println(logMessage);
    }
}
```

Ziel teilweise erreicht

Die Abhängigkeit der Tests vom Dateisystem wurde zwar nicht vollständig eliminiert, aber auf einen einzigen reduziert. Nämlich genau jenen, der die Zusammenarbeit mit Dateien verifiziert.

6.4 Wir bauen an

Hat sich die Mühe nun wirklich gelohnt? Betrachten wir ein paar Erweiterungen des kleinen Frameworks und überlegen, welche Auswirkungen diese auf unsere Tests haben könnten:

Zusätzlicher Parameter

Erweiterung 1: Wir erweitern unser Logging-Interface um eine `log`-Methode mit zusätzlichem Parameter `module`, der angibt, welches Modul der Anwendung gerade loggt.

Auswirkung: Eine zusätzliche Testmethode und ein wenig Refactoring im `LogServerTest`, etwa so:

```
public class LogServerTest extends TestCase {
    private LogServer logServer;
    private DummyLogger logger;
    public LogServerTest(String name) {
        super(name);
    }
    protected void setUp() {
```

```

    logger = new DummyLogger();
    logServer = new LogServer(logger);
}
public void testLoggingWithMbdule() {
    logServer.log(0, "Erste Zeile", "test");
    assertEquals("test(0): Erste Zeile",
        logger.getLogString(0));
}
public void testSimpleLogging() {
    logServer.log(0, "Erste Zeile");
    logServer.log(1, "Zweite Zeile");
    logServer.log("Dritte Zeile");
    assertEquals("(0): Erste Zeile",
        logger.getLogString(0));
    assertEquals("(1): Zweite Zeile",
        logger.getLogString(1));
    assertEquals("(2): Dritte Zeile",
        logger.getLogString(2));
}
}
}

```

Die Tests für die eigentlichen Logger bleiben unberührt.

Erweiterung 2: Wir erlauben, dass ein Log-Server mehrere Logger beherbergen kann, an die er alle Log-Nachrichten verteilt.

Mehrere Logger pro Server

Auswirkung: Ein paar Tests hier, um das Hinzufügen und Entfernen der Logger zu testen, und ein paar Tests da, um zu überprüfen, dass auch jeder Logger alle Nachrichten erhält. Selbst diese größere funktionale Erweiterung lässt die Tests für die eigentlichen Logger unberührt.

Ein zusätzlicher Vorteil bei der Separierung von LogServer und Logger und der Einführung einer Dummy-Implementierung ist, dass wir im Fall einer *Test-Failure* das Problem im Quellcode sehr genau eingrenzen können. Wir wissen nämlich, dass unser Log-Server die ganze Schuld trägt, und nicht etwa ein Fehler beim Zugriff auf das Dateisystem zum Scheitern des Tests geführt hat.

Eingrenzbarkeit von Fehlern

6.5 Endoskopisches Testen

In der XP-Szene werden Dummy-Objekte meist *Mock-Objekte* genannt. In dem empfehlenswerten Artikel von [Mackinnon00] über *Endo-Testing* – eine Anspielung auf endoskopische Operationstechniken – wird dieser Begriff das erste Mal verwendet. Es geht also um das »Testen von innen« durch die Einschleusung eines Testmediums: das

Testen von innen

Dummy-Objekt. Der Dummy-Logger hat es schließlich ermöglicht, das korrekte Verhalten des Log-Servers zu überprüfen, ohne dass wir dazu die Innereien des Log-Servers sichtbar machen mussten.

Mock-Objekte, die den Namen verdienen, gehen noch einen Schritt weiter als die hier bislang betrachteten Dummy-Objekte. Mock-Objekte holen den Großteil des eigentlichen Testcodes zu sich. Bauen wir unseren Dummy-Logger zu einem Mock-Logger um:

```
import java.util.*;
public class MckLogger implements Logger {
    private List expectedLogs = new ArrayList();
    private List actualLogs = new ArrayList();
    public void addExpectedLine(String logString) {
        expectedLogs.add(logString);
    }
    public void logLine(String logString) {
        actualLogs.add(logString);
    }
    public void verify() {
        if (actualLogs.size() != expectedLogs.size()) {
            Assert.fail("Expected " + expectedLogs.size() +
                " log entries but encountered " +
                actualLogs.size());
        }
        for (int i = 0; i < expectedLogs.size(); i++) {
            String expectedLine =
                (String) expectedLogs.get(i);
            String actualLine =
                (String) actualLogs.get(i);
            Assert.assertEquals(expectedLine, actualLine);
        }
    }
}
```

Typisch für ein Mock-Objekt sind die beiden Methoden `addExpectedLine()` und `verify()`. Während die erste dazu dient, das *erwartete Verhalten* unseres »Klienten« (des Log-Servers) zu setzen, führt die zweite das eigentliche Überprüfen des korrekten Verhaltens *am Ende des Tests* durch. Natürlich muss sich nun auch unsere Testklasse anpassen:

```
public class LogServerTest extends TestCase {
    private LogServer logServer;
    private MckLogger logger;
    public LogServerTest(String name) {
        super(name);
    }
}
```

```

protected void setUp() {
    logger = new MckLogger();
    logServer = new LogServer(logger);
}
public void testLoggingWithModule() {
    logger.addExpectedLine("test(0): Erste Zeile");
    logServer.log(0, "Erste Zeile", "test");
    logger.verify();
}
public void testSimpleLogging() {
    logger.addExpectedLine("(0): Erste Zeile");
    logger.addExpectedLine("(1): Zweite Zeile");
    logger.addExpectedLine("(2): Dritte Zeile");
    logServer.log(0, "Erste Zeile");
    logServer.log(1, "Zweite Zeile");
    logServer.log("Dritte Zeile");
    logger.verify();
}
}

```

Bislang haben wir nur ein wenig Code hin- und hergeschoben. Ein paar Veränderungen unseres Mock-Loggers können daher nicht schaden:

```

import java.util.*;
public class MckLogger implements Logger {
    private List expectedLogs = new ArrayList();
    private List actualLogs = new ArrayList();
    public void addExpectedLine(String logString) {
        expectedLogs.add(logString);
    }
    public void logLine(String logLine) {
        Assert.assertNotNull(logLine);
        if (actualLogs.size() >= expectedLogs.size()) {
            Assert.fail("Too many log entries");
        }
        int index = actualLogs.size();
        String expectedLine =
            (String) expectedLogs.get(index);
        Assert.assertEquals(expectedLine, logLine);
        actualLogs.addElement(logLine);
    }
    public void verify() {
        if (actualLogs.size() < expectedLogs.size()) {
            Assert.fail("Expected " + expectedLogs.size() +
                " log entries but encountered " +

```

```

        actualLogs.size());
    }
}

```

Der Gewinn scheint subtil: Ein Teil des Verifikationscodes wurde aus der `verify()`- in die `logLine()`-Methode verlagert. Dies hat den Vorteil, dass das Feedback eines fehlerhaften Log-Eintrages nun unmittelbar erfolgt und nicht erst am Ende des Tests. Man kann dies gut sehen, wenn man beispielsweise die Zeile

```
logServer.log(0, "Erste Zeile");
```

zu

```
logServer.log(0, "Falsche erste Zeile");
```

abändert und mit dem Debugger verfolgt, wann die Exception `TestFailure` geworfen wird. Bei komplexen Testfällen kann diese genauere Lokalisierbarkeit eines Fehlers die Debugging-Zeit spürbar verkürzen. Eine weitere Änderung war die zusätzliche Zeile:

```
Assert.assertNotNull(logLine);
```

Das Testen dieser wichtigen Vorbedingung mit unserem »alten« `DummyLogger` hätte erfordert, sie für jede einzelne Log-Zeile gesondert einzufügen.

Wie man hier sieht, besteht der Fortschritt, den uns Mock-Objekte gegenüber einfachen Dummy-Objekten bringen, zu einem wichtigen Teil in der Vermeidung von dupliziertem Code. Dieser Vorteil wird umso größer, je mehr Mock-Objekte gebaut werden, da sich der Code, den wir für den Vergleich von erwartetem und wirklich erfolgtem Verhalten benötigen, von Mock-Objekt zu Mock-Objekt sehr ähnelt und daher in eigene Klassen ausgelagert werden kann.

Des Weiteren vergrößern Mock-Objekte die Kommunikationsfähigkeit unseres Codes. Werden sie so verwendet, wie in [Mackinnon00] vorgeschlagen, so ergibt sich ein standardisiertes Verwendungsmuster (*Pattern*), das den eigentlichen Testcode vereinfacht und damit lesbarer macht. Wie bei anderen Patterns auch lässt sich dadurch die Kommunikation zwischen all denen, die das Muster kennen, deutlich verbessern. Unsere leicht adaptierte Fassung des in [Mackinnon00] vorgeschlagenen »Pattern for unit testing« besteht aus folgenden Schritten, die das Aussehen eines einzelnen Unit Tests beschreiben:

1. Erzeuge die nötigen Mock-Objekte.
2. Setze, wenn nötig, den internen Zustand dieser Mock-Objekte.

3. Setze die Erwartungen in den Mock-Objekten.
4. Rufe den zu testenden Code mit den Mock-Objekten als Parameter auf.
5. Überprüfe, wenn angebracht, Zustandsänderungen in den zu testenden Objekten durch direkte Tests.
6. Verifiziere die Konsistenz der Mock-Objekte mittels `verify()`.

In [Mackinnon00] fehlt Punkt 5 ersatzlos, da die Autoren davon ausgehen, dass auch einfache Zustandsänderungen aus Konsistenzgründen am besten über Mock-Objekte getestet werden. Unsere Erfahrung zeigt jedoch, dass das direkte Abfragen bloßer Zustandsänderung häufig viel einfacher ist, als entsprechende Mock-Objekte zu bauen, die nur an dieser Stelle Verwendung fänden. Auch hier gibt es wieder mal keine feste Regel; ob die Verwendung von Mock-Objekten eine Verbesserung unseres Codes zur Folge hat oder nicht, muss im Einzelfall entschieden werden.

*Mock-Objekte für
Zustandstests?*

Häufig bietet sich zunächst der direkte Zustandstest als einfachste Möglichkeit an. Werden die inneren Objekte später komplizierter und stellen wir Codeduplikation in unseren Tests fest, so führen wir nach und nach entsprechende Interfaces und zugehörige Mock-Implementierungen ein. Auch beim Testen ist iteratives Vorgehen die Methode der Wahl.

Fertige Mock-Objekte

Die Tatsache, dass Mock-Objekte eine Standardtechnik des Test-First-Ansatzes sind, hat einige interessante Software entstehen lassen:

- In [Mackinnon00] wird eine Bibliothek von *Expectation Classes* erwähnt, die mittlerweile auch frei verfügbar ist [URL:MockObjects] und uns eine Menge Implementierungsaufwand abnehmen kann.
- Das Tool MockMaker [URL:MockMaker] dient zum Erzeugen von Quellcode für Mock-Objekte, die auf den erwähnten »Expectation Classes« aufbauen. Das Werkzeug geht von einem Interface aus und generiert Klassen, die sowohl die Spezifikation des erwarteten Verhaltens erlauben als auch die Rückgabe vorbestimmter Funktionswerte.
- Ähnlich wie MockMaker stellt MockCreator [URL:MockCreator] eine Umgebung zum automatischen Erzeugen von Mock-Objekten bereit. Das Tool ist zurzeit noch ausschließlich für IBMs Entwicklungsumgebung *Visual Age for Java* [URL:VAJava] erhältlich.

- Ab JDK 1.3
- Einen anderen Ansatz verfolgen die *EasyMocks* [URL:EasyMock]. Anstatt sich für jeden Verwendungszweck selbst Interfaces und Mock-Implementierungen schreiben zu müssen, erlauben diese »einfachen Mocks« das erwartete Verhalten programmatisch zu bestimmen und sich so unter Umständen einiges an Programmieraufwand zu sparen.
 - Im Bereich der Sanitäranlagen und des Heizungsbaus machen sich Mocks auch bereits breit [URL:MockSanitaer].

Ein Blick auf diese frei verfügbaren Bibliotheken und Tools ist jedem empfohlen, der Mock-Objekte nicht nur sporadisch einsetzt.

6.6 Testen von Grenzwerten und Exceptions

In Kapitel 4.4 haben wir begründet, dass Testfälle sich in besonderem Maße auf die Grenzbereiche von Ein- und Ausgabe konzentrieren sollten. Vorausgesetzt man kennt diese Grenzwerte, kann man diese Tests genau dann recht leicht durchführen, wenn sie der zu testenden Methode als Parameter übergeben werden.

Ein Beispiel: Zum Test bereit steht unsere Klasse `TextFormatter`, die einen von uns übergebenen Text zeilenweise umformatieren soll. Unsere Grenzbedingung ist, dass Zeilen von maximal 32 Zeichen verarbeitet werden; längere Zeilen werden abgeschnitten. Unser grenzwertbasierter Test lautet daher:

```
public void testLongLines() {
    TextFormatter formatter = new TextFormatter();
    String line32 = " abcdefg hijklm opqrs tuvwx";
    String line33 = " abcdefg hijklm opqrs tuvwx";
    assertEquals("abcdefg hijklm opqrs tuvwx",
        formatter.formatLine(line32));
    assertEquals("abcdefg hijklm opqrs tuvwx",
        formatter.formatLine(line33));
}
```

So weit, so gut. Nun fällt uns jedoch eine weitere Anforderung an unseren Textformatierer ein: Er soll keine einzelne Zeilen formatieren, sondern komplette Dateien und das Ergebnis seinerseits in eine Datei schreiben; d.h., unser öffentliches Interface ist nicht mehr zeilenbasiert, sondern dateibasiert. Wir haben also wieder ein ähnliches Problem wie mit unserem Log-Server (siehe Kapitel 6.3), nur dass wir außer der Ausgabedatei auch noch eine Eingabedatei mit entsprechendem Inhalt vor dem Test erzeugen müssen. Verwenden wir jedoch ein Pärchen von Mock-Klassen, ein `MockLineReader` und ein `MockLineWri-`

ter, die jeweils das Interface `LineReader` bzw. `LineWriter` implementieren, dann ist der Test gemäß dem vorgestellten Mock-Pattern leicht hinzuschreiben:

```
public void testLongLines() {
    MockLineReader reader = new MockLineReader();
    String line32 = " abcdefg  hijklmn opqrs tuvwx";
    String line33 = " abcdefg  hijklmn opqrs tuvwxz";
    reader.addLineToBeRead(line32);
    reader.addLineToBeRead(line33);
    MockLineWriter writer = new MockLineWriter();
    writer.addExpectedLine("abcdefg hijklmn opqrs tuvwx");
    writer.addExpectedLine("abcdefg hijklmn opqrs tuvwx");
    TextFormatter formatter = new TextFormatter();
    formatter.format(reader, writer);
    writer.verify();
}
```

Während in diesem konstruierten Fall das Erzeugen entsprechender Testdateien noch denkbar wäre, gibt es andere Fälle, in denen Grenzbedingungen kaum anders als durch Dummy- bzw. Mock-Objekte zu erreichen sind. Man denke beispielsweise an den Zugriff auf einen Server, der sich mit seiner Antwort nur x Sekunden Zeit lassen darf, bevor eine `TimeoutException` vom Client geworfen werden soll. Wie bringe ich einen entfernten Server dazu, genau $x-1$ bzw. $x+1$ Sekunden mit seiner Antwort zu warten, damit ich die korrekte Reaktion meines Client in diesen Grenzfällen überprüfen kann? Eine `MockServer`-Klasse, bei der ich nicht nur die gewünschte Antwort, sondern auch die Verzögerungszeit konfigurieren kann, macht diesen Test zu einem Kinderspiel.

*Unkontrollierbare
Grenzwerte*

Ein ähnlich gelagerter Fall wie Grenzwerte sind *Exceptions*. Denken wir an unser Beispiel in Kapitel 6.1, verändern jedoch das Interface unserer `getRateFromTo()`-Methode zu:

*Unkontrollierbare
Exceptions*

```
public double getRateFromTo(String from, String to)
    throws ServerNotAvailableException;
```

Es dürfte einiger Überzeugungskraft bedürfen, unseren Anbieter von Finanzinformationen dazu zu bringen, immer wenn wir testen, den Wechselkursserver für wenige Millisekunden vom Netz zu nehmen. Aber mit einer kleinen Änderung der Klasse `DummyProvider` können wir uns die nötigen Verhandlungen für wichtigere Aufgaben sparen:

```
public class DummyProvider extends ExchangeRateProvider {
    private double dummyRate;
    private boolean serverAvailable = true;
    public DummyProvider(double dummyRate) {
```

