

Selbstvalidierende Mock-Objects

Verbesserung der Testbarkeit im EAI-Umfeld

Hasko Heinecke
Credit Suisse
Zürich, Schweiz
Hasko@Heinecke.com

Christian Noack
Agile Methoden
Dortmund, Deutschland
Christian.Noack@agile-methoden.de

Net.Object Days 2004

Zusammenfassung

Ein generelles Problem bei der Softwareentwicklung ist das Testen von Anwendungen mit Schnittstellen zu externen Systemen. Im Rahmen von Unit Tests werden Schnittstellen in der Regel mit sogenannten Mock Objects getestet, die die externen Systeme simulieren. Das funktioniert solange gut, wie sich die externen Systeme nicht verändern. In einem Umfeld mit vielen externen Systemen (beispielsweise einem EAI-Umfeld) ergibt sich jedoch im Laufe der Zeit eine hohe Wahrscheinlichkeit von Veränderungen an den externen Systemen. Die verwendeten Mock Objects repräsentieren u.U. dann nicht mehr das externe System und Tests laufen ggf. fälschlicherweise durch. Der vorliegende Artikel schlägt vor, wie man durch die Einführung von selbstvalidierenden Mock Objects mit diesem Auseinanderlaufen umgehen kann.

sind meist historisch gewachsen oder von Drittanbietern zugekauft. Heute wächst der Wunsch nach übergreifender Integration der Geschäftsprozesse, ohne die bestehenden Software-Systeme ersetzen zu müssen. Es existiert also ein großes Bedürfnis, verschiedenste Systeme zu integrieren. Dabei beschränkt sich die Integration nicht bloß auf den parallelen Betrieb verschiedener Systeme in einer Infrastruktur.

Häufig spiegelt sich ein Geschäftsprozess in gleichzeitigen oder voneinander abhängigen Aktionen in unterschiedlichen Systemen wieder. Das Fehlerrisiko wächst schon allein wegen der Quantität der Systeme erheblich an. Bedroht wird die Stabilität und Sicherheit der Anwendungen darüber hinaus durch die Komplexität, die der Integration von vielen Anwendungen innewohnt. Neben dem Risiko steigt in den letzten Jahren aber auch die unbedingte Wichtigkeit, die solche Anwendungen für das Überleben von Unternehmen haben.

1 Enterprise Architecture Integration

Unternehmen setzen Software-Lösungen ein, um kritische Geschäftsabläufe zu automatisieren oder die Sachbearbeiter zu unterstützen. Die Einsatzgebiete sind vielfältig; dazu gehören zum Beispiel das Buchungssystem, logistische Aufgaben und Kunden- sowie Auftragsverwaltung. Die Systeme

2 Entwicklung im EAI-Umfeld

Die EAI (enterprise application integration) genannte Integration von verschiedensten Anwendungen innerhalb eines Unternehmens stellt Softwareentwickler vor das Problem sicherzustellen, dass die Vielfalt an unternehmensweiten Anwendungen, die ggf. in unterschiedlichen Programmen

miersprachen und auf unterschiedlichen Plattformen implementiert sind, effektiv alle relevanten Informationen austauschen. Die verschiedenen Systeme (Host, DBs, Java-Applikationen) sind über verschiedene Protokolle (CORBA, Web Service usw.) miteinander verknüpft. Es gibt eine Vielzahl von EAI-Produkten (z.B. MQSeries und andere Middleware), die diese Integration vereinfachen sollen. Die so integrierten Systeme sind ziemlich anfällig, vor allem, wenn die einzelnen Komponenten von unterschiedlichen Personen und Organisationen betreut und weiterentwickelt werden. Unterschiedliche Lifecycle führen dazu, dass plötzlich Dinge nicht oder anders funktionieren. Zum Teil ist der Informationsaustausch zwischen den Verantwortlichen der integrierten Komponenten schlecht oder nicht vorhanden. Das geht so weit, dass die Abhängigkeiten zwischen den Komponenten nicht bekannt sind.

Ein möglicher Ansatz ist eine strikte Hierarchie: Eines ist das „Master“-System, das bestimmt, wann gemacht werden muss. Aber spätestens wenn die Systeme von Drittanbietern außerhalb des eigenen Unternehmens zur Verfügung gestellt werden, ist so ein einfaches Beziehungsnetz nicht mehr möglich: Einerseits benutzt der Kunde mehrere verschiedene, eventuell sogar konkurrierende Systeme, andererseits hat der Anbieter mehr als einen Kunden. Ein solcher Zwang zum Gleichschritt skaliert also nicht.

