

Testen von Webanwendungen

Weiterführende Techniken für jUnit

Rolf Sabsch

- Testen einer Webapplikation
 - Der intuitive Weg – Stubs
 - Ressourcen Stub
 - Connection Stub
 - Testen unter Zuhilfenahme von Mockobjekten
 - Was sind MockObjekte
 - EasyMock als Vertreter von dynamischen Mockobjekten
 - Beispiele zum In-Container Testen
 - Jetty
 - Maven
- Fazit

- Als Servlet bezeichnet man im Rahmen der Java 2 Platform Enterprise Edition (J2EE) ein Java-Objekt, an das ein Webserver Anfragen seiner Clients delegiert, um die Antwort an den Client zu erzeugen. Der Inhalt dieser Antwort wird dabei erst im Moment der Anfrage generiert, und ist nicht bereits statisch (etwa in Form einer HTML-Seite) für den Webserver verfügbar. - Wikipedia

Stubs – der intuitive Weg

- Benötigte Schnittstellen werden einfach durch Stubs implementiert
1. Möglichkeit: Es werden die Ressourcen des Servers simuliert
 2. Möglichkeit: Es wird ein Stub für die gesamte Verbindung verwendet

- Initialisierung des Jetty- Servers

```
public void startserver() throws Exception
{
    server = new HttpServer();
    SocketListener listener = new SocketListener();

    listener.setPort(8080);
    server.addListener(listener);

    HttpContext context = new HttpContext();

    context.setContextPath("/");
    context.setResourceBase("./");
    context.addHandler(new ResourceHandler());
    server.addContext(context);

    server.start();
}
```

- Ressourcen die vom Server eigentlich vom Benutzer aufgerufen werden, werden von einem Handler definiert
`context1.setContextPath("/testGetContentOk");`
`context1.addHandler(new TestGetContentOkHandler());`

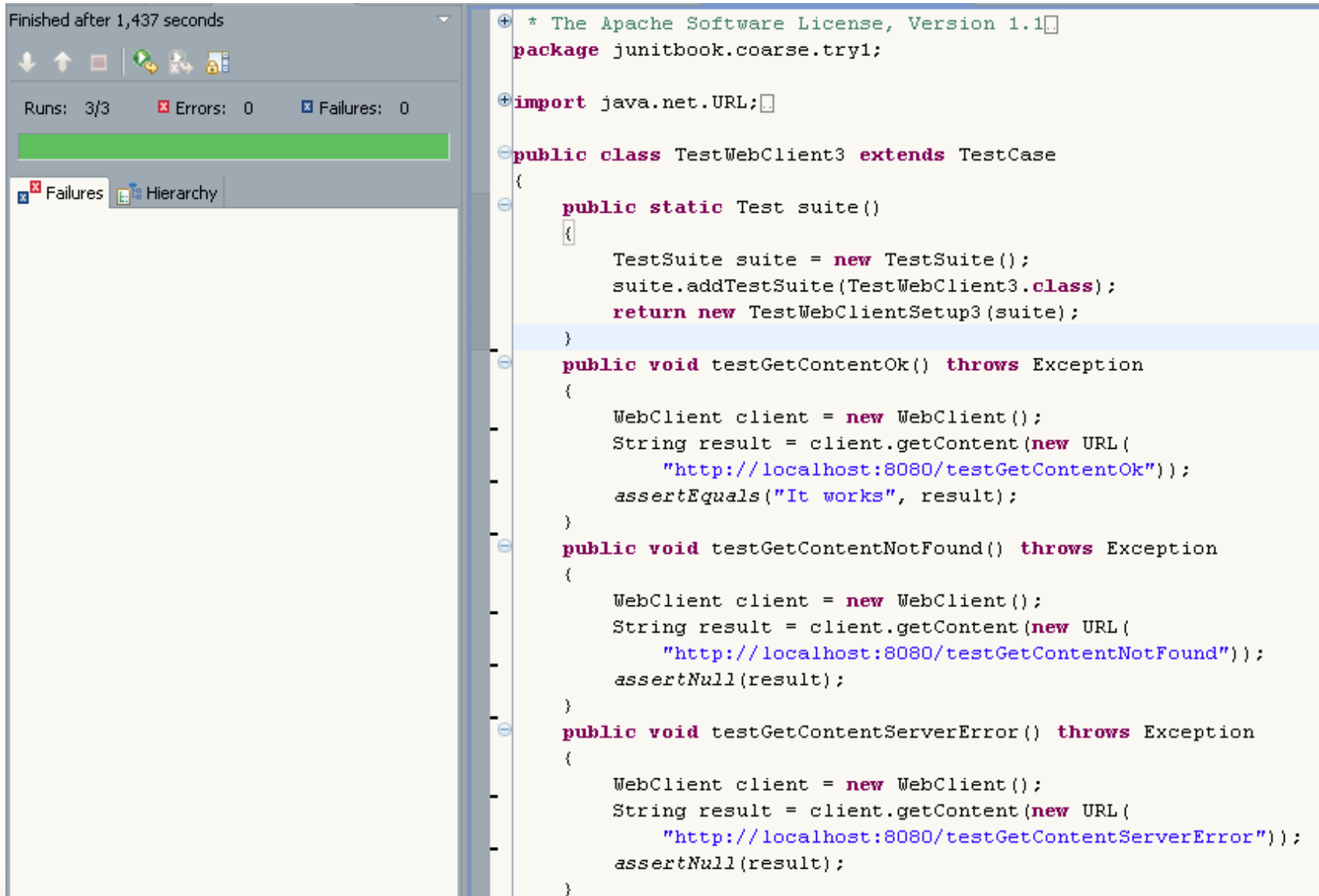
```
private class TestGetContentOkHandler extends AbstractHandler<Context> {  
    public void handle(String pathInContext, String pathParams,  
        HttpRequest request, HttpResponse response) throws IOException  
    {  
        OutputStream out = response.getOutputStream();  
        ByteArrayISO8859Writer writer =  
            new ByteArrayISO8859Writer();  
        writer.write("It works");  
        ...  
        request.setHandled(true);  
        ...  
    }  
}
```

