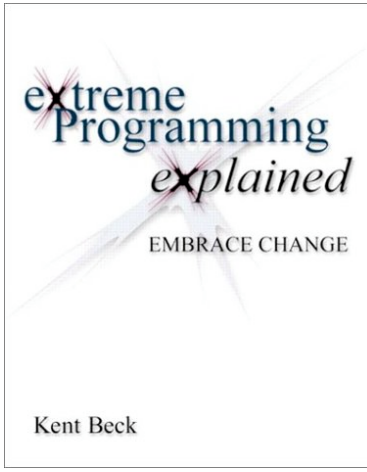
JUnit

Programming by Contract

Frank.Weberskirch@entory.com

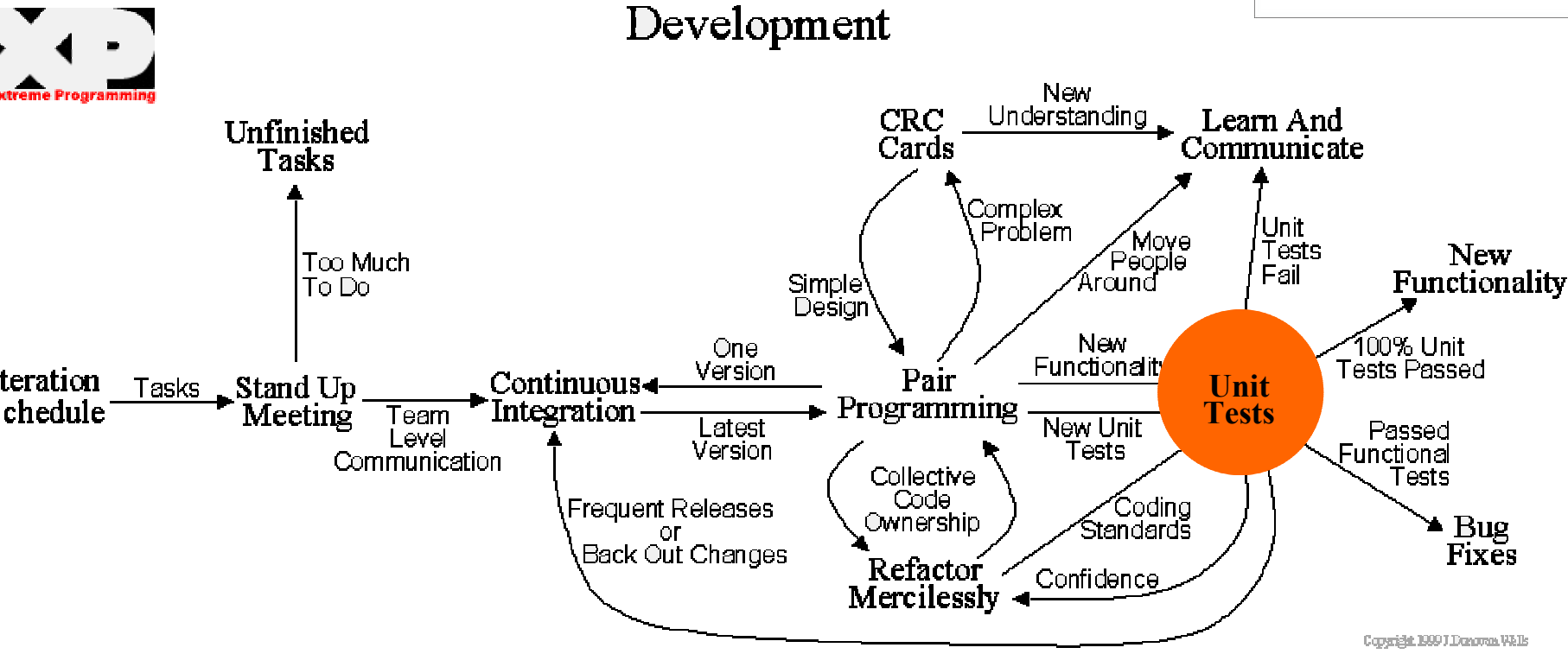
entory

JUnit: Programming by Contract



JUnit:

- **Programming by Contract**
- **eXtreme Testing**
- **eXtreme Programming**



entory

JUnit: Programming by Contract

Programming by Contract

- Konsequente Einsatz von Unit Tests und Functional Tests in allen Phasen der Entwicklung
  - Unit Tests:
    Test von Einzelkomponenten (Entwickler)
  - Functional Tests:
    Test von kompletten Funktionsblöcken bzw. Test der ganzen Anwendung (Kunde)

Die Grundidee

- Der Test wird geschrieben noch bevor mit der eigentlichen Implementierung begonnen wird.
- Zu jeder Komponente (Unit) existieren Testroutinen die automatisiert und unabhängig voneinander ausgeführt werden können.