```

        this.dummyRate = dummyRate;
    }
    public double getRateFromTo(String from, String to)
        throws ServerNotAvailableException {
        if (!serverAvailable) {
            throw new ServerNotAvailableException("Test");
        }
        return dummyRate;
    }
    public void setServerAvailable(boolean isAvailable) {
        serverAvailable = isAvailable;
    }
}

```

Wir hätten gerne, dass unser EuroCalculator, falls der Wechselkursserver nicht zur Verfügung steht, einen Wechselkurs von 1.0 benutzt (ob diese Vorgabe sinnvoll ist, soll hier nicht diskutiert werden ;-). Der Test hierfür könnte so aussehen:

```

public void testServerNotAvailable() {
    //Kurs des DummyProvider egal, da er Exception wirft
    DummyProvider provider = new DummyProvider(1.1324);
    provider.setServerAvailable(false);
    double result = new EuroCalculator().
        valueInEuro(1.5, "USD", provider);
    assertEquals(1.5, result, ACCURACY);
}

```

*Soll jede mögliche
Exception getestet
werden?*

Auf diese Art und Weise ermöglichen uns Mock-Objekte, korrektes Verhalten in Ausnahmesituationen und Grenzfällen zu testen, die wir ohne sie außen vor lassen müssten. Jedoch auch hier gilt wieder: Nur weil es möglich ist, wird nicht jedes Objekt mit allen denkbaren und undenkbaeren Exceptions bombardiert. Wollten wir beispielsweise alle Stellen unseres Programms, an denen eine NullPointerException auftreten kann, auch dahingehend testen, kämen wir zu nichts anderem mehr. Ein Abwägen zwischen Aufwand und Nutzen ist hier ganz besonders angebracht.

6.7 Wie kommt der Test zum Mock?

In den bisherigen Beispielen hatten wir kein größeres Problem dabei, dem Testobjekt das Dummy- bzw. Mock-Objekt unterzuschieben. Während in unserem Euro-Rechner das Interface der valueInEuro()-Methode die Übergabe eines ExchangeRateProvider vorsah, konnte man dem LogServer einen Logger im Konstruktor übergeben. Ob man

die eine oder andere Möglichkeit wählt, hängt von unterschiedlichen Punkten ab:

- Nehmen wir das Helferobjekt als Parameter in die entsprechenden Methoden auf, wie im EuroCalculator, so können wir das OUT mit unterschiedlichen Instanzen des Helfers immer wieder verwenden. Dafür müssen wir uns aber auch bei jedem Methodenaufruf überlegen, woher wir die richtige Helferinstanz nehmen – ohne sie zu stehlen.
- Wird jedoch das Helferobjekt im Konstruktor des Testobjekts übergeben, wie bei unserem Log-Server, so haben wir ein für alle Mal die Überlegung, welche Instanz wir wann benötigen, vom Hals. Dies ist vor allem dann sinnvoll, wenn der Helfer in mehreren Methoden des Objekts benötigt wird und darüber hinaus für die gesamte Lebenszeit des Objekts unverändert bleibt.

Beide Möglichkeiten erlauben uns das einfache Ersetzen eines Helfer- oder Serverobjekts durch Dummy- bzw. Mock-Objekte. Existierende Programme sind jedoch meist geschrieben worden, ohne die Anforderungen an das Testen zu berücksichtigen, d.h., die intern verwendeten Objekte sind fest verdrahtet. Häufig werden bei der Initialisierung eines Objekts die benötigten Helferobjekte erzeugt und in Instanzvariablen festgehalten. Solche Objekte kann man relativ leicht durch das Anbieten zusätzlicher Methoden zum Austausch dieser Helfer testbar machen. Unser Euro-Rechner sähe dann etwa so aus:

*Nachträgliche
Modifikation*

```
public class EuroCalculator {
    private ExchangeRateProvider provider =
        new ExchangeRateProvider();
    public void setProvider(
        ExchangeRateProvider newProvider) {
        provider = newProvider;
    }
    double valueInEuro(double amount, String currency) {...}
}
```

Dementsprechend muss unsere Testklasse in den Testmethoden den richtigen Provider durch einen Dummy-Provider explizit ersetzen:

```
public void testUSD2EUR() {
    ExchangeRateProvider dummyProvider =
        new DummyProvider(1.1324);
    EuroCalculator calculator = new EuroCalculator();
    calculator.setProvider(dummyProvider);
    double result = calculator.valueInEuro(1.5, "USD");
}
```

```

    assertEquals(1.6986, result, ACCURACY);
}

```

Man sieht, dass der Test länger und schlechter lesbar wird. Zudem besteht die Gefahr, dass man bei komplexeren Testszenarien vergisst, die eine oder andere Komponente durch ihr Mock-Pendant zu ersetzen. Dies kann zu subtilen und schwer ergründbaren Failures oder Errors im Test führen. Vorteilhaft hingegen ist, dass der Applikationscode nichts über das Austauschen des Provider-Objekts wissen muss.

Noch schwieriger gestaltet sich das Testen mittels Attrappen, wenn das Serverobjekt an jeder Stelle seiner Verwendung neu erzeugt wird, um beispielsweise Synchronisationsprobleme zu umgehen. Unsere (vereinfachte) `valueInEuro()`-Methode sähe dann folgendermaßen aus:

```

public double valueInEuro(double amount, String currency) {
    ExchangeRateProvider provider =
        new ExchangeRateProvider();
    double exchangeRate =
        provider.getRateFromTo(currency, "EUR");
    return amount * exchangeRate;
}

```

In diesem Fall ist unser Konstruktoraufruf `new ExchangeRateProvider()` nichts anderes als eine implizite Konstante und spielt daher auch die gleiche unschöne Rolle bei der Wartung und beim Testen der Software: Eine Änderung der Konstanten erfordert die Suche nach allen im Code verteilten Verwendungsstellen – ein erster Schritt auf dem Weg in die »Wartungsfalle«.

Die Entschlossenheit, eine solche Methode vernünftig zu testen, erfordert einen größeren Umbau: Der `ExchangeRateProvider` muss auf die eine oder andere Weise austauschbar gemacht werden. Wollen wir den Provider dennoch nicht als zusätzlichen Parameter übergeben, bleibt noch ein letzter Trick: Statt der Instanz selbst übergeben wir ein *Factory*-Objekt, das wir dann im Testfalle wiederum durch eine *Mock-Factory* ersetzen können. Dafür wird ein Interface mit zwei Implementierungen benötigt:

```

public interface ProviderFactory {
    public ExchangeRateProvider createProvider();
}

public class RealProviderFactory implements ProviderFactory {
    public ExchangeRateProvider createProvider() {
        return new ExchangeRateProvider();
    }
}

```

Letzte Rettung: Factory

```

    }
}

public class MockProviderFactory implements ProviderFactory {
    private double rate;
    public MockProviderFactory(double rate) {
        this.rate = rate;
    }
    public ExchangeRateProvider createProvider() {
        return new DummyProvider(rate);
    }
}

```

In der `EuroCalculator`-Klasse kann nun das entsprechende Factory-Objekt entweder im Konstruktor übergeben werden oder mittels einer Setter-Methode austauschbar sein. Ein typischer Testfall wäre dann:

```

public void testUSD2EUR() {
    ProviderFactory factory =
        new MockProviderFactory(1.1324);
    EuroCalculator calculator =
        new EuroCalculator(factory);
    double result = calculator.valueInEuro(1.5, "USD");
    assertEquals(1.6986, result, ACCURACY);
}

```

Die Testbarkeit geht jedoch hier auf Kosten der Einfachheit und Lesbarkeit: Die zusätzliche Umleitung ist schwieriger zu verstehen als ein unmittelbarer Konstruktoraufruf. Im Austausch gewinnen wir jedoch die Unabhängigkeit des `EuroCalculator` von einer konkreten `ExchangeRateProvider`-Implementierung.

*Nachteile der
»Fabriklösung«*

6.8 Böse Singletons

Einen Sonderfall des Wie-bringe-ich-den-Dummy-ins-Objekt-Problems stellen *Singletons* dar. Die Popularität von *Entwurfsmustern* [Gamma95] unter heutigen Programmierern hat dazu geführt, dass vor allem die einfachen Muster sehr häufig angewandt werden, ohne deren Nachteile zuvor abzuwägen. »Singleton« stellt das einfachste der verbreiteten Muster dar. Es soll sicherstellen, dass nur eine Instanz einer bestimmten Klasse erzeugt wird, welche zudem von allen Objekten im System leicht angesprochen werden kann. Praktisch scheint dieses Vorgehen bei systemweit verwendeten Objekten, wie z.B. Ressourcen-Verwalten, Datenbanken, Voreinstellungen, und überhaupt allen global

nützlichen Objekten, die der Entwickler gerne allzeit und überall zur freien Verfügung hat.

*Singleton == globale
Variable*

Doch Vorsicht, Singletons sind, ohne Nachdenken angewandt, nichts anderes als die globalen Variablen objektorientierter Systeme – inklusive all ihrer Nachteile, z.B. der Anfälligkeit für Nebeneffekte und dem Aufweichen der Kapselung [Rainsberger01]. Auch trifft man bei der Verwendung von Singletons in Applikationsservern manchmal auf unerwartete Probleme, ausgelöst durch die Verwendung von Threads und applikationseigener Class Loader. Doch wir wollen hier keine Grundsatzdiskussion zum Thema »*Singletons are evil*« führen², sondern ein testspezifisches Problem untersuchen: Da es von jeder Singleton-Klasse während der Laufzeit eines Programms nur genau eine Instanz gibt und auf diese Instanz nur lesend zugegriffen werden kann, stellt sich die Frage, wie man, wenn nötig, diese Instanz gegen eine Mock-Instanz austauscht.

Folgende Lösung scheint denkbar (unsere Singleton-Klasse abstrahiert diesmal von allen sinnvollen Tätigkeiten):

```
public class Singleton {
    protected static Singleton instance = null;
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Damit kann unser Mock-Singleton als Unterklasse eine Initialisierungsmöglichkeit für Testzwecke anbieten:

```
public class MockSingleton extends Singleton {
    public static void initMockSingleton() {
        instance = new MockSingleton();
    }
}
```

In unseren Tests müssen wir jetzt dafür sorgen, dass die `initMockSingleton()`-Methode zu Beginn des Tests oder im Setup ausgeführt wird:

```
public class MockSingletonTest extends TestCase {
    public MockSingletonTest(String name) {
        super(name);
    }
}
```

2. Diese findet sich ausführlich unter [URL:CoSingle] und [URL:WikiSAE].

```
public void testInitialization() {
    MockSingleton.initMockSingleton();
    assertTrue(Singleton.getInstance()
               instanceof MockSingleton);
}
}
```

Dies ist zwar ein gangbarer Weg, wenn wir uns partout nicht von unserem Singleton trennen wollen, er hat aber auch einige Nachteile:

- Wir müssen stets gewährleisten, dass jeder Test auch wirklich alle benötigten Singletons zu Beginn in den korrekten Testzustand bringt und danach wieder durch das Original ersetzt. Vergisst man dies, beispielsweise für ein neu hinzugekommenes Singleton, so kann dies zu schmerzhaft langen Debugging-Sessions führen.
- Manchmal benötigt jeder Test eine individuell konfigurierte Instanz unseres Mock-Singletons. Dies kann bewirken, dass sich immer mehr Initialisierungscode in der Singleton- oder MockSingleton-Klasse anhäuft.

Beide Probleme lassen sich dadurch angehen, dass man eine *Setter-Methode* für das Singleton anbietet. Damit ist das Singleton jedoch kein echtes Singleton mehr, sondern zu einer gefährlichen Mutation geworden: einem global zugänglichen Zustandsbehälter, anfällig für Nebeneffekte aller Art.

Ein Ausweg aus dieser Singleton-Krise ist möglich, aber nicht kostenfrei. Hinter den meisten Singletons versteckt sich nämlich ein anderes Konzept: Wir benötigen Objekte, die innerhalb eines gewissen Kontextes gleich bleiben und nur einmal vorhanden sind. Dieser Kontext kann unser *System* sein oder unser *User* oder vielleicht auch unsere *Session*. Warum nicht also ein *Systemobjekt* zur Verfügung stellen bzw. ein User- oder ein Session-Objekt, das uns Zugriff auf die Objekte gewährt, die wir andernfalls zu Singletons gemacht hätten. Dieses Systemobjekt können wir dann entweder allen Objekten, die es benötigen, bei ihrer Erzeugung mit auf den Weg geben – oder, weil wir kompromissbereit sind, zu einem Singleton machen. So bleiben wir am Ende nur noch auf einem einzigen Singleton sitzen, das wir dann jedoch genau im Auge behalten müssen.

Auch in dieser Diskussion haben wir gesehen, dass der Wunsch nach lokaler Testbarkeit gängige Programmiermuster in Frage stellt und uns manchmal auf Designprobleme hinweist, die wir andernfalls einfach übersehen oder zumindest ignoriert hätten. Erstellt man Software nach dem *Test-First-Ansatz*, so lassen sich die meisten Schwierigkeiten von vorneherein vermeiden. Versucht man jedoch, eine beste-

Singleton-Alternativen

hende Anwendung im Nachhinein mit einem dichten Netz von Entwicklertests auszustatten, so gerät man an einen Punkt, an dem die Implementierung dieser Tests eine umfangreiche Restrukturierung des Programms voraussetzt. Wir wagen es schon kaum mehr zu sagen: Auch hier ist ein Abwägen zwischen Kosten und Nutzen angesagt, bevor man sich an monatelangen Umbauarbeiten versucht.

6.9 Leicht- und schwergewichtige Mocks

Bislang haben wir zwei Ansätze gesehen, ein Dummy-Objekt zu bauen:

1. Indem wir es als Unterklasse von der richtigen Implementierung ableiten, wie beispielsweise der `DummyRateProvi` der.
2. Indem sowohl die richtige als auch die Mock-Klasse das gleiche Interface implementieren.

Während die erste Variante die einfachere ist, da wir kein eigenes Interface implementieren müssen, birgt sie gewisse Gefahren. So passiert es relativ schnell, dass man bei einer Änderung der Signatur der richtigen Klasse vergisst, die Mock-Klasse anzupassen. Das OOT ruft nun die neue Methode auf und der Test wird eine unerwartete und häufig schwer zu ergründende Failure erzeugen.

Die zweite Variante dagegen erzeugt zusätzlichen Programmieraufwand, da sie zunächst einmal die Extraktion des Interfaces erfordert und auch die Implementierung aller Methoden in der Mock-Klasse. Änderungen der Signatur ziehen dementsprechend auch Änderungen an (mindestens) drei unterschiedlichen Stellen nach sich: dem Interface selbst, der richtigen Implementierung und aller Mock-Klassen. Dennoch bevorzugen wir meist diese Variante, da das Interface zusätzlich eine dokumentierende Funktion ausübt und die zu betrachtende Komplexität spürbar verringert wie bei unserem Übergang vom `DummyPrintWriter` zum `DummyLogger` (siehe Kapitel 6.3). Zudem kann der Aufwand zur Synchronisation zwischen Interface und Implementierung durch eine entsprechend ausgestattete Entwicklungsumgebung oder die Verwendung der erwähnten Easy-Mocks minimiert werden.

Das UML-Diagramm in Abbildung 6-2 soll den Vollausbau unseres kleinen Musters zur Einführung von Mock-Objekten verdeutlichen. Die Idee dahinter ist, dass die Klasse `AbstractMock` für alle im Interface deklarierten Methoden eine `NotImplementedException` wirft. Konkrete Mock-Klassen leiten von ihr ab und überschreiben nur die interessanten Methoden. Gemeinsamkeiten der konkreten Mock-

Objekte lassen sich zudem nach oben in `AbstractMock` verschieben, um auch in den Tests Codeduplikation zu vermeiden.