- In der WebClient Klasse wird die angegebene Ressourcen nun zum Vergleich herunter geladen

- ```
public String getContent(URL url)
{
 ...
 HttpURLConnection connection = (HttpURLConnection)
 url.openConnection();
 connection.setDoInput(true);

 InputStream is = connection.getInputStream();

 byte[] buffer = new byte[2048];
 int count;
 while (-1 != (count = is.read(buffer))) {
 content.append(new String(buffer, 0, count)); }
 ...
}
```



Finished after 1,437 seconds

Runs: 3/3    Errors: 0    Failures: 0

Failures    Hierarchy

```
* The Apache Software License, Version 1.1
package junitbook.coarse.try1;

import java.net.URL;

public class TestWebClient3 extends TestCase
{
 public static Test suite()
 {
 TestSuite suite = new TestSuite();
 suite.addTestSuite(TestWebClient3.class);
 return new TestWebClientSetup3(suite);
 }

 public void testGetContentOk() throws Exception
 {
 WebClient client = new WebClient();
 String result = client.getContent(new URL(
 "http://localhost:8080/testGetContentOk"));
 assertEquals("It works", result);
 }

 public void testGetContentNotFound() throws Exception
 {
 WebClient client = new WebClient();
 String result = client.getContent(new URL(
 "http://localhost:8080/testGetContentNotFound"));
 assertNull(result);
 }

 public void testGetContentServerError() throws Exception
 {
 WebClient client = new WebClient();
 String result = client.getContent(new URL(
 "http://localhost:8080/testGetContentServerError"));
 assertNull(result);
 }
}
```

- Im folgenden wird die openConnection Funktion der Url Klasse überschrieben. Zuerst wird dabei die URLStreamHandlerFactory, welche Handler auf Ein bzw. Ausgabeströme zuweist, auf die selbst definierte Klasse gesetzt. Anschließend wird diese Handlerklasse ersetzt. Die HttpURLConnection Klasse liefert dann beim openConnection - Aufruf den neuen Inputstream zurück.

```
protected void setUp() {
 URL.setURLStreamHandlerFactory(new StubStreamHandlerFactory());
}
```

```
private class StubStreamHandlerFactory implements URLStreamHandlerFactory {
 public URLStreamHandler createURLStreamHandler(String protocol)
 {
 return new StubHttpURLConnection();
 }
}
```

```
private class StubHttpURLConnection extends URLStreamHandler {
 protected URLConnection openConnection(URL url) throws IOException
 {
 return new StubHttpURLConnection(url);
 }
}
```

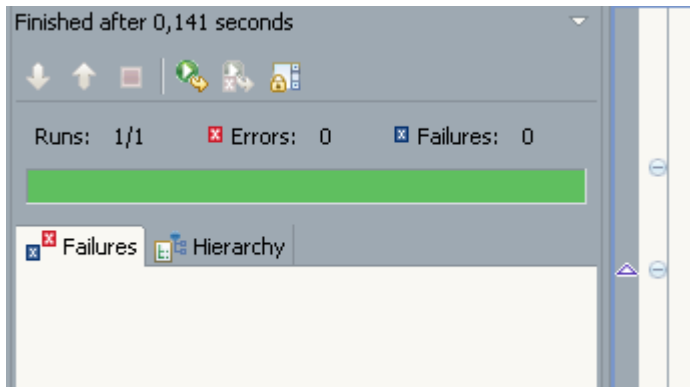
# Connection Stub

- Die benötigte Operation `getInputStream()` wird hier mit einem Stream überschrieben, der später getestet wird.

```
public class StubURLConnection extends HttpURLConnection
{
 ...
 public InputStream getInputStream() throws IOException
 {
 if (!isInput)
 {
 throw new ProtocolException(
 "Cannot read from URLConnection"
 + " if doInput=false (call setDoInput(true))");
 }
 ByteArrayInputStream bais = new ByteArrayInputStream(
 new String("It works").getBytes());

 return bais;
 }
 ...
}
```

- Hierbei kann eine Differenzierung der Ausgabe über die URL erreicht werden. Damit ist diese Version genauso Leistungsstark wie die Ressourcen Stub Variante.



```
public void testGetContentOk() throws Exception
{
 WebClient client = new WebClient();

 String result = client.getContent(
 new URL("http://jakarta.apache.org"));

 assertEquals("It works", result);
}
```