JUnit: Programming by Contract

Welchen Nutzen bringen Unit Tests?

- **Änderungen am Code können sofort überprüft werden**
- **Mehr Sicherheit bezüglich der Qualität des Codes**
- **Weniger Angst vor Änderungen an Kernkomponenten**
- **Entwicklungszeiten werden kürzer (trotz der zusätzlich zu implementierenden Tests)**
- **Einfacheres Überarbeiten (Refactoring) alter Codesegmente (Für alten Code wird bei Bedarf zuerst ein Test geschrieben und dann die Änderung durchgeführt.)**

JUnit: Programming by Contract

Problem:

➔ **Vorraussetzungen für die Akzeptanz der Unit Tests**

- **einfache Implementierung der Tests,**
- **einfache und schnelle Ausführung,**
- **einfache Analyse der Ergebnisse.**

Lösung:

➔ **Für Java: JUnit**

Ein Test-Framework, das genau die oben geforderten Erleichterungen für den Entwickler bietet.

JUnit: Die Idee

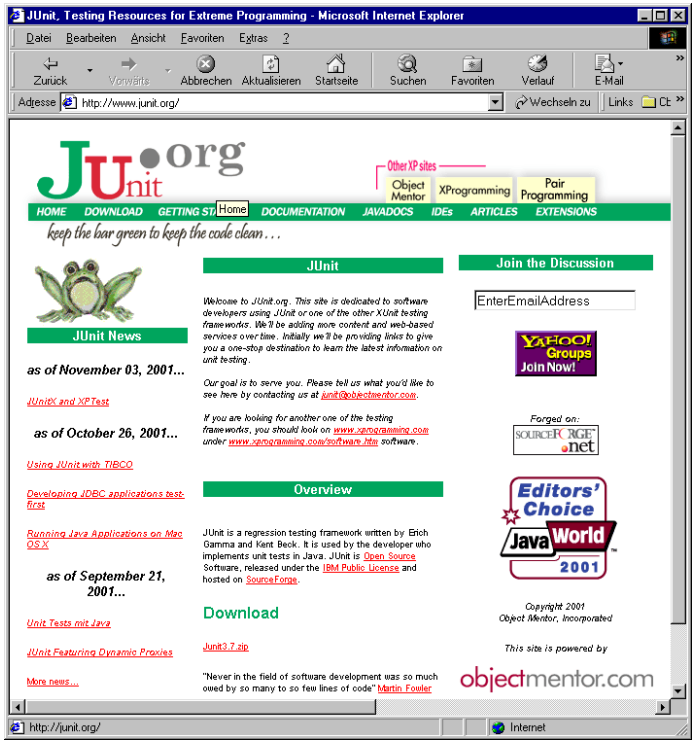
Was ist JUnit?

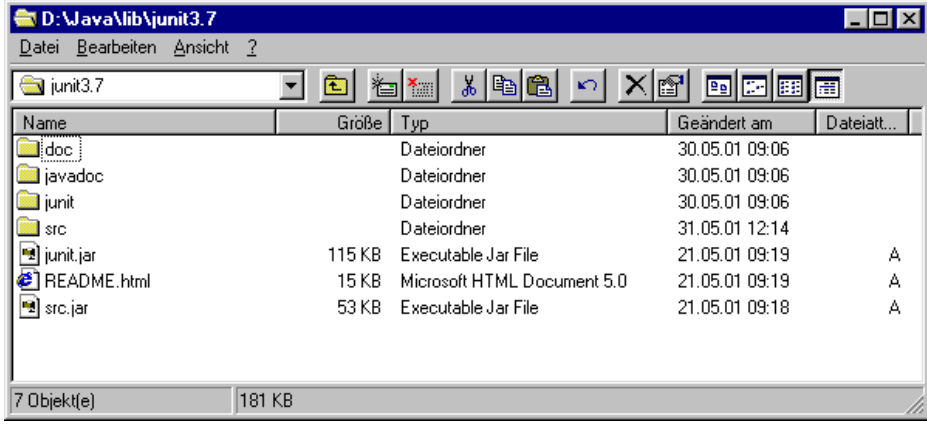
- **Framework zum Aufbau von Unit Tests**
  - Bestandteile: Java-Klassen und Interfaces zum Aufbau und zum Ausführen von Unit Tests
  - Einfachheit bei Implementierung und Analyse der Ergebnisse
  - Schnelligkeit bei Ausführung von Tests
- **Open Source**
- **Autoren: Erich Gamma und Kent Beck**