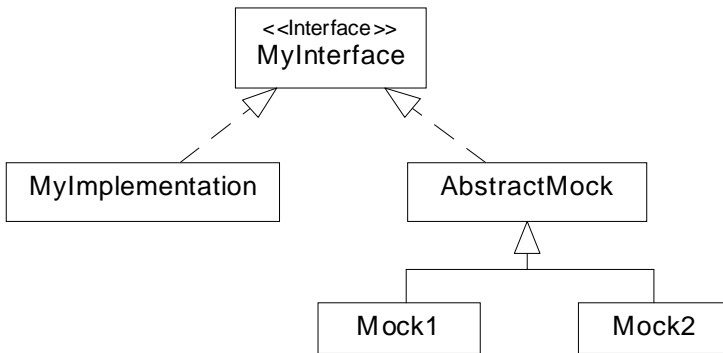


Abb. 6-2

Mock-Objekte-Hierarchie

Betrachten wir uns dieses Vorgehen an einem Beispiel: Unser Logging-Framework soll dahingehend erweitert werden, dass einzelne Logger im laufenden Betrieb ausgetauscht werden können. Dies erfordert zunächst einmal, dass in der Klasse `LogServer` die Methode

```
public void setLogger(Logger newLogger) {...}
```

eingefügt wird. Darüber hinaus muss unser Logger-Interface um die Methode

```
public void close();
```

erweitert werden, um zu gewährleisten, dass ein zu ersetzender Logger die Möglichkeit erhält, vor seinem Ruhestand die nun nicht mehr benötigten Ressourcen freizugeben. Vor der Implementierung schreiben wir zwei Tests für diese neue Funktionalität:

```
public void testSetLogger() {
    MockLogger newLogger = new MockLogger();
    logServer.setLogger(newLogger);
    newLogger.addExpectedLine("1): Test");
    logServer.log(1, "Test");
    newLogger.verify();
}
```

```
public void testCloseOnSetLogger() {
    logger.setCloseExpected();
    logServer.setLogger(new MockLogger());
    logger.verify();
}
```

Wir fordern im zweiten Test, dass einem Logger, bevor er ersetzt wird, eine `close()`-Nachricht geschickt wird. Folgende Erweiterungen sind dafür in der `MockLogger`-Klasse nötig:

```
public class MockLogger implements Logger {
    /.../
    private boolean closeExpected = false;
    private boolean closeInvoked = false;
    public void close() {
        closeInvoked = true;
    }
    public void setCloseExpected() {
        closeExpected = true;
    }
    public void verify() {
        if (closeExpected) {
            Assert.assertTrue(
                "close() should have been called", closeInvoked);
        }
        if (actualLogs.size() < expectedLogs.size()) {
            Assert.fail("Expected " + expectedLogs.size() +
                " log entries but encountered " +
                actualLogs.size());
        }
    }
}
```

Die auffälligste Änderung hat in der `verify()`-Methode stattgefunden, die um eine zusätzliche Überprüfung ergänzt wurde. Man kann sich vorstellen, wie sich `verify()` im Zuge aller zukünftigen Erweiterungen der Logger-Funktionalität zu einem Sammelbecken zahlreicher möglicher und unmöglicher Validierungsfunktionen entwickelt, von denen jedoch in jedem einzelnen Test nur ein oder zwei benötigt werden. Durch die Einführung eines abstrakten Mock-Loggers und diverser Unterklassen (z.B. `TestCloseMockLogger` und ein `TestLinesMockLogger`) ließe sich diese Vermischung umgehen, dafür hätten wir auf Dauer vermutlich mit einer sich stetig vermehrenden Zahl von Mock-Klassen zu kämpfen, von denen die meisten nur ein einziges Mal Verwendung fänden.

*Tricks zur
Klassenbegrenzung*

Diese übermäßige Vermehrung von Klassen lässt sich in Java durch zwei kleine Tricks vermeiden:

1. Wird eine bestimmte Mock-Implementierung nur für einen einzelnen Test gebraucht, so erzeugt man sie als *anonyme Klasse* direkt in der Testmethode.

2. Wird eine bestimmte Mock-Implementierung nur innerhalb einer Testklasse benötigt, dann legt man sie als *innere Klasse* der Testklasse an.

Trick Nummer 1 ist vor allem geeignet, um Methoden zu simulieren, die im Test feste Werte zurückgeben. Komplexere Validierungsfunktionen sind in anonymen Klassen nur durch leichte bis mittelschwere Verrenkungen zu erreichen, da Java diesen leichtgewichtigen Klassen einige Einschränkungen auferlegt. Unser Euro-Rechner bietet sich als typisches Beispiel für diese Technik an. Seine Tests kämen ohne die Klasse `DummyProvider` aus, dafür wäre jeder einzelne Test etwas schwerfälliger:

```
public void testUSD2EUR() {
    ExchangeRateProvider provider =
        new ExchangeRateProvider() {
            public double getRateFromTo(String from, String to) {
                return 1.1324;
            }
        };
    double result = new EuroCalculator().
        valueInEuro(1.5, "USD", provider);
    assertEquals(1.6986, result, ACCURACY);
}
```

Trick Nummer 2 ist nichts anderes als die Reduzierung der Sichtbarkeit der Mock-Klasse; die Übertragung dieses Prinzips auf einen `TestLineMockLogger` überlassen wir dem geneigten Leser. Ob man eine Mock-Klasse im konkreten Fall als innere oder »normale« Klasse implementiert, hängt nicht zuletzt von der Unterstützung dieses Java-Features durch die verwendete Entwicklungsumgebung ab.

Zu guter Letzt soll auch die Möglichkeit nicht unerwähnt bleiben, die Testklasse selbst als Mock-Objekt zu nutzen, indem man sie das entsprechende Interface selbst implementieren lässt. Dies ist eine leicht abgewandelte Form des Ansatzes mit der inneren Klasse, jedoch ohne die Möglichkeit, von einer bestehenden abstrakten Klasse zu erben. Dieses Vorgehen wird in [Feathers00] als das »*Self*«-*Shunt*-Testmuster beschrieben.

Und welche dieser zahlreichen Möglichkeiten wird für die Praxis empfohlen? Am besten halten wir uns auch hier an die XP-Regel: »Tue das Einfachste, das möglicherweise funktioniert!«³ Im `EuroCalculator`

*Evolution eines
Mock-Objekts*

3. »We [...] generally do [...] the simplest thing that could possibly work.« [Jeffries00], S. 74.

tor-Beispiel würden wir mit einer anonymen Klasse beginnen und dann beim zweiten Test, der die `getRateFromIo()`-Methode überschreibt, auf eine innere Klasse umschwenken. Sobald wir diese Klasse außerhalb benötigen oder sobald wir feststellen, dass die erschwerte Handhabbarkeit der inneren Klasse den kleinen Vorteil der reduzierten Sichtbarkeit aufhebt, extrahieren wir die innere Klasse und machen sie zu einem vollwertigen Mitglied unserer Java-Gesellschaft.

Ähnlich war die Entwicklung im Fall unseres `MockLogger`. Im ersten Ansatz hatten wir einen `DummyPrintWriter` als direkte Unterklasse von `PrintWriter`, um jedoch festzustellen, dass ein dediziertes Interface (Logger) den wahren Zweck des Objekts viel besser kommunizierte und auch den Code besser lesbar machte – trotz der Notwendigkeit, ein Interface und zwei Klassen neu zu programmieren. In einem späteren Schritt sind wir zu dem Schluss gekommen, dass es unseren Test vereinfacht, wenn wir die eigentliche Validierung aus der Testklasse in die Dummy-Klasse ziehen und so unseren `DummyLogger` zu einem `Mock-Logger` befördern.

Eine Erweiterung des Logging-Frameworks führte schließlich zu einer Erweiterung des Logger-Interfaces und im Zuge der Tests zum Wunsch nach unterschiedlichen `MockLogger`-Klassen. Das ist die Geburtsstunde unseres `AbstractMockLogger`:

```
public class AbstractMockLogger implements Logger {
    public static class NotImplemented
        extends RuntimeException {
    }
    public void close() {
        throw new NotImplemented();
    }
    public void logLine(String logMessage) {
        throw new NotImplemented();
    }
}
```

Diesen können wir dann, je nach Bedarf, in einer anonymen, inneren oder richtigen Klasse zu einem konkreten Mock-Logger-Objekt machen. Iteratives Vorgehen und Entwicklung unseres Testframeworks ersetzt auch hier das starre Festhalten an unumstößlichen Regeln.

6.10 Dateiattrappen

In Kapitel 6.3 haben wir einen Test für die `FileLogger`-Klasse geschrieben. Das Unschöne an diesem Test war, dass wir den Namen einer Testdatei als Konstante angeben mussten. Dies macht uns nicht nur vom verwendeten Dateisystem abhängig, sondern auch von Dingen wie den *Security*-Einstellungen und dem vorhandenen Plattenplatz. Seit JDK 1.2 stellt uns die Klasse `java.io.File` zwar einen Mechanismus zur Verfügung, um temporäre Dateien unabhängig von einem konstanten Dateipfad zu erzeugen, aber die Abhängigkeit von der Verfügbarkeit des Filesystems und dessen Zugriffsrechten bleibt auch mit dieser Methode bestehen.