# Nachteile von Stubs

- **Abhängigkeit von der Umgebung** — Wenn der Server nicht gestartet ist und alle Funktionalität verfügbar ist → Es werden Fehler angezeigt, obwohl der Code funktioniert
- **separate Testlogik** — Die Logik ist einmal im jUnit Testfall und einmal auf der Webseite, die getestet werden soll hinterlegt → beide müssen ständig synchronisiert werden
- **Tests sind schwer zu automatisieren** → Der Code muss deployed werden und zusätzlich der Webserver gestartet werden → mit Ant oder Maven kann dieses Problem gelöst werden

# Einführung in Mockobjekte

- Mockobjekte werden verwendet, um das Verhalten von einzelnen Komponenten zu testen
- Dabei wird die Implementierung eines Interfaces bzw. einer Klasse vorgetäuscht, welche durch die Mockobjekte testbar wird.
- Beim Testen wird dann die Rückgabe des Mockobjektes, mit den eingestellten Werten bzw. mit dem angegebenen Verhalten verglichen.

# Einsatz von Mock Objekten

Können eingesetzt werden, wenn

- Das echte Objekt kein deterministisches Verhalten besitzt
- Das echte Objekt schwer zu konfigurieren ist
- Das echte Objekt Fehler erzeugt, die schwer aufzuspüren sind (Netzwerk Fehler)
- Das echte Objekt ist langsam bzw. hat Benutzereingaben
- Der Test hat Rückfragen an das echte Objekt (würden bestimmte Funktionen wirklich aufgerufen)

# Arten von Mockobjekten

1. Handgeschriebene Mockobjekte
  2. Generierte Mockobjekte (MockMaker, MochCreator)
  3. Dynamisch erzeugte Mockobjekte (EasyMock, JMock)
- Handgeschriebene Mockobjekte erfordern, dass ein Interface bzw. eine Klasse implementiert wird und dass die MockKlasse vor dem Testen kompiliert wird  
public class MockTableModelListener implements  
    TableModelListener ...  
→ Muss bei jeder Änderung des Interfaces / der Klasse neu geschrieben werden
  - Mit MockMaker kann der Code für die Mockobjekte automatisch erzeugt werden.
  - Die folgenden Beispiele verwenden EasyMock ([www.easymock.org](http://www.easymock.org)), wo Mockobjekte dynamisch zur Laufzeit erzeugt werden.  
→ Dabei wird die Klasse `java.lang.reflect.Proxy` benutzt

# Anwendung von EasyMock

- Zuerst wird ein Kontrollobjekt für alle benötigten Klassen erzeugt.  
`controlHttpServletRequest = MockControl.createControl(  
 HttpServletRequest.class);`
- Danach werden Instanzen der benötigten Klasse erzeugt, die dann durch das obige Objekte gesteuert werden können. (dynamischer Proxy)  
`mockHttpServletRequest =  
 (HttpServletRequest) controlHttpServletRequest.getMock();`
- Anschließend wird die erforderliche Verhaltensweise bei Anfragen, an die angelegten Objekte, festgelegt.  
`mockHttpSession.getAttribute("authenticated");  
controlHttpSession.setReturnValue(null);`
- Was einfach per Replay zur Laufzeit wiederholt wird.  
`controlHttpServletRequest.replay();  
controlHttpSession.replay();`