Besser ist es, die zu integrierenden Komponenten zu identifizieren und klare Schnittstellen zwischen diesen festzulegen. Bei der Entwicklung von neuer Software in einem so strukturierten System wird es möglich, automatische Tests zu installieren, wenn es gelingt im Testumfeld die externen Systeme durch sogenannte Mock Objects zu ersetzen. Diese simulieren reale Komponenten oder ganze Systeme. Sie sind dabei mehr als Stubs, die nur Schnittstellen repräsentieren. Mocks ermöglichen, gegen sie zu programmieren, indem sie das Verhalten des externen Systems simulieren soweit es in diesem Kontext erforderlich ist. Es findet eine Simulation der Interaktion statt, die vom aufrufen-

den System (i.d.R. im Rahmen von UnitTests) nicht als Simulation wahrgenommen wird.

Entscheidend ist dabei Folgendes: Es reicht nicht, das Funktionieren des eigenen Systems gegen die Mocks des zugrunde liegenden zu testen. Änderungen am zugrunde liegenden System würde man so nicht bemerken. Von solchen Änderungen ist in integrierten (EAI-) Systemen grundsätzlich auszugehen. Man muss also zusätzlich testen, ob die Mock Objects noch gültige Annahmen über simulierte Systeme treffen. Zur Lösung dieses Problems schlagen wir die Einführung von selbstvalidierenden Mock Objects vor.

3 Beispielsystem

Die Funktionsweise von selbstvalidierenden Mock Objects soll an der beispielhaften Entwicklung einer Anwendung erklärt werden. Als Beispiel wollen wir ein so genanntes Management-Informationssystem (MIS) in Java entwickeln. Diese Anwendung liefert zu einem Manager den Gesamtumsatz aller Mitarbeiter seiner Abteilung. Dazu werden Informationen von einem vorhandenen Sales-System (Schnittstelle siehe Abbildung 1) und vom ebenfalls bereits existierenden Personalsystem (Schnittstelle siehe Abbildung 2) benötigt.

Um uns darüber klar zu werden, was das MIS leisten soll, schreiben wir zuerst einen Unit Test (siehe Abbildung 3). Nach Implementierung der entsprechenden Klasse und Methode führen wir den Test aus; er funktioniert erwartungsgemäß.

In den kommenden Wochen wird das MIS weiter entwickelt. Allerdings treten Probleme beim Testen auf: Wegen einer Reorganisation benötigt die für das Personalsystem zuständige Abteilung die entsprechende Testumgebung selbst und kann sie nur sporadisch zur Verfügung stellen. Unsere Tests schlagen dadurch schon beim `setUp` oft fehl.

```

public interface SalesSystem {

    /**
     * Returns the revenue for the given employee ID for the given date.
     * Only month and year are taken from the date, day of month and time
     * are ignored. The method always returns monthly revenues.
     *
     * @param employeeId
     *         the unique employee id
     * @param date
     *         the month and year to evaluate
     * @return the revenue generated in the given month by the employee
     *         with the given id
     */
    double getRevenue(int employeeId, Date date);
}

```

Abbildung 1: Schnittstelle des Sales-Systems

```

public interface HrSystem {

    /**
     * For a given manager id returns the ids of this manager's team.
     * @param employeeId the manager's id
     * @return the array of team ids
     *         (or an empty array, never <code>null</code>)
     */
    int[] getTeam(int employeeId);
}

```

Abbildung 2: Schnittstelle des Personalsystems

```

public class ManInfoSystemTest extends TestCase {

    private SalesSystem s;
    private HrSystem hr;
    private ManInfoSystem mi;

    public ManInfoSystemTest(String name) {
        super(name);
    }

    protected void setUp() {
        s = SalesSystemHelper.getInstance();
        hr = HrSystemHelper.getInstance();
        mi = new ManInfoSystemImpl(hr, s);
    }

    public void testGetAggregatedRevenue() {
        Date date = new Date();
        double expected = 0;
        for (int i = 0; i < 100; i++) {
            expected += s.getRevenue(i, date);
        }
        assertEquals(expected, mi.getAggregatedRevenue(0, date), 0.01);
    }
}

```

Abbildung 3: Unit Test mit direktem Zugriff

Wir reagieren schnell auf die veränderte Situation und ersetzen den Zugriff auf das tatsächliche Personalsystem durch ein Mock Object. Wir verwenden dazu *EasyMock*. Da wir bei der Implementierung des MIS das Pattern *DependencyInjection* verwendet haben, können wir nun durch kleine Änderungen das Mock Object für das Personalsystem einfügen. Weil wir gerade dabei sind, ersetzen wir auch noch schnell das Sales-System durch ein Mock Object (siehe Abbildung 4).