JUnit: Die Idee

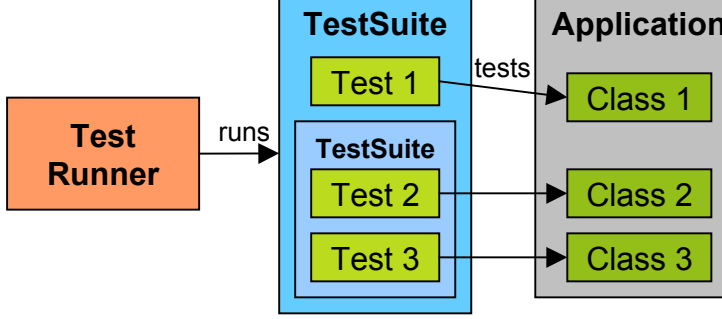
Wie beginnt man mit JUnit?

- **Homepage:** `http://www.junit.org`
- **Download:** `[junit3.7.zip]` `junit3.8.1.zip`
- **Inhalt:**
  - **JAR-Archiv** `junit.jar`,
  - **Source Code,**
  - **API-Dokumentation,**
  - **Dokumentation zu JUnit**





JUnit: Die Idee



- **Grundkonzepte: Test Cases und Test Suites**
  - **Unit Test = Subklasse von `junit.framework.TestCase`**
  - **Test Cases lassen sich zu einer größeren Einheit kombinieren, einer sog. TestSuite**
  - **Test Suites lassen sich innerhalb anderer Test Suites einbinden (zu beliebig tiefen hierarchischen Strukturen von TestCases und TestSuites)**
  - **TestRunner arbeitet Tests (d.h. TestCases oder TestSuites) nach einheitlichem Muster ab**

Class1Test

setup()
tearDown()

testFunction1
testFunction2

- **Bestandteile eines Unit Tests**
  - **Name: idealerweise nach dem Muster *Test*, z.B. FileReaderTest**
  - **Fixture: Mechanismus zum Herstellen von Rahmenbedingungen**
  - **Testmethoden: idealerweise beginnend mit „test“**

JUnit: Die Idee

Class1Test
extends TestCase

main()
suite()

setup()
tearDown()

testFunction1
testFunction2

Schreiben eines Unit Tests: Was ist zu tun?

- **Definiere Unterklasse von `TestCase`.**
- **Rahmen für Tests (Fixture)**
  - **Überschreibe Methode `setUp()` zum Initialisieren von Objekte, die in Tests eine Rolle spielen**
  - **Überschreibe Methode `tearDown()` um Objekte aus Tests wieder freizugeben.**
- **Definiere eine oder mehrere Methoden `testXXX()`, die auf den erzeugten Objekten die konkreten Tests ausführen.**
- **Definiere Klassenmethode `suite()`, welche eine Test Suite erstellt, die alle `testXXX()` Methoden enthält.**
- **Definiere die `main()` Methode, die den Test Case ausführt.**

JUnit: Praktische Anwendung

Beispiel: Unit Test für Java-Klasse `FileReader`

- **Testdatei „data.txt“**

```
Testdatei für FileReader
Zeile 2 der Testdatei
Zeile 3 der Testdatei

Letzte Zeile der Testdatei
```

- **Unit Test als Subklasse von `junit.framework.TestCase`**

```
class FileReaderTest extends TestCase {

public FileReaderTest(String name) {
          super(name);
     }
}
```

- **Konstruktor obligatorisch (Adapter Pattern bei TestCase)**

JUnit: Praktische Anwendung

- **Fixture ist optional (Template Pattern)**
  - `setUp()` schafft einheitliche Voraussetzung für die Durchführung eines Tests, z.B. Erzeugung von Objekten, Öffnen von Netzwerkverbindungen, ...
  - `tearDown()` räumt wieder auf, d.h. entfernt Objekte wieder, schließt Netzwerkverbindungen wieder

- **Beispiel:**

```
class FileReaderTest extends TestCase
...
    protected void setUp() {
        try {
            input = new FileReader(„data.txt");
        } catch(FileNotFoundException e) {
            throw new RuntimeException(„unable to open test file");
        }
    }

    protected void tearDown() {
        try {
            input.close();
        } catch(IOException e) {
            throw new RuntimeException(„unable to close test file");
        }
    }
}
```

JUnit: Praktische Anwendung

Testmethoden eines Unit Tests

- **Ein oder mehrere Testmethoden nach dem Muster**

```
public void testXXX(){
      ...
}
```