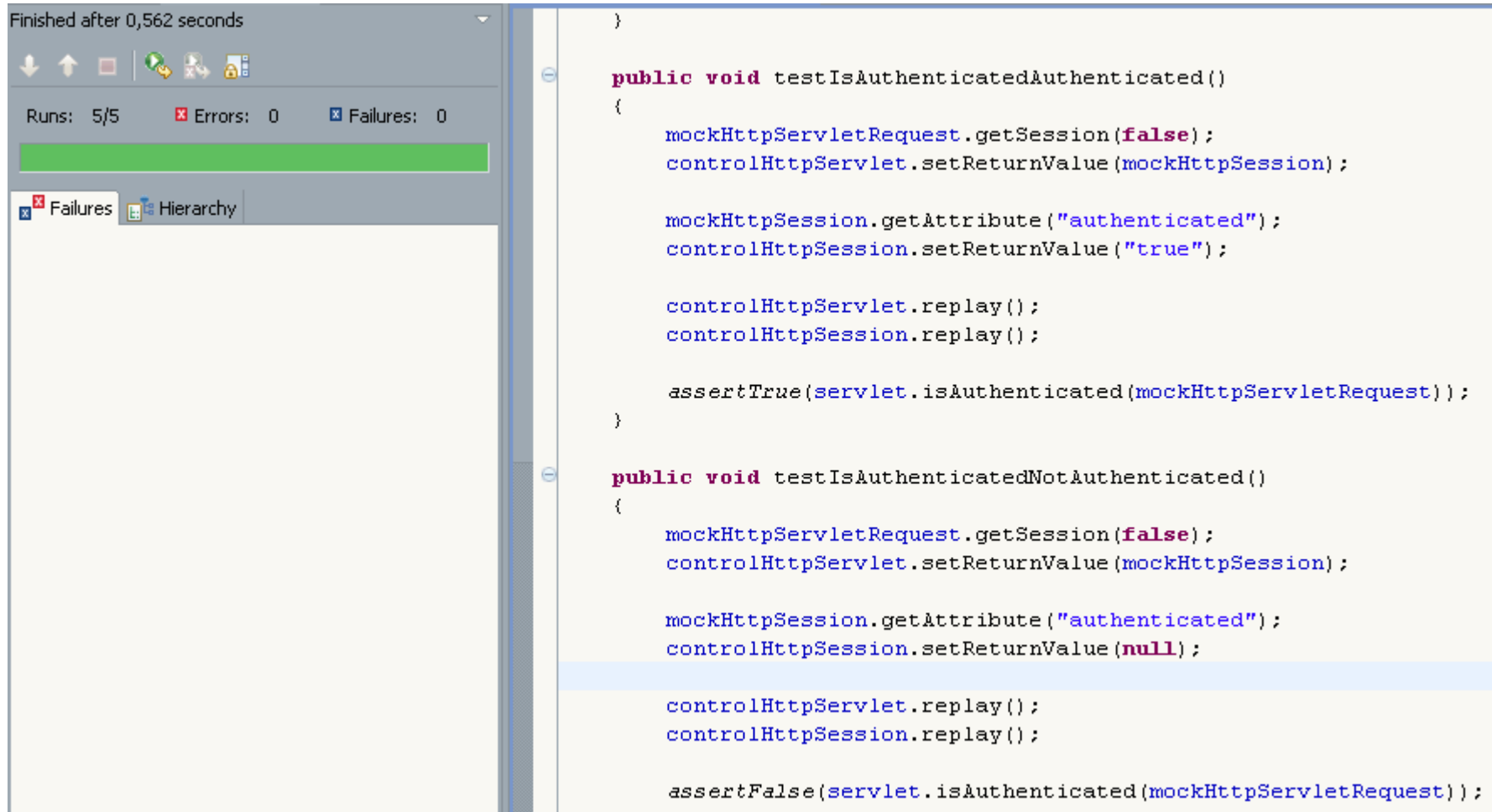
- Es handelt sich bei dem zu testenden Objekt um ein Servlet mit einer Authentifizierungsfunktion.

```
public class SampleServlet extends HttpServlet
{
 public boolean isAuthenticated(HttpServletRequest request)
 {
 HttpSession session = request.getSession(false);
 if (session == null)
 return false;
 String authenticationAttribute =
 (String) session.getAttribute("authenticated");
 return Boolean.valueOf(
 authenticationAttribute).booleanValue();
 } ...
}
```

- Der Befehl `mockHttpServletRequest.getSession(false)` bewirkt hier, dass das `HttpSession` Objekt (das eigentliche Mockobjekt) zurückgegeben wird.

```
public void testIsAuthenticatedNotAuthenticated()
{
 mockHttpServletRequest.getSession(false);
 controlHttpServletRequest.setReturnValue(mockHttpSession);
 mockHttpSession.getAttribute("authenticated");
 controlHttpSession.setReturnValue(null);

 controlHttpServletRequest.replay();
 controlHttpSession.replay();
 assertFalse(servlet.isAuthenticated(mockHttpServletRequest));
}
```