Nun sind wir in der Lage, zu entwickeln und zu testen, ohne auf die integrierten Systeme zuzugreifen. Unsere Komponententests (Unit Tests) laufen auch ohne Rückgriff auf die externen Systeme. Das ist besonders nützlich, weil auch keinerlei Middleware benötigt wird. Im Normalfall können externe Systeme nämlich nicht so einfach wie im Beispiel dargestellt aufgerufen werden. Statt dessen sind sie

hinter einer Kommunikationsschicht wie *WebServices*, *EJB*, *CORBA* oder hinter proprietären Protokollen gewissermassen verborgen.

All diese Komponenten können potentiell nicht verfügbar oder fehlerhaft sein. Erschwerend kommt hinzu, dass in den wenigsten Fällen vollständige Testumgebungen für das integrierte System bereitgestellt werden. Statt dessen laufen die Tests unseres MIS gegen die Testumgebung des Personalsystems, also der Umgebung, in der dessen Entwickler ebenfalls testen. Streng genommen müsste es ja drei Umgebungen geben: Die produktive Umgebung, die Testumgebung für die Entwickler des Personalsystems sowie die Testumgebung für die Entwickler dritter Systeme wie uns. Berücksichtigt man nun noch, dass unterschiedliche Versionen eines produktiven Systems parallel getestet werden müssen und außerdem die Dritt-

```

public class ManInfoSystemTest extends TestCase {

    private MockControl hrCtrl;
    private HrSystem hr;
    private MockControl sCtrl;
    private SalesSystem s;
    private ManInfoSystem mi;

    public ManInfoSystemTest(String name) {
        super(name);
    }

    protected void setUp() {
        hrCtrl = MockControl.createControl(HrSystem.class);
        hr = (HrSystem) hrCtrl.getMock();
        sCtrl = MockControl.createControl(SalesSystem.class);
        s = (SalesSystem) sCtrl.getMock();
        mi = new ManInfoSystemImpl(hr, s);
    }

    public void testGetAggregatedRevenue() {
        Date date = new Date();

        /* Set up HR mock */
        int[] team0 = { 1 };
        int[] team1 = { 2, 3 };
        hrCtrl.expectAndReturn(hr.getTeam(0), team0);
        hrCtrl.expectAndReturn(hr.getTeam(1), team1);
        hrCtrl.expectAndReturn(hr.getTeam(2), new int[0]);
        hrCtrl.expectAndReturn(hr.getTeam(3), new int[0]);

        /* Set up Sales mock */
        sCtrl.setDefaultMatcher(MockControl.ALWAYS_MATCHER);
        sCtrl.expectAndReturn(s.getRevenue(0, date), 3.0d, 4);

        /* Set mocks to replay mode */
        hrCtrl.replay();
        sCtrl.replay();

        /* Run actual test */
        assertEquals(12.0d, mi.getAggregatedRevenue(0, date), 0.01);

        /* Verify mock calls, optional */
        hrCtrl.verify();
        sCtrl.verify();
    }
}

```

Abbildung 4: Unit Test mit Mock Obejcts

systeme zum Teil gegen verschiedene (zukünftige) Versionen entwickelt werden, dann explodiert die Zahl der notwendigen Testumgebungen. Aus diesem Grund verzichten die meisten Unternehmen auf diesen Kostenfaktor und gehen den Kompromiss ein, nur eine (oder wenige) Testumgebungen zur Verfügung zu stellen. Dabei kommen sich die Entwicklungsprojekte regelmäßig in die Quere, was aber immer noch billiger ist als die saubere Lösung. Dieses Problem ist durch unsere Mock Objects zumindest für die Komponententests weitgehend gelöst.

Parallel zur Entwicklung des MIS führen wir – wenn möglich – regelmäßig Integrationstests durch. Leider finden sie aus den genannten Gründen nicht täglich, sondern zum Beispiel alle 14 Tage statt. Ein solcher Integrationstest könnte irgendwann trotz funktionierender Komponententests fehlschlagen, weil beispielsweise die zurück gelieferten Umsätze zu hoch sind. Offensichtlich hat sich eines der externen Systeme geändert und verursacht die Fehler. Unsere Mock Objects repräsentieren nicht mehr das Verhalten der von ihnen vorgetäuschten Systeme.