Warum nutzen wir nicht unser neues Wissen über Dummy- und Mock-Klassen, um unsere eigene `MockFile`-Klasse zu programmieren? Leichter gesagt als getan, da uns die Entwickler des JDK ein paar Steine in den Weg zu diesem Ziel gelegt haben: Das größte Hindernis ist die Tatsache, dass die Klasse `java.io.File` nicht das hält, was der Name verspricht: Sie ist nämlich keineswegs eine Abstraktion aller Dinge, die wir gerne mit Dateien tun würden, sondern lediglich eine vom konkreten Filesystem unabhängige Abstraktion eines Dateinamen und seines Zugriffspfades.

Generische Mock-Dateien

Die eigentliche Dateifunktionalität ist in den Klassen `java.io.FileInputStream` und `java.io.FileOutputStream` verborgen. Lohnt es sich, Mock-Klassen für diese beiden »Ströme« zu erzeugen, indem man die diversen `read`- und `write`-Methoden überschreibt?

In den meisten Fällen lohnt es sich nicht, da uns die Funktionalität von Streams, die ihre Daten in Buffer schreiben und aus Buffern holen, schon in Form diverser anderer Stream-Klassen (z.B. `ByteArrayInputStream` und `ByteArrayOutputStream`) zur Verfügung steht. Diese können dann für Testzwecke den Platz der File-Streams einnehmen.

Schauen wir uns das am Beispiel unseres `FileLogger` von oben an (Seite 97 ff.). Zunächst machen wir aus dem `FileLogger` einen `StreamLogger`, der zwei Konstruktoren anbietet:

```
import java.io.*;
public class StreamLogger implements Logger {
    private PrintWriter writer;
    public StreamLogger(OutputStream out)
        throws IOException {
        writer = new PrintWriter(out);
    }
    public StreamLogger(String filename) throws IOException {
        this(new FileOutputStream(filename));
    }
}
```

```

    }
    public void close() {
        writer.close();
    }
    public void logLine(String logMessage) {
        writer.println(logMessage);
    }
}

```

Nun können wir im Konstruktor einen beliebigen `InputStream` übergeben. Dies ermöglicht einen vom Dateizugriff unabhängigen Test:

```

public void testLogLine() throws IOException {
    ByteArrayOutputStream out =
        new ByteArrayOutputStream();
    StreamLogger logger = new StreamLogger(out);
    logger.logLine("Zeile 1");
    logger.logLine("Zeile 2");
    logger.close();
    ByteArrayInputStream in =
        new ByteArrayInputStream(out.toByteArray());
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(in));
    assertEquals("Zeile 1", reader.readLine());
    assertEquals("Zeile 2", reader.readLine());
    assertNull("Dateiende erreicht", reader.readLine());
    reader.close();
}

```

Das ganze Herumhantieren mit verschiedenen `Stream`- und `Writer`-Klassen lässt sich jetzt noch schön in eine wiederverwendbare `Mock`-Klasse stecken:

```

import java.io.*;
import java.util.*;
public class MockTextOutputStream extends OutputStream {
    private ByteArrayOutputStream outputStream;
    private List expectedLines = new ArrayList();
    private boolean streamClosed = false;
    public MockTextOutputStream() {
        outputStream = new ByteArrayOutputStream();
    }
    public void addExpectedLine(String line) {
        expectedLines.add(line);
    }
    public void close() throws IOException {
        streamClosed = true;
    }
}

```

```

        outputStream.close();
    }
    public void flush() throws IOException {
        outputStream.flush();
    }
    private InputStreamReader getReader() {
        InputStream input =
            new ByteArrayInputStream(outputStream.toByteArray());
        return new InputStreamReader(input);
    }
    public void verify() throws IOException {
        if (!streamClosed) {
            Assert.fail("Stream was not closed");
        }
        BufferedReader reader =
            new BufferedReader(this.getReader());
        Iterator i = expectedLines.iterator();
        while(i.hasNext()) {
            String expectedLine = (String) i.next();
            String actualLine = reader.readLine();
            Assert.assertEquals(expectedLine, actualLine);
        }
        Assert.assertNull("EOF expected", reader.readLine());
    }
    public void write(byte[] b) throws IOException {
        outputStream.write(b);
    }
    public void write(int b) throws IOException {
        outputStream.write(b);
    }
}

```

Dieser `MockTextOutputStream` kann immer dann verwendet werden, wenn es darum geht, zeilenweise Ausgaben in einen beliebigen `OutputStream` zu überprüfen. Häufig wird eine Anpassung dieser Mock-Klasse an die lokalen Bedürfnisse nötig sein. Unser geänderter Test sieht durch die Verwendung von `MockTextOutputStream` nun so aus:

```

public void testLogLine() throws IOException {
    MockTextOutputStream mockStream =
        new MockTextOutputStream();
    mockStream.addExpectedLine("Zeile 1");
    mockStream.addExpectedLine("Zeile 2");
    StreamLogger logger = new StreamLogger(mockStream);
    logger.logLine("Zeile 1");
    logger.logLine("Zeile 2");
}

```

```

        logger.close();
        mockStream.verify();
    }

```

Der eigentliche Testcode ist kürzer geworden. Je mehr Tests dieser Art man besitzt, desto lohnender ist die Investition in den Mock-Stream.

Im vorliegenden Beispiel fällt auf, dass die `testLogLine()`-Methode stark an die letzte Fassung der `testSimpleLogging()`-Methode in `LogServerTest` erinnert (vgl. Kapitel 6.5). Dies liegt daran, dass der `StreamLogger` nichts weiter tut, als die hereinkommenden Log-Zeilen in einen `PrintWriter` zu schieben. Ob diese bloße Delegation überhaupt eines eigenen Tests bedarf, kann durchaus unterschiedlich beurteilt werden, wir plädieren jedoch für die Erstellung des Tests. Denn selbst wenn wir im Augenblick sehen, dass der Code unserer `logLine()`-Methode genau das tut, was er tun soll, kann das nach dem nächsten Refactoring schon völlig anders sein.

Angelehnt an `MockTextOutputStream` empfehlen wir an dieser Stelle die Implementierung einer `MockTextInputStream`-Klasse zur Übung. Oder besser doch eine Kaffeepause ...

Testen einer einfachen
Delegation

6.11 Noch mehr typische Mock-Objekte

Weitere
Anwendungsgebiete

Die Verwendung von Mock-Objekten bietet sich an zahlreichen Stellen unseres Codes an. Die Mock-Streams des vorangegangenen Unterkapitels sind dabei ebenso typisch wie die Beispiele, die wir u.a. in den Kapiteln 9 und 12 noch kennen lernen werden. Hier noch einige Ideen:

- Das korrekte Versenden von *Events* an entsprechende *Listener-Objekte* lässt sich durch *MockListener* überprüfen. Je nach Komplexität der Event-Instanzen könnte man dabei die erwartete Sequenz an Events durch String-Objekte beschreiben, die den `toString()`-Repräsentationen der Events entsprechen.
- Möchte man die genaue *Reihenfolge von empfangenen Nachrichten* verifizieren, dann bietet sich auch hier die Umwandlung der Nachrichten im Mock-Objekt zu Strings an. Unter Umständen erfordert dieses Vorgehen jedoch häufige Anpassungen im Zuge von Methodenumbenennungen.
- Die Überprüfung einer Nachrichten- oder Event-Sequenz über mehrere Clients (und damit Mock-Objekte) hinweg, lässt sich bewerkstelligen, indem man allen betroffenen Mock-Objekten einen *Nachrichtenregistrator* mitgibt. Dieser Registrator spielt die Rolle des eigentlichen Mocks und vergleicht die erwartete mit der tatsächlichen Nachrichtenfolge.

Diese Art von Tests reagiert äußerst empfindlich auf kleine Änderungen der Implementierung unserer CUT und kann daher nur für Fälle empfohlen werden, in denen die exakte Reihenfolge Teil der Spezifikation ist. Dies trifft häufig auf Frameworks zu, in denen abstrakte Framework-Oberklassen die Aufrufreihenfolge abstrakter Methoden garantieren⁴.

6.12 Fremde Komponenten

Das Testen mit Dummy-Objekten funktioniert immer dann wunderbar, wenn unsere Hilfs- bzw. Serverobjekte die einfache Erstellung von Attrappen erlauben. Arbeiten wir ausschließlich mit eigenem Code, dann können wir diese Art der Testbarkeit schlimmstenfalls durch größere Refactoring-Maßnahmen herstellen. Haben wir à la *Test-First* gearbeitet, so ergibt sich die Testbarkeit meist von alleine.