Finished after 0,562 seconds

Runs: 5/5    Errors: 0    Failures: 0

Failures    Hierarchy

```
}

public void testIsAuthenticatedAuthenticated()
{
 mockHttpServletRequest.getSession(false);
 controlHttpServlet.setReturnValue(mockHttpSession);

 mockHttpSession.getAttribute("authenticated");
 controlHttpSession.setReturnValue("true");

 controlHttpServlet.replay();
 controlHttpSession.replay();

 assertTrue(servlet.isAuthenticated(mockHttpServletRequest));
}

public void testIsAuthenticatedNotAuthenticated()
{
 mockHttpServletRequest.getSession(false);
 controlHttpServlet.setReturnValue(mockHttpSession);

 mockHttpSession.getAttribute("authenticated");
 controlHttpSession.setReturnValue(null);

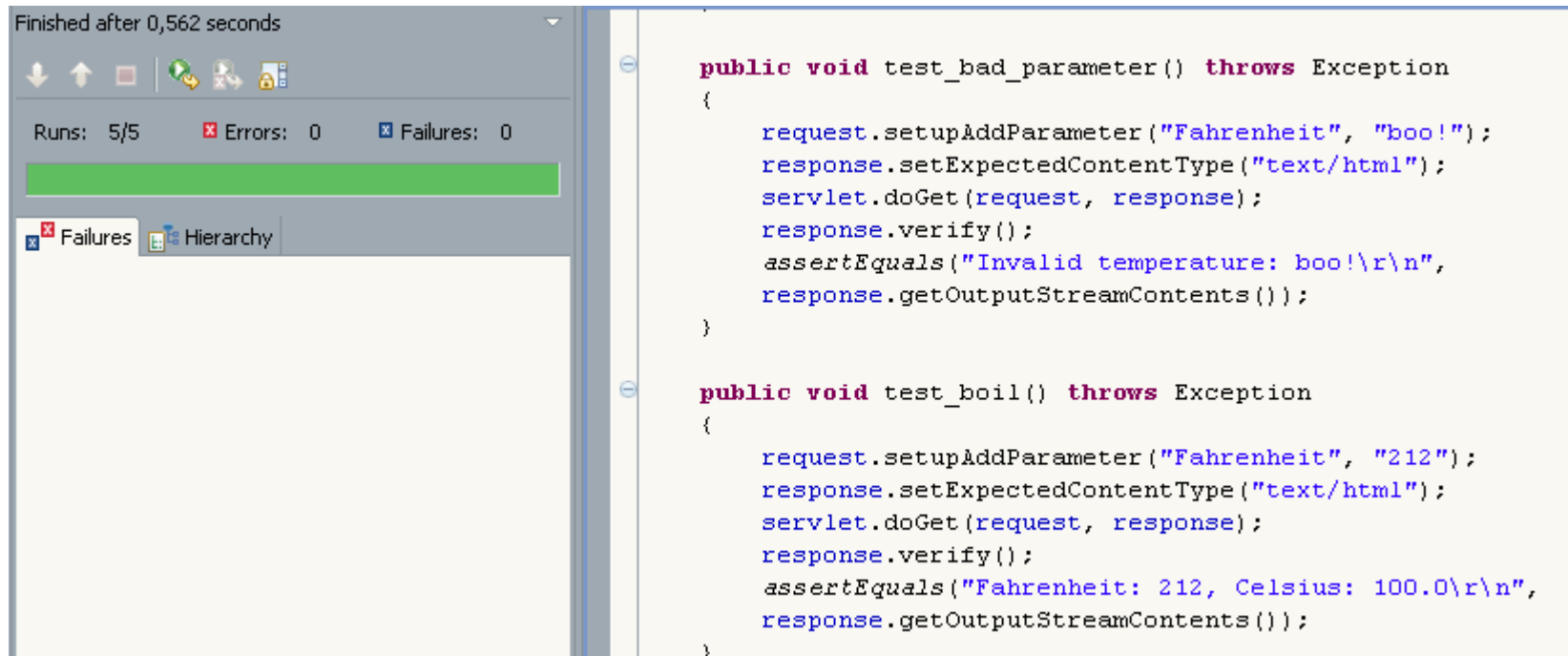
 controlHttpServlet.replay();
 controlHttpSession.replay();

 assertFalse(servlet.isAuthenticated(mockHttpServletRequest));
}
```

- Es ist aber auch möglich direkt ein Request- und ein Responsemockobjekt zu erzeugen.  
MockHttpServletRequest request;  
MockHttpServletResponse response;
- Zusätzlich können Parameter auch direkt gesetzt werden.  
void setupAddParameter(java.lang.String paramName,  
java.lang.String[] values)

- Es wird hier nun folgendes Servlet für Temperaturumrechnungen getestet.

```
public void doGet(HttpServletRequest req, HttpServletResponse res)
 throws ServletException, IOException
{
 String str_f = req.getParameter("Fahrenheit");
 res.setContentType("text/html");
 PrintWriter out = res.getWriter();
 try
 {
 int temp_f = Integer.parseInt(str_f);
 double temp_c = (temp_f - 32) * 5.0 / 9.0;
 out.println("Fahrenheit: " + temp_f + ", Celsius: " +
 temp_c);
 }
 ...
}
```



Finished after 0,562 seconds

Runs: 5/5    ✘ Errors: 0    ✘ Failures: 0

✘ Failures    ✘ Hierarchy

```
public void test_bad_parameter() throws Exception
{
 request.setupAddParameter("Fahrenheit", "boo!");
 response.setExpectedContentType("text/html");
 servlet.doGet(request, response);
 response.verify();
 assertEquals("Invalid temperature: boo!\r\n",
 response.getOutputStreamContents());
}

public void test_boil() throws Exception
{
 request.setupAddParameter("Fahrenheit", "212");
 response.setExpectedContentType("text/html");
 servlet.doGet(request, response);
 response.verify();
 assertEquals("Fahrenheit: 212, Celsius: 100.0\r\n",
 response.getOutputStreamContents());
}
```

# Unterschiede zwischen Stub und Mock

- Mockobjekte haben keine Businesslogik
- Mockobjekte wiederholen die vorgegebene Verhaltensweise
- Man kann Anforderungen an das Objekt hinzufügen (Expectedcontenttype)
- Das Verhalten des Objektes kann verifiziert werden (wurde die Methode ausgeführt)
- Testen der Zusammenarbeit zwischen Objekten (Interaction-based)
  
- Stubs implementieren die gesamte Logik, die zum Testen des Objektes nötig ist
- Stub-Objekte sind Platzhalter für geplante, aber noch nicht umgesetzte Funktionalität
- Zum Testen von Zuständen bzw. genau definiertes Verhalten (State-based)

# Bewertung von Mock-Objekten

## Vorteile:

- Braucht keinen eigenen Container, in dem der Code ausgeführt werden muss
- Tests können für einzelne Objekte bzw. Methodenaufrufe ausgeführt werden → Granularität ist sehr fein
- Es muss nicht die gesamte spätere Laufzeitumgebung simuliert werden
- Der Code wird durch die frühe Testmöglichkeit verbessert
- Einfache Handhabung