Doch wo ist der Fehler zu suchen? Liegt er im Personalsystem oder im Sales-System? Oder ist er doch in deren impliziten Zusammenspiel oder gar in unserem eigenen Code zu suchen? Hier bieten uns die Mock Objects von *EasyMock* leider keine Unterstützung. Im Folgenden ist beschrieben, wie eine solche Unterstützung aussehen könnte.

Die Schlüsselidee ist, dass eigentlich alle Informationen zur Validierung der Mock Objects vorhanden sind: Wir geben im Unit Test an, welche Methoden aufgerufen werden und welche Resultate wir erwarten. Im Prinzip müssen wir die Mock Objects nur mit einem Zugang zu den vorgetäuschten, echten Systemen versehen, so dass sie die entsprechenden Methoden dort aufrufen könnten. Mit Hilfe eines Schalters könnte man so aus den Mock-Aufrufen echte Aufrufe machen und deren Resultate mit den erwarteten Resultaten vergleichen. Weichen diese voneinander ab, so kann eine entsprechende Fehlermeldung zurück gegeben

werden.

Eine einfache Implementation dieses Verhaltens kann man in *EasyMock* mit Modifikationen in zwei Klassen erreichen. Die erste betrifft die Klasse `MockControl` (siehe Abbildung 5). Die Modifikation ermöglicht im Unit Test wahlweise ein reales Objekt hinter dem Mock mitzugeben (siehe Abbildung 6). Die Klasse `ReplayState` (siehe Abbildung 7) wird um den in Abbildung 8 gezeigten Code erweitert. Zentral sind dabei die Änderungen in der Methode `invoke()`¹. Dort wird, wenn ein `realObject` definiert wurde, bei diesem dieselbe Methode wie beim Mock Object aufgerufen und die beiden Resultate verglichen. Stimmen sie nicht überein, wird ein Fehler gemeldet.

Durch Aufrufen der Test Cases mit den tatsächlichen Objekten erhalten wir die Fehlermeldung, dass sich die Struktur der Teams geändert hat. Wir sind bisher von der impliziten Annahme ausgegangen, dass eine Person immer nur in einem Team sein kann. Die Änderungen im Personalsystem hatten aber unter anderem zum Ziel, dass auch eine anteilige Beteiligung in mehreren Teams möglich ist. So arbeitet zum Beispiel Mitarbeiter 2 nun zu 50% für Mitarbeiter 0 und zu 50% für Mitarbeiter 1. Die Umsätze sind entsprechend diesem Faktor anteilig zuzurechnen. Unser Code aber rechnet die Umsätze stets zu 100% auf das Team, so dass der entsprechende Umsatz plötzlich doppelt gerechnet wird. Mit Hilfe der selbstvalidierenden Mock Objects sind wir in der Lage, den Fehler sehr schnell auf eines der beteiligten Systeme einzugrenzen, ohne an unseren Test Cases Änderungen vorzunehmen. Es ist diese Art von erleichterten Testmechanismen, die benötigt wird, wenn ernsthaft und kostengünstig EAI-Applikationen entwickelt werden sollen.

¹Die TODO-Zeilen markieren die beiden wesentlichsten ausstehenden Erweiterungen dieses Modifikation: Das Werfen von Exceptions muss behandelt werden, um auch Unit Tests abzudecken, die die korrekte Fehlerbehandlung testen. Ausserdem sollten neben dem Vergleich der beiden Resultate mit `equals()` auch konfigurierbare Vergleicher angegeben werden können.

```

public void replay() {
    try {
        state.replay();
        state = new ReplayState(behavior);
    } catch (RuntimeException e) {
        throw (RuntimeException) e.fillInStackTrace();
    }
}

```

Abbildung 5: Original-Implementierung der Klasse MockControl, EasyMock, Version 1.1

```

public void replay(Object realObject) {
    try {
        state.replay();
        state = new ReplayState(behavior, realObject);
    } catch (RuntimeException e) {
        throw (RuntimeException) e.fillInStackTrace();
    }
}