- **Abarbeitung von Tests: (automatisch durch Klasse TestCase)**
  1. **Aufruf von `setup()`**
  2. **Aufruf von `testXXX()`**
  3. **Aufruf von `tearDown()`**
- **Testmethoden sollten von einander unabhängig sein, da i.A. keine Reihenfolge bei der Testabarbeitung vorausgesetzt werden kann.**

JUnit: Praktische Anwendung

Prüfmechanismen in Testmethoden

- **Ziel von Unit Tests:**
  - Tests laufen vollautomatisch, nur bei Fehlern wird abgebrochen und der Fehler angezeigt.
  - Zur Erkennung von Fehlern werden Annahmen über Zustände von Objekten mit Hilfe „assertions“ in den Testmethode programmiert
- **Assertions („Behauptungen“) werden innerhalb der Methoden erfolgt durch Varianten folgender Methoden ausgedrückt:**

```
assertEqual( Object, Object )
assertSame( Object, Object )
assertNull( Object )
assertNotNull( Object )
assertTrue( boolean )
```

- **Wird beim Testlauf eine Assertion nicht erfüllt, bricht der Test mit einer Fehlermeldung ab.**

JUnit: Praktische Anwendung

- **Beispiel: Testmethod für Unit Test „FileReaderTest“**

```
public void testRead() throws IOException {
    char ch = ´´;
    for (int i=0, i<5; i++)
          ch = (char) input.read();
    assertSame(ch, ,´d´);
}
```

- **Was passert bei assertSame?**
  - **Testmethode liest bis zum 5. Zeichen aus der Datei**
  - **Bei der vorgegebenen Testdatei muss dann der Buchstabe ‚d‘ gelesen worden sein, ansonsten wir Fehler gemeldet.**
- **Erweiterung (zum leichteren Lokalisieren im Code)**

```
assertSame(String, Object, Object)
```

- **Beispiel:**

```
assertSame(„5. Zeichen nicht korrekt", ch, ‚d')
```

JUnit: Praktische Anwendung

Wie startet man einen Unit Tests?

- Definition einer (zunächst) minimalen Test Suite
- Abarbeiten der Test Suite mit einem sog. `TestRunner` (mit einfacher Textausgabe oder Swing GUI)
- Beispiel: (inkl. Textausgaben)

```
class FileReaderTest ...

    public static main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(
                new FileReaderTest(„testRead“));
        return suite;
    }
}
```

```
.
Time: 0.123


OK (1 tests)
```

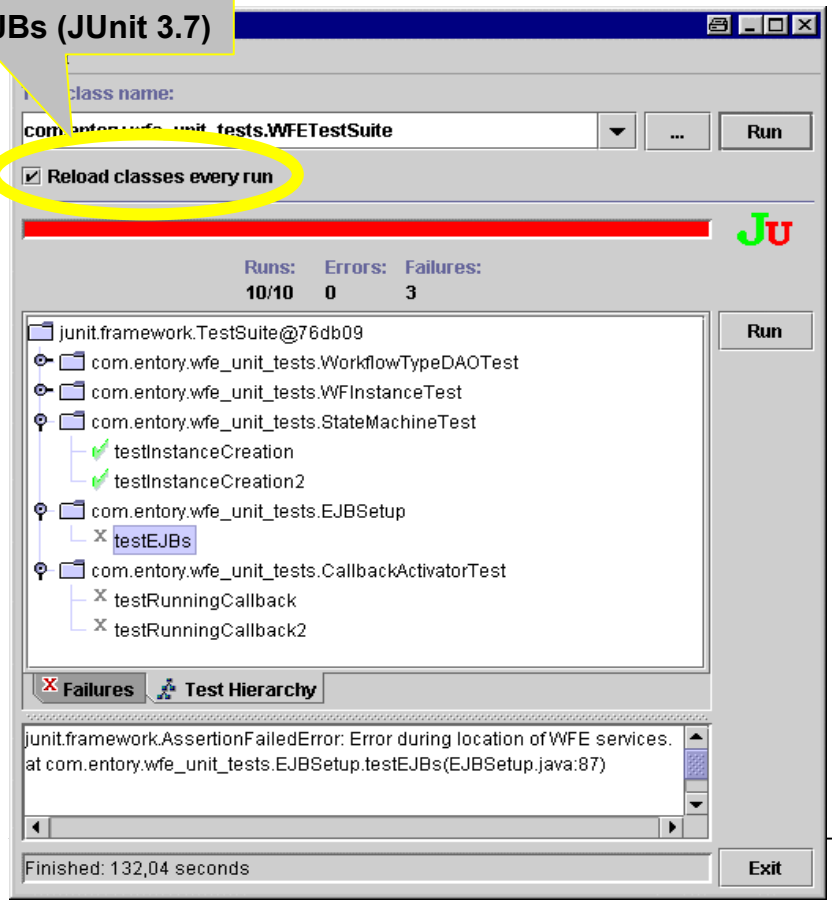
```
.F
Time: 0.23

!!!FAILURES!!!

Test Results:
Run: 1 Failures: 1 Errors: 0

There was 1 failure:
1) FileReaderTest.testRead
test.framework.AssertionFailedError
```