## Nachteile:

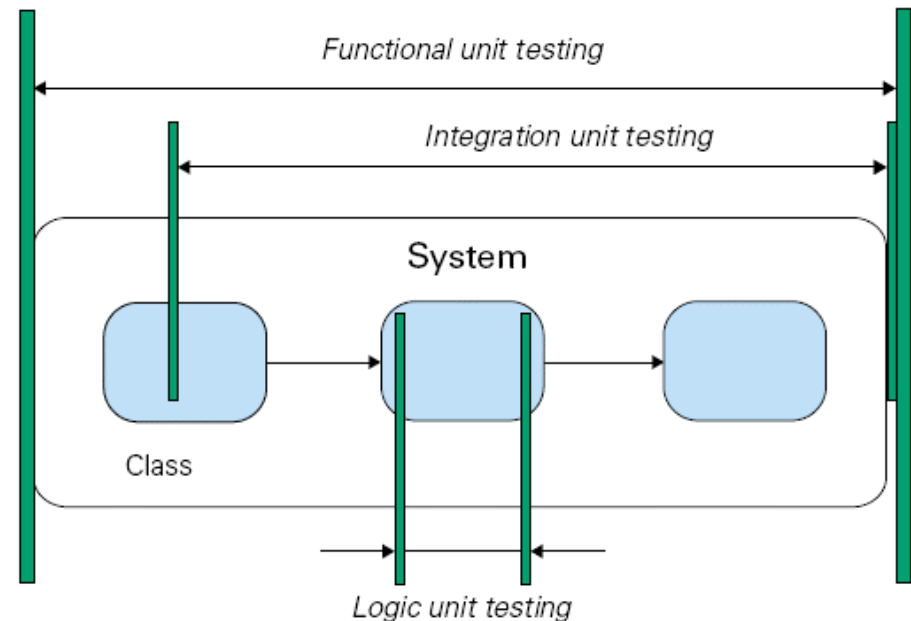
- Es wird nicht in der Umgebung getestet, wo der Code ausgeführt wird → keine Interaktionsmöglichkeiten bzw. keine Testmöglichkeit auf evt. Seiteneffekte
- Man muss die Laufzeitumgebung sehr genau kennen. Der folgende Code funktioniert innerhalb eines Tomcat Servers, aber bspw. Ein OrionServer erwartet einen anderen Wert (Text/html).

```
public void doGet(HttpServletRequest request,
 HttpServletResponse response)
{
 response.setContentType("text/xml");
}
```

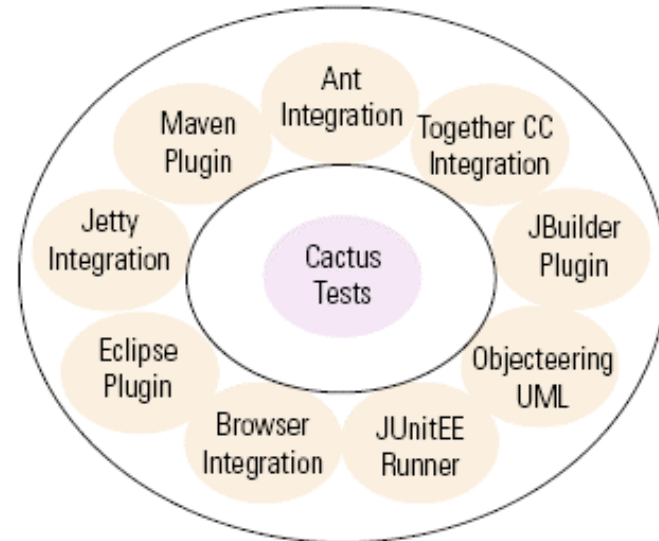
- Es nicht möglich new HttpRequest innerhalb des Codes zu erzeugen, da es sich um „out-Container Testen“ handelt.

# Testen innerhalb von Containern

- Funktionale Test treffen Aussagen über die Interaktion zwischen Methoden
- Die Überprüfung der logischen Einheiten kann nur auf Methodenebene stattfinden. (Mockobjekte)
- Bei Integrationstests handelt es sich um Generalisierungen von beiden Varianten

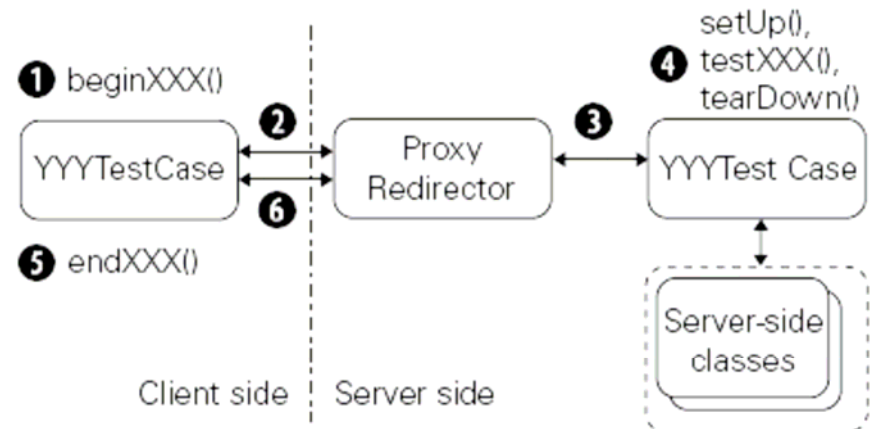


- Das Ausführen von Integrationstests geht über das normale jUnit Testen hinaus.
- Die Tests bzw. der zu testenden Quellcode müssen in Packages gebracht werden, deployed werden und anschließend innerhalb des Containers gestartet werden.
- Frontends, wie Ant integration, das Maven plugin, und die Jetty integration verbergen diese Komplexität.