Anders sieht dies bei unseren Schnittstellen zur Java-Bibliothek oder eingekauften Komponenten aus. Mit großem Glück sind auch hier die externen APIs mittels Interfaces oder abstrakter Klassen gekapselt und können in unseren Tests leicht durch Mock-Objekte ersetzt werden. Ein Beispiel dafür ist die Klasse `java.io.OutputStream`, die wir oben ohne größere Schwierigkeiten durch unseren `MockTextOutputStream` ersetzen konnten. Gerade jedoch bei Bibliotheken von Drittanbietern, und wenn wir mit den Teilen der Java-Bibliothek zu tun haben, die noch aus JDK 1.0 übrig geblieben sind, sieht die Situation oft so aus⁵:

Testbarkeit fremder Klassen

```
import thirdparty.*;
public class MyClient {
    public void doSomething(String arg) {
        TheirRequest request = new TheirRequest(arg);
        TheirResponse response = request.send();
        String answer = response.getAnswer();
        // do something with answer...
    }
}
```

Dabei sind die `Their*`-Klassen die vom Drittanbieter zur Verfügung gestellten Schnittstellen; `MyClient` ist unsere eigene Klasse. Gemäß dem bislang Erlernen planen wir nun folgendes Vorgehen: Wir bauen uns eine Klasse `MockRequest`, die von `TheirRequest` abgeleitet wird, und

4. Dies entspricht dem *Template-Method*-Entwurfsmuster aus [Gamma95].

5. Das Beispiel ist sinngemäß der Diskussion in [URL:WikiUTATP] entnommen.

diese gibt uns dann bei `send()` eine Instanz von `MockResponse`, ihrerseits abgeleitet von `TheirResponse`, zurück, deren Reaktion auf `getAnswer()` wir natürlich wieder vorher festgelegt haben. Gelingt uns das, so haben wir auch diese Fremdschnittstelle mit unserem Attrappenangriff zähmen können. Doch häufig scheitern wir bei diesem Versuch aus einem oder mehreren der folgenden Gründe:

- `TheirRequest` und/oder `TheirResponse` sind *final*, d.h., von ihnen können keine Unterklassen abgeleitet werden.
- Die Klassen selbst sind nicht *final*, dafür aber die Methoden `send()` und/oder `getAnswer()`.
- Die vorhandenen Konstruktoren von `TheirResponse` sind für eine Mock-Unterklasse nicht zu gebrauchen, da sie Parameterobjekte benötigen, deren Erzeugung von außerhalb der Bibliothek nicht möglich ist bzw. wieder neue Parameter erfordert usw.
- Die Bibliotheksklassen tun Dinge, die wir mit unserem Mock-Ansatz gerade vermeiden wollten, z.B. Netzwerkzugriffe.

Es hilft nicht weiter, über die Unzulänglichkeiten der unbekanntem Entwickler zu jammern. Nein, wir müssen unser Testproblem lösen. Bei vorliegendem Quellcode könnten wir die entsprechenden Klassen so ändern, dass sie sich unseren Testanstrengungen nicht mehr entgegenstellen. Damit hängen wir uns jedoch für zukünftige Versionen der Fremdbibliothek einen Wartungsklotz ans Bein.

Einführung eines Adapters

Gibt es vielleicht einen anderen Weg? Bauen wir doch einfach noch eine Schicht um die Fremdschnittstelle herum: Zunächst definieren wir ein Interface, das die Funktionalität der Fremdbibliothek für unsere speziellen Bedürfnisse definiert. Dieses Vorgehen ist in [Gamma95] auch als *Adapter*-Muster beschrieben. Im vorliegenden Fall sieht das so aus:

```
public interface AnswerFactory {
    String getAnswer(String arg);
}
```

Unser eigener Client benutzt von nun an nur noch diese »Fabrik« zum Erzeugen des `answer`-Objekts:

```
import thirdparty.*;
public class MyClient {
    private AnswerFactory factory;
    public MyClient(AnswerFactory factory) {
        this.factory = factory;
    }
    public void doSomething(String arg) {
```

```

        String answer = factory.getAnswer(arg);
        // do something with answer...
    }
}

```

Jetzt fehlen nur doch die beiden Implementierungen von AnswerFactory ...

```

public class MockAnswerFactory implements AnswerFactory {
    private String answer;
    public MockAnswerFactory(String presetAnswer) {
        answer = presetAnswer;
    }
    public String getAnswer(String arg) {
        return answer;
    }
}

import thirdparty.*;
public class AnswerFactoryAdaptor implements AnswerFactory {
    public String getAnswer(String arg) {
        TheirRequest request = new TheirRequest(arg);
        TheirResponse response = request.send();
        return response.getAnswer();
    }
}

```

Und schon sind wir da, wo wir hin wollten: Wir besitzen ein Mock-Objekt, mit dessen Hilfe wir unsere MyClient-Klasse testen können. Eine kleine Lücke hat das Ganze jedoch bekommen: Die Klasse AnswerFactoryAdaptor bleibt ungetestet. Könnten wir sie testen, hätten wir uns die Mühe von Anfang an nicht machen müssen. Aus diesem Grund sollten die Methoden dieser »Weiterleitungsklasse« so einfach wie möglich bleiben. Benötigen wir außer der reinen Übersetzung von Methodenaufrufen zusätzliche Logik in dieser Klasse, empfiehlt sich eine weitere Aufteilung in *Adapter* und *Delegator*.

Außer der Testbarkeit haben wir übrigens noch etwas Weiteres gewonnen: die Unabhängigkeit unseres Clients von der externen Schnittstelle. Entscheiden wir uns später für die Verwendung einer anderen Bibliothek, müssen wir »nur noch« die Adaptor-Klasse austauschen.

Es gehört normalerweise nicht zu unseren Aufgaben, die Funktionalität der externen Bibliothek zu testen; dies sollte bereits andernorts geschehen sein. Sinnvoll ist jedoch, eine Hand voll Tests aufzunehmen, die unsere spezielle Verwendung der Bibliothek abdecken, um sicher-

*Gewonnene
Unabhängigkeit*

*Tests für externe
Bibliotheken*

zugehen, dass wir die Schnittstelle richtig verstanden haben und dass auch eine neue Version der Bibliothek noch wie erwartet funktioniert. Aber das ist eine andere Geschichte.

6.13 Pro und Contra

Die Verwendung von Dummy- und Mock-Objekten ist sowohl in der Gemeinde der Softwaretester als auch in der XP-Welt nicht unumstritten. Tragen wir die Argumente beider Seiten nochmal zusammen. Zunächst die Vorteile von Dummy-Objekten:

Vorteile von
Dummy-Objekten

- Wir können mit einer feineren Granularität und größeren Genauigkeit testen. Dies zeigt sich u.a. darin, dass wir eine *Test-Failure* immer auf einen Fehler im Testobjekt oder den Test selbst zurückführen können und nicht in Schichten des Programms wühlen müssen, die augenblicklich nicht von Interesse sind.
- Die Attrappe erlaubt es, uns auf das zu testende Objekt zu konzentrieren und den für den Test nötigen Anfangszustand leichter zu erzeugen. Der Aufbau eines komplexen Anfangszustandes mit den richtigen (eventuell persistenten) Objekten kann dagegen ein schwieriges Problem darstellen.
- Testeigene Dummy-Objekte stellen die Wiederholbarkeit unserer Tests sicher. Echte Serverobjekte ändern durch einen Test möglicherweise ihren Zustand und müssten nach dem Test wieder zurückgesetzt werden. Dies ist im besten Fall zusätzlicher Aufwand, unter Umständen sogar völlig unmöglich.
- Die Verwendung echter Serverobjekte im Test stellt eine Art von *Mikrointegrationstest* dar. *Integrationstests* überlappen jedoch stark mit den *Funktionstests* und sind nur in besonderen Fällen Teil der *Entwicklertests*. Die Erfahrung zeigt auch, dass Tests, die Objekte vieler Schichten integrieren, auf Dauer zu langsam für ein schnelles und ständiges *Feedback* werden.
- Dummy-Objekte ermöglichen uns, das Verhalten des Testobjekts an den Rändern erlaubter Wertebereiche zu testen sowie Fehlerfälle und Exceptions sehr dediziert zu simulieren. Die Erzeugung bestimmter Randbedingungen und Fehlerfälle durch mehrere Abstraktions- und Zugriffsschichten hindurch ist äußerst schwierig und in vielen Fällen sogar unmöglich.
- Dummy-Objekte erlauben darüber hinaus ein Top-down-Vorgehen bei der Softwareentwicklung, wo wir sonst, wegen der Abhängig-

keiten der Objekte untereinander und von Schicht zu Schicht, nur bottom-up entwickeln könnten. Dies bedeutet auch, dass wir bei Verwendung von Mock-Objekten nicht mehr die komplette Infrastruktur unseres Systems zu Beginn festlegen müssen, sondern diese iterativ und inkrementell auf- und ausbauen können.