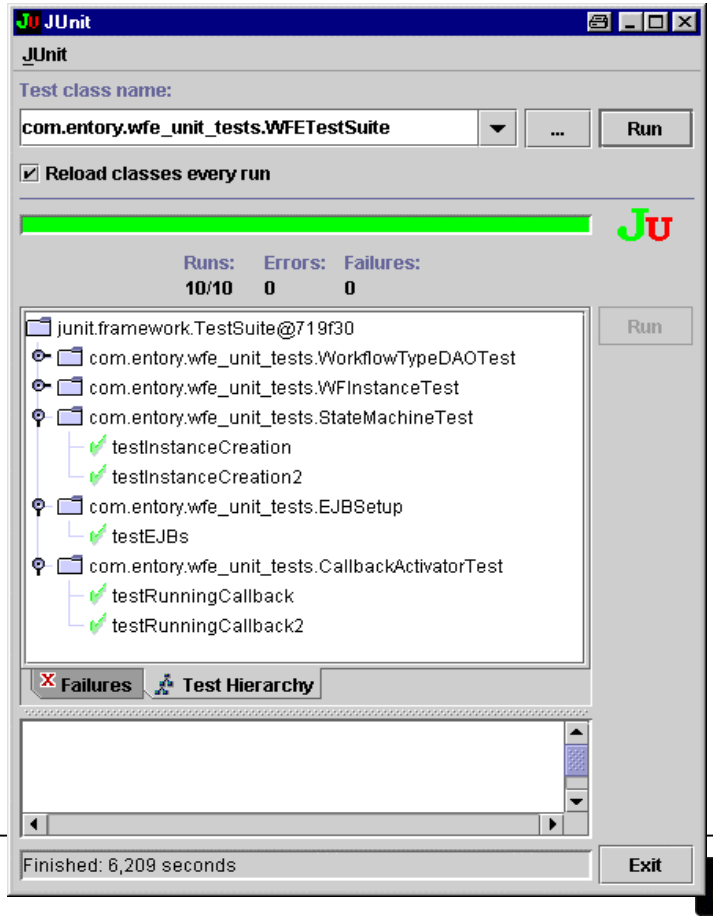
JUnit: Praktische Anwendung

Graphisches Tool für Test-ausführung und Auswertung

```
public static main(String[] args) {
    String[] testName =
            {FileReaderTest.class.getName()};
    junit.swingui.TestRunner.main(testName);
}
```

roblematisch bei
EJBs (JUnit 3.7)





entory

JUnit: Praktische Anwendung

Verschiedene Arten von Fehlern

- **Failures:**
  - ➔ In einer Testmethode wurde eine Assertion nicht erfüllt.
- **Errors:**
  - ➔ In einer Testmethode wurde eine (unerwartete) Exception geworfen.

```
public void testRead() throws IOException {
    char ch = ,‘;
    input.close();
    for (int i=0, i<5; i++)
        ch = (char) input.read();  // will throw exception
    assertSame(ch, ,d‘);
}
```

```
.E
Time: 0.15

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 0 Errors: 1
There was 1 error:
1) FileReaderTest.testRead
java.io.IOException: Stream closed
```

JUnit: Praktische Anwendung

Tests zur Überprüfung, dass Exceptions geworfen werden

```
public void testReadAfterClose() throws IOException {
    input.close();
    try {
        input.read();
        fail(„no exception for read past end“);
    } catch(IOException ioex) {}
}
```

- **Erkennung von Fehlern mittels `fail(String)`:**
  Markierung von Stellen im Code, die bei korrekter Funktionsweise nicht erreicht werden dürfen

JUnit: Praktische Anwendung

Granularität von Unit Tests

- **Jede Klasse mit „kritischer Funktionalität“ wird über Unit Tests geprüft, d.h. Unit Test prüfen typischerweise sehr lokal.**
- **Jede Funktionalität wird mit einer Test-Methode `testXXX()` abgedeckt**
- **Jeder Fehler, der in Software gefunden wird, führt zu neuem Unit Test, der diesen Fehler offenbart**

Effekt: Man schafft sich bei der Entwicklung die Möglichkeit, nachzuweisen, dass Funktionalitäten erhalten bleiben.