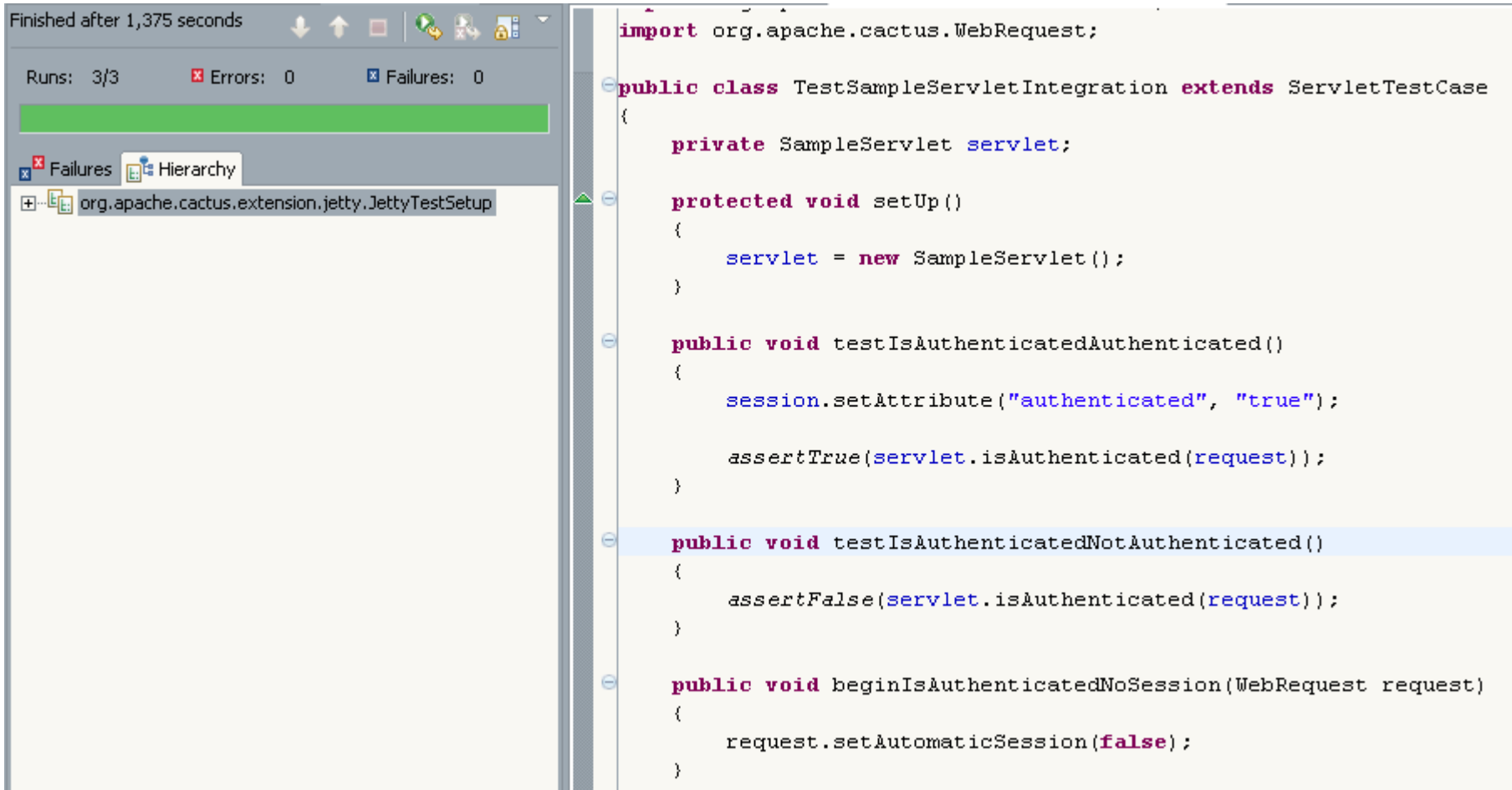
# Cactus Lebenszyklus

1. Beginxxx erlaubt es Informationen an den Redirector zu schicken (Initialisierung von Cookies...)
2. Verbindung zum Redirektor wird aufgebaut. Der Proxy Redirector ist dabei lediglich ein normales Servlet.
3. Es wird eine Instanz der Testklasse auf dem Server erzeugt. Die Ergebnisse werden in einem Objekt namens ServletConfig gespeichert.
4. Die Methoden setUp, testXXX und tearDown werden ausgeführt
5. endXXX räumt die Daten auf bzw. wertet diese aus (bspw. Cookies)
6. Ergebnisse der Test werden ausgewertet



- Die Einstellung des Jetty Servers erfolgt analog zu den Stubs.

```
public class TestAllWithJetty
{
 public static Test suite()
 {
 System.setProperty("cactus.contextURL",
 "http://localhost:8080/test");
 TestSuite suite = new TestSuite("All tests with Jetty");
 suite.addTestSuite(TestSampleServletIntegration.class);
 return new JettyTestSetup(suite);
 }
}
```



The screenshot displays an IDE interface. On the left, a test runner window shows the results of a test execution. It indicates that the test finished after 1,375 seconds, with 3/3 runs completed, 0 errors, and 0 failures. Below this, a tree view shows the test setup class: `org.apache.cactus.extension.jetty.JettyTestSetup`.