```

Abbildung 6: Modifikation an der Klasse MockControl

```

public class ReplayState implements IMockControlState {
    ...

    public ReplayState(IBehavior behavior) {
        this.behavior = behavior;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        return behavior.addActual(new MethodCall(method, args)).returnObjectOrThrowException();
    }
    ...
}

```

Abbildung 7: Original-Implementierung der Klasse ReplayState, EasyMock, Version 1.1

```

public class ReplayState implements IMockControlState {
    ...
    private Object realObject = null;
    ...
    public ReplayState(IBehavior behavior, Object realObject) {
        this(behavior);
        this.realObject = realObject;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        Object mockResult = behavior.addActual(
            new MethodCall(method, args)).returnObjectOrThrowException();
        // TODO Catch exceptions
        if (realObject != null) {
            Object realResult = method.invoke(realObject, args);
            // TODO provide for custom comparison
            if (!realResult.equals(mockResult)) {
                String msg = createMockRealErrorMessage(method,
                    args, mockResult, realResult);
                throw new AssertionError(msg);
            }
        }
        return mockResult;
    }

    private String createMockRealErrorMessage(Method method, Object[] args,
        Object mockResult, Object realResult) {
        StringBuffer buf = new StringBuffer();
        buf.append("Mock result differs from real result in ");
        buf.append(method);
        buf.append(", arguments were: ");
        for (int i = 0; i < args.length; i++) {
            Object object = args[i];
            buf.append(object);
            buf.append(", ");
        }
        if (args.length > 0) {
            buf.delete(buf.length() - 2, buf.length());
        }
        buf.append("; mock returned ");
        buf.append(mockResult);
        buf.append(", real object returned ");
        buf.append(realResult);
        return buf.toString();
    }
    ...
}

```

Abbildung 8: Modifikation an der Klasse ReplayState

4 Fazit

Das vorliegende Papier schildert einen Vorschlag, wie auf Grundlage der bekannten Technologie der *Mock Objects* mit einfachen Erweiterungen grundlegende Probleme in EAI-Applikationen angegangen werden können. Zwar adressiert die vorgestellte Lösung nicht sämtliche Fragestellungen, aber den sehr wesentlichen Aspekt der *entkoppelten* Entwicklung der einzelnen Komponenten dieser Systeme vermag sie deutlich zu vereinfachen.

Abzugrenzen vom in diesem Papier betrachteten Problemfeld sind die Fragestellungen der Integrations- und Regressionstests sowie die Erstellung von Demonstrations-Prototypen. Jedoch glauben wir, dass auch für diese Situationen der gewählte Ansatz Potential bietet, etwa über die Aufzeichnung des realen Verhaltens der integrierten Systeme.

Vor allem angesichts der bei den uns bekannten Tools unterentwickelten Möglichkeiten zum Debugging von integrierten Applikationen ist die vorgestellte Methode zur Validierung der *Mock Objects* vielversprechend. Einer der Autoren (Hasko Heinecke) arbeitet zur Zeit daran, die vorgestellten Konzepte in einem umfangreichen EAI-Umfeld (ca. 80 Einzelapplikationen, ca. 2 Mio. Zeilen Java-Code) umzusetzen und an den notwendigen Stellen zu ergänzen und zu modifizieren. Die Schwierigkeiten von EAI-Systemen sind gross, grundlegend und nicht durch eine einzige Lösung zu erledigen. *There is no silver bullet*. Die vorgestellte Strategie ist ein Schritt auf einem langen Weg. Die Autoren sind für Feedback und weitergehende Ideen jederzeit dankbar.

Abschliessend möchten die Autoren noch Tammo Freese und den Entwicklern von EasyMock danken. Ohne die sehr saubere Implementierung dieses Tools wäre die Umsetzung unserer Ideen nicht so einfach vonstatten gegangen.

5 Die Autoren

Hasko Heinecke befasst sich seit 1991 mit der objekt-orientierten Software-Entwicklung und sammelte Erfahrungen in mehreren unter- wie auch überorganisierten Projekten. Er begann Ideen und Techniken *Agiler Prozesse* einzuführen seit er 1999 zusammen mit Kent Beck bei Daedalos Consulting gearbeitet hatte. Seit 2002 ist Hasko Softwarearchitekt bei der Credit Suisse.

Christian Noack entwickelt seit über zehn Jahren objekt-orientierte Software. Dabei arbeitete er die letzten Jahre intensiv als Berater und Trainer im Bereich Softwareprozesse, wobei er sich auf *Agile Prozesse* konzentriert. Christian ist selbstständiger Berater.