Nutzung von JUnit für Funktionale Tests (Use Case Tests)

- **Ziel hier nicht Test einzelner Klassen sondern größerer typischer Abläufe innerhalb von Software (Use Cases)**

JUnit: Praktische Anwendung

Hierarchische Strukturierung von Unit Tests in Suites

- Beliebig tiefe Hierarchien möglich

Empfehlungen:

- Eine Test Suite pro Java Package
- Build-Prozess sollte Unit Tests immer direkt mit übersetzen (Konsistenz der Tests mit Sourcen)

Beispiel-Hierarchie von Unit Tests:

```
MasterTestSuite - Top-level test suite
  SmokeTestSuite - Structural integrity tests
    EcommerceTestSuite
      ShoppingCartTestCase
      CreditCartTestSuite
        AuthorizationTestCase
        CaptureTestCase
        VoidTestCase
      UtilityTestSuite
        MoneyTestCase
    DatabaseTestSuite
      ConnectionTestCase
      TransactionTestCase
  LoadTestSuite - Scalability tests
    DatabaseTestSuite
      ConnectionPoolTestCase
    ThreadPoolTestCase
```

JUnit: Praktische Anwendung

Wie erstellt man eine Test Suite?

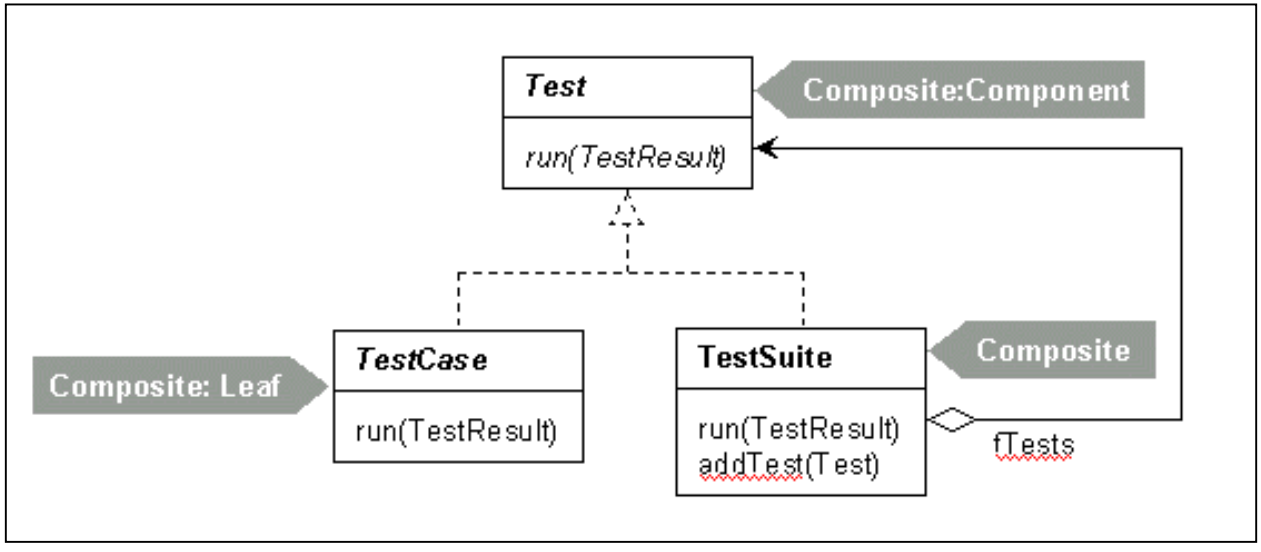
- Definiere Unterklasse von `TestCase`.
- Definiere Klassenmethode `suite()`, welche eine Test Suite erstellt, die alle Tests enthält, die zusammengefasst werden.
- Definiere die `main()` Methode, die die Test Suite ausführt.

```
class FileIOSuite extends TestCase {

    public static main(String[] args) {
         String[] testName = { FileIOSuite.class.getName() };
        junit.swingui.TestRunner.main(testName);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new TestSuite(FileReaderTest.class));
        return suite;
    }
}
```