On the right, the source code for the `TestSampleServletIntegration` class is shown. The code is as follows:

```
import org.apache.cactus.WebRequest;

public class TestSampleServletIntegration extends ServletTestCase
{
 private SampleServlet servlet;

 protected void setUp()
 {
 servlet = new SampleServlet();
 }

 public void testIsAuthenticatedAuthenticated()
 {
 session.setAttribute("authenticated", "true");

 assertTrue(servlet.isAuthenticated(request));
 }

 public void testIsAuthenticatedNotAuthenticated()
 {
 assertFalse(servlet.isAuthenticated(request));
 }

 public void beginIsAuthenticatedNoSession(WebRequest request)
 {
 request.setAutomaticSession(false);
 }
}
```

## Nachteile:

- Es werden spezielle Tools für jedes Framework gebraucht, in dem die Umgebungen angesprochen werden können
- Es benötigt längere Zeit, um die Tests auszuführen, da immer erst der Server gestartet werden muss.
- Man benötigt eine komplexe Konfiguration, da die Anwendung in Packages überführt und deployed werden muss.

## Argumente gegen die „Nachteile“:

- Das ständige Deployen kann als Übung angesehen werden
- Das komplizierte Einrichten fällt eigentlich nur einmal vor dem ersten Start an

- Es müssen die Skripte zum Packen, Deployen und zum Starten des Servers nicht selber geschrieben werden (im Gegensatz zu Ant), da sie von Maven direkt erzeugt werden können.
- Nachdem `maven cactus:test` in einer Shell ausgeführt worden ist, werden die Java Daten kompiliert, in War-Dateien gepackt und in den angegebenen Server geschoben.
- Maven bietet die Möglichkeit automatisch Htmlseiten für das Projekt zu erzeugen, wo auch die Testergebnisse zu finden sind.

```
cactus:test-war:
[cactus] -----
[cactus] Running tests against Tomcat 4.1.31
[cactus] -----
[cactus] Deleting 24 files from C:\DOKUME~1\tR\LOKALE~1\Temp\cactus\tomcat4x
[cactus] Deleted 14 directories from C:\DOKUME~1\tR\LOKALE~1\Temp\cactus\tomcat4x
[cactus] Running junitbook.servlets.TestAdminServlet
[cactus] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 1,282 sec
[cactus] Testsuite: junitbook.servlets.TestAdminServlet
[cactus] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 1,282 sec
[cactus]
[cactus] Testcase: testGetCommandOk took 0,984 sec
[cactus] Testcase: testGetCommandNotDefined took 0,047 sec
[cactus] Testcase: testCallView took 0,078 sec
[cactus] Testcase: testDoGet took 0,079 sec
[cactus] Running junitbook.servlets.TestSecurityFilter
[cactus] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 4,75 sec
[cactus] Testsuite: junitbook.servlets.TestSecurityFilter
[cactus] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 4,75 sec
[cactus]
[cactus] Testcase: testDoFilterAllowedSQL took 0,703 sec
[cactus] Testcase: testDoFilterForbiddenSQL took 3,906 sec
```

## junitbook.servlets

| Class                              | Tests | Errors | Failures | Success Rate | Time  |
|------------------------------------|-------|--------|----------|--------------|-------|
| <a href="#">TestAdminServlet</a>   | 4     | 0      | 0        | 100,00%      | 1,282 |
| <a href="#">TestSecurityFilter</a> | 2     | 0      | 0        | 100,00%      | 4,750 |

## Test Cases

[ [summary](#)] [ [package list](#)] [ [test cases](#)]

### TestAdminServlet

|                                          |         |      |
|------------------------------------------|---------|------|
| <a href="#">testGetCommandOk</a>         | Success | 0,98 |
| <a href="#">testGetCommandNotDefined</a> | Success | 0,05 |
| <a href="#">testCallView</a>             | Success | 0,08 |
| <a href="#">testDoGet</a>                | Success | 0,08 |

### TestSecurityFilter

|                                          |         |      |
|------------------------------------------|---------|------|
| <a href="#">testDoFilterAllowedSQL</a>   | Success | 0,70 |
| <a href="#">testDoFilterForbiddenSQL</a> | Success | 3,91 |

## Vorteile:

- Testen unterstützt die Entwicklung von guten Code
- In der Regel sind die Frameworks nicht schwer aufzusetzen

## Nachteile:

- Eigentlich nur sinnvoll wenn man früh beginnt
- Teilweise ist aber auch zuviel Wissen über die Funktionsweise der Testumgebung nötig
- Black Box Testen ist nicht immer möglich