- Testen mit Dummy-Objekten verbessert die Struktur des resultierenden Programms, da es kleine Objekte bevorzugt und für die Einhaltung des *Dependency Inversion Principle* und des *Law of Demeter* (siehe Anhang C: *Glossar*, Seite 310) sorgt.

Zusätzliche Vorteile bieten Mock-Objekte als besondere Attrappen-Spezies:

- Herkömmliche Tests sind auf das Überprüfen von Rückgabewerten und nach außen sichtbaren Zustandsänderungen des Testobjekts angewiesen. Mock-Objekte erlauben es uns, zu überprüfen, ob der Zugriff des Testobjekts auf seine Helfer- und Serverobjekte richtig ist. Wir testen sozusagen von innen.
- Mock-Objekte dienen als Behälter, in dem wir sich wiederholende Testfunktionalität sammeln können. Sie erleichtern das *Refactoring* von Testcode und stellen ein Muster dar, das die Kommunikation des Codes verbessert.

Vorteile von Mocks

Alle genannten Punkte haben entweder mit der Erhöhung der Unabhängigkeit (von Tests und Code) oder der Verbesserung der Kommunikation zu tun. All diesen Vorteilen stehen auch Nachteile gegenüber:

Nachteile

- Dummy-Klassen können Fehler enthalten. Dieses Problem wird jedoch dadurch relativiert, dass die Wahrscheinlichkeit, dass sich Fehler in der Dummy-Klasse und in der Testklasse gegenseitig aufheben, sehr klein ist. Fehler der Dummy-Klasse werden daher meist sofort entdeckt.
- Mock-basiertes Testen findet keine Fehler, die sich aus dem Zusammenspiel mehrerer Komponenten ergeben. Diese Fehlerkategorie decken wir einfacher durch Funktionstests ab. Werden solche Fehler dennoch zu einem häufigen Problem, sollte man über zusätzliche lokale Integrationstests an den kritischen Stellen des Systems nachdenken, die idealerweise aber nur zwei aneinander grenzende Schichten integrieren.
- Änderungen am Interface der echten Implementierung erfordern Änderungen am Dummy-Objekt. Dieser zusätzliche Aufwand macht erfahrungsgemäß jedoch nur einen kleinen Teil des

Gesamtaufwands zur Aktualisierung aller Tests aus. Häufig hilft auch die IDE beim Finden der zu ändernden bzw. zu ergänzenden Signaturen.

- Das Testen mit Attrappen muss von den Entwicklern erlernt werden. Mit der Zeit wächst jedoch nicht nur die Erfahrung, sondern auch die Bibliothek an wiederverwendbaren Dummy- und Mock-Objekten.
- »Testen von innen« erfordert, dass man weiß, was in der Klasse geschieht bzw. geschehen soll. Ändert sich die Implementierung, z.B. weil man bessere Wege gefunden hat, mit einem Serverobjekt zu arbeiten, muss häufig auch der Testcode (inklusive Mock-Objekten) geändert werden, obwohl das Testobjekt nach außen ein unverändertes Verhalten zeigt. Mock-Objekte werden daher bevorzugt für das Testen relativ stabiler Implementierungen verwendet.

*Kosten der Dummy-
Programmierung*

Robert Binder beurteilt den Aufwand zur Erstellung von *Stubs* als sehr hoch (siehe [Binder99] S. 662 ff.). Vor allem die große Anzahl von Stub-Objekten, die gebraucht werden, um jeden einzelnen Test mit den nötigen Antworten zu versorgen, sieht er als großes Hindernis bei der generellen Nutzung dieser Technik. Unsere Erfahrung stützt diese These nicht; wir kommen meist mit einem einzigen und leicht zu konfigurierenden Mock-Objekt pro Test aus. Diese Diskrepanz in der Erfahrung ergibt sich teils aus den Unterschieden zwischen der Anwendung von *Test-First*- und *Test-After*-Entwicklung, von dem die klassische Testtheorie ausgeht. Zum anderen simulieren herkömmliche Stubs oft das echte Verhalten des Systems, was einen deutlich höheren Implementierungsaufwand erfordert als schlanke Mock-Objekte.

Brian Marick sieht zwei Hauptprobleme bei der Verwendung von »Stubs« (siehe [Marick00] S. 110): (a) Jede falsche Vorstellung (*Misconception*), die wir über das echte Objekt haben, implementieren wir auch im Dummy-Objekt. (b) Fehler, die wir sonst über indirekte Aufrufe im Hilfsobjekt gefunden hätten, bleiben unentdeckt. Grund (b) warnt uns vor der Annahme, dass durch die Mock-Technik Interaktionstests, wie sie in Kapitel 4.6 beschrieben werden, völlig wegfallen können; bestenfalls reduziert sich deren Anzahl.

Heuristiken für den Einsatz von Mocks

Die richtige Abwägung zwischen den zahlreichen Vor- und Nachteilen erfordert Erfahrung und Mut zum Experimentieren. Die Autoren verwenden Dummy- und Mock-Objekte in folgenden Situationen:

- Wir kommen ohne sie nicht aus, wenn die Tests zu langsam laufen, wenn die »richtige« Klasse noch nicht existiert oder wenn bestimmte Grenz- und Fehlerfälle nicht anders testbar sind.
- Sie verbessern die Lesbarkeit und Wartbarkeit unseres Testcodes, z.B. durch Entfernung von Codeduplikation.

Wenn wir den Test genauso leicht und genauso lesbar und mit der gleichen oder weniger Redundanz ohne Dummy-Objekte hinschreiben können, dann verzichten wir auf sie. Je mehr wir uns jedoch an sie gewöhnt haben, desto häufiger finden wir gute Gründe für ihre Verwendung.

Wir müssen jedoch darauf achten, dass unsere Mock-Objekte nicht zu komplex werden. Anzeichen für zu große Komplexität sind:

- Sie duplizieren Programmlogik aus den »richtigen« Klassen.
- Sie rufen ihrerseits andere Mock- oder Dummy-Objekte auf.
- Wir haben das Bedürfnis, Testfälle für die Mock-Objekte selbst zu schreiben.

In diesen Fällen hilft es, einen Schritt zurückzutreten und uns zu fragen, ob wir die Mocks nicht vereinfachen können, z.B. durch die Aufteilung in mehrere Mock-Klassen, ob wir sie vielleicht gar nicht benötigen oder ob unser Mock-Problem nicht eigentlich unsere Aufmerksamkeit auf ein Designproblem lenken möchte.

6.14 Zusammenfassung

Ein *Dummy-Objekt* ist ein Objekt, das ein anderes für die Dauer eines Tests ersetzt. Dabei implementiert es das gleiche Interface wie die »richtigen« Objekte, ersetzt dabei jedoch komplexe Berechnungen durch konstante Rückgaben, wirft auf Befehl Exceptions, führt zusätzliche Parameterüberprüfungen durch oder tut andere Dinge, die man nur in den Tests benötigt. *Mock-Objekte* sind besondere Dummy-Objekte, die zusätzlich die Spezifikation des erwarteten Verhaltens und die Überprüfung des tatsächlichen Verhaltens an sich ziehen.

Hauptargument für die Verwendung von Attrappen ist die gewonnene Unabhängigkeit in den Tests und die damit einhergehenden Designverbesserungen. Es gibt jedoch noch zahlreiche weitere Vor- und Nachteile, daher sollten auch Mock-Objekte nicht reflexartig, sondern nur nach Abwägung der positiven und negativen Auswirkungen verwendet werden. Typische Anwendungsfälle sind die Zugriffe auf Dateien oder andere externe Ressourcen sowie die Anbindung von Fremdkomponenten. Auch in Kapitel 9 (»Persistente Objekte«) und

Kapitel 12 (»Web-Applikationen«) werden Dummy-Objekte und Mocks eine wichtige Rolle spielen.