JUnit: Architektur

Framework aufgebaut nach verschieden Design Pattern

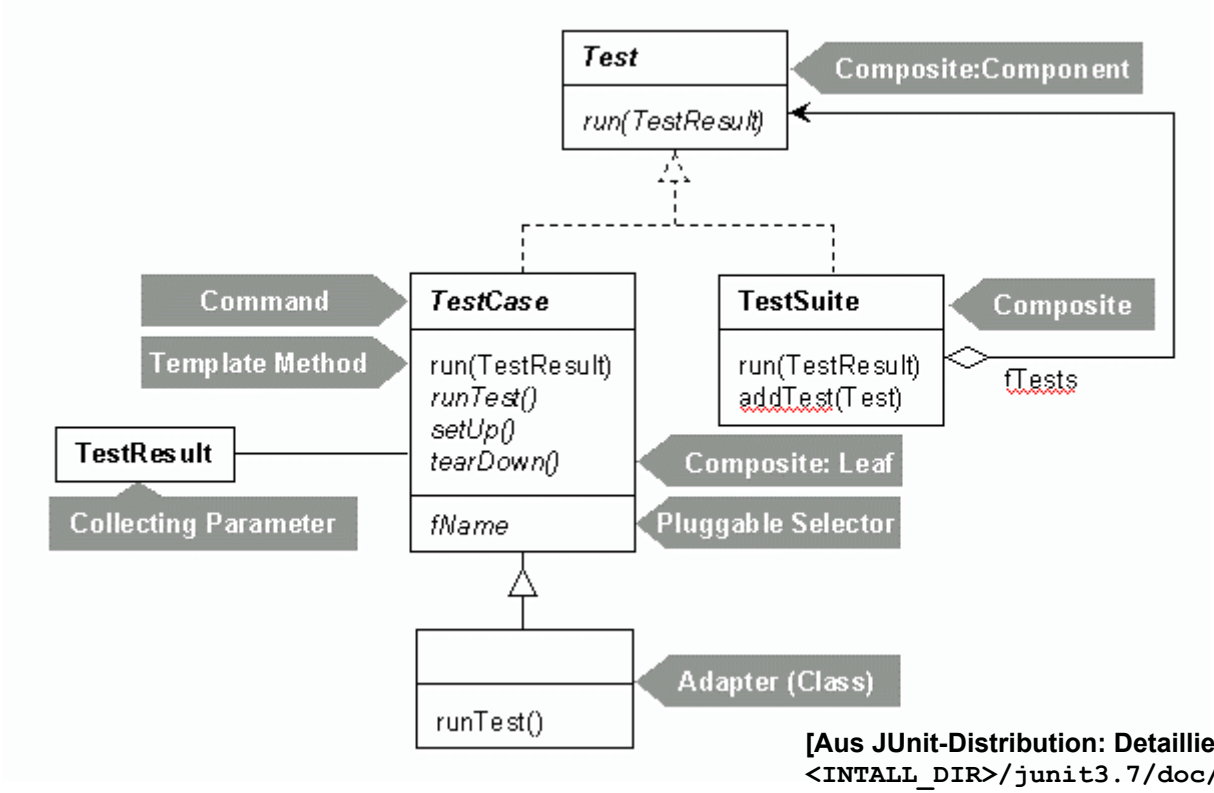
- **Composite Pattern bei TestSuite ermöglicht beliebig tiefe Hierarchien von TestSuites und TestCases**



"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." **[Gamma et al., 1995]**

JUnit: Architektur

- **Weitere Design Pattern bei JUnit Implementation: Command, Adapter, Pluggable Selector, Template Method, ...**



[Aus JUnit-Distribution: Detaillierte Erläuterungen in `<INTALL_DIR>/junit3.7/doc/cookstour/cookstour.htm`]

JUnit: Erweiterungen

- Erweiterungen von JUnit
  - HTTPUnit: Blackbox Web Testing
  - JUnitEE: Unit Testing in J2EE
  - Cactus: Serverseitiges Testen
    `http://jakarta.apache.org/cactus/`
  - Siehe auch:
    `http://www.junit.org/news/article`
- JUnit Best Practices
  `http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html`
- JUnit für andere Programmiersprachen

JUnit: Erfahrungen

Erfahrungen entory AG

- Component: Source Code + Unit Tests + Use Case Tests
- Wie entsteht Source Code?
  - Anforderungen und Identifikation von Use Cases
  - Grobdesign von Klassen
  - Festlegung der Funktionalität von Klassen, d.h. Methoden und ihre Signatur (grob)
  - Unit Test für eine Klasse mit einzelnen Testmethode: Entwickler macht sich damit klar, was realisiert werden soll
  - Nachdem der Test geschrieben ist, wird die Funktionalität realisiert, d.h. die Methode implementiert
  - Permanenter Zyklus: Testen, Funktionalität entwickeln, Testen, Korrigieren, bei Bedarf Test korrigieren/erweitern, Testen, Funktionalität korrigieren, ...
  - Schließlich: Tests laufen fehlerfrei, d.h. Funktionalität ist realisiert.

JUnit: Erfahrungen

Szenarien bei der entory AG

- Workflow Engine (2001)
  - Portierung von Source Code:
    EJB 1.0 >>> EJB 1.1, JDK 1.2 >>> JDK 1.3, ...
  - Funktionalitätserweiterung
  - Refactoring, Überarbeitung des Designs
- BondTr@der (2001-2002)
  - Komponentenbasiertes Online-Handelssystem für Bonds
  - JUnit für Tests von Einzelkomponenten
  - JUnit/HttpUnit für UseCase-Tests von Teilen des Gesamtsystems

JUnit: Erfahrungen

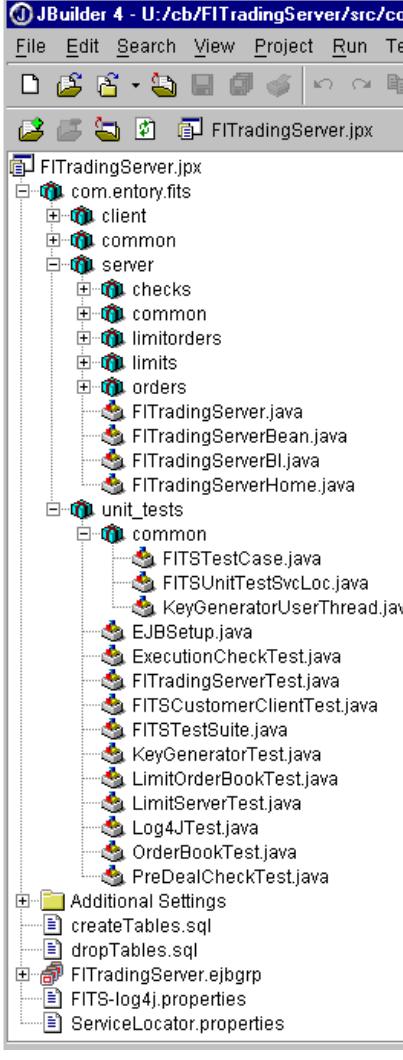
- **Vorgehensweise: (Workflow Engine)**
  - **Code Review**
  - **Refactoring**
  - **Schreiben von Use Case Tests**
    - Use Case „Create Workflow“
    - Use Case „Activate Callback“
    - Use Case „Process Event“
  - **Schreiben von Unit Tests**
    - Jeder Fehler führte zum Schreiben von Unit Tests
    - Jede neue Funktionalität führte zu einem oder mehreren neuen Unit Tests
    - Test Suite kann mittlerweile große Teile der Basisfunktionalität automatisch testen

JUnit: Weitere Empfehlungen

Packages für Unit-Tests

- **Generelle Strukturierung**
  - **`com.entory.<component>.unit_tests`**
    Enthält Unit Tests und mindestens eine Test Suite, in welcher alle zusammengefasst werden.
  - **`com.entory.<component>.unit_tests.common`**
    Enthält allgemeine Hilfsklassen für alle Unit Tests, z.B.
    - Allgemeine Oberklasse für alle Unit Tests, welche oft verwendete Funktionalitäten zur Verfügung stellt.
      Beispiel: Aufbau/Freigabe einer DB-Connection, Erzeugen von Remote-Referenzen von EJBs via ServiceLocator
    - Spezieller ServiceLocator, der mehr „Einblick“ in Server-Seite bietet als der normale Client-ServiceLocator.
- **Zusätzliche Strukturierungsmöglichkeiten unterhalb von `com.entory.<component>.unit_tests` ist oft sinnvoll, wenn die Zahl von Unit Tests größer wird**

JUnit: Weitere Empfehlungen



Beispiel: FI Trading Server (FITS)

- `com.entory.fits.unit_tests.common`
  - **FITSTestCase (extends TestCase)**
    Allgemeine (abstrakte) Oberklasse für alle Unit Tests von FITS.
  - **FITSUnitTestSvcLoc (extends FITSServiceLocator)**
    Spezieller ServiceLocator, der neben dem Zugriff auf die Session Facade (FITradingServer) Zugriff auf innere EJBs (OrderBook, LimitServer, ...) ermöglicht.
- `com.entory.fits.unit_tests`
  - **EJBSetup**
    Unit Test, welcher lediglich überprüft, ob Remote-Referenzen von EJBs der Serverseite via ServiceLocator erzeugt werden können.
  - **FITSTestSuite**
    Fasst alle Unit Tests zu einer Suite zusammen, um schnell einen Gesamttest machen zu können.
  - **Unit Tests**
    KeyGeneratorTest, OrderBookTest, LimitServerTest, FITradingServerTest, PreDealCheckTest, ...