



# FIESTA

**Modul für TI- COACH**

TESTSPEZIFIKATION

---

Bachelorpraktikum WS 05/06

Fachgebiet Knowledge Engineering

31. März 2006



Technische Universität  
Darmstadt

**TEAM WOLFSKIN**

Stephan Henkel

Ingo Reimund

Gregor Karzelek

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>2</b>
<b>2</b>	<b>Werkzeuge</b>	<b>3</b>
2.1	JUnit . . . . .	3
<b>3</b>	<b>Testmethoden</b>	<b>4</b>
3.1	Zweigabdeckung . . . . .	4
3.2	Use-Case basiert . . . . .	4
3.3	Black-Box-Test . . . . .	4
<b>4</b>	<b>Use-Case Tests</b>	<b>5</b>
4.1	Undo Test . . . . .	6
4.2	Vereinigen von Zuständen Test . . . . .	11
<b>5</b>	<b>Klassen Tests</b>	<b>17</b>
5.1	Automaten Test . . . . .	18
5.1.1	Hinzufügen von Zuständen . . . . .	20
5.1.2	Hinzufügen von Zustandsübergängen . . . . .	22
5.1.3	Akzeptieren von Zustandsübergängen mit Zeichen . . . . .	24
5.1.4	Test auf doppelte Zustandsübergänge . . . . .	26
5.1.5	Entfernen eines Zustandes . . . . .	28
5.1.6	Entfernen eines Zustandsübergang . . . . .	30
5.1.7	Iterator über die Startzustände . . . . .	32
5.1.8	Überprüfen auf vorhandenes Zeichen . . . . .	34
5.1.9	Hinzufügen von Zeichen . . . . .	36
5.1.10	Entfernen eines bereits vorhandenes Zeichens . . . . .	38
5.2	CommandList Test . . . . .	40
5.2.1	Hinzufügen von Befehlen . . . . .	42
5.2.2	Test ob ein Element folgt . . . . .	44
5.2.3	Test ob ein Element vorhergeht . . . . .	46
5.2.4	Rückgabe eines Befehls . . . . .	48
5.2.5	Einfrieren der Liste . . . . .	50
5.2.6	Einen Schritt vor . . . . .	52
5.2.7	Einen Schritt zurück . . . . .	54
5.2.8	Sprung an den Anfang . . . . .	56
5.2.9	Klonen der Liste . . . . .	58
<b>A</b>	<b>Glossar</b>	<b>60</b>

# 1 Einleitung

Bei der Entwicklung von FieStA geht es nicht nur um das Erstellen einer Software, die den Mindestanforderungen genügt, sondern auch um die Einhaltung gewisser Qualitätsstandards. Dieses Dokument beschäftigt sich mit den Methoden, die verwendet werden, um die gewünschten Standards zu erreichen und erläutert in den folgenden Kapiteln, wie genau diese Methoden angewandt werden.

Natürlich stellt sich an dieser Stelle die Frage, warum die Tests im einzelnen schon zu einer frühen Entwicklungsphase nötig sind. Eine denkbare Möglichkeit bestünde auch darin, den gesamten Code zu schreiben und erst nach der Fertigstellung des Programms an das eigentliche Debugging zu gehen. Diese Vorgehensweise hat jedoch eine Reihe von Nachteilen, so fällt das Debugging wesentlich einfacher, wenn schon während der Entwicklung einzelner Methoden überprüft wird, ob sie ihren Anforderungen gerecht werden. Bei der Fertigstellung des ersten Releases der Software ist es so möglich, garantierte Aussagen über die Funktionalität der Methoden zu treffen, so daß auftretende Probleme und Bugs leicht lokalisiert werden können. Zudem ermöglicht frühes Testen das Aufdecken von Fehlern, die bereits in der Entwicklung stattfinden. So können Fehler, die bereits bei der Planung der Architektur gemacht wurden, noch während der Programmierung korrigiert werden, ohne das große Teile vom Code dadurch betroffen sind. Zu guter Letzt kann durch das Testen mit verschiedenen Ausgangssituationen die Programmstabilität überprüft werden.

In diesem Dokument beziehen wir uns häufig mit dem Wort „Testen“ auf das Überprüfen der Korrektheit von Programmteilen. Hier wird dieses Wort nicht im Sinne von „Testen durch einen Anwender“ sondern eher im Sinne der Informatik gebraucht. In diesem Zusammenhang bedeutet dies: „In der Softwaretechnik bezeichnet Test meist ein Verfahren zum Aufspüren von Programmfehlern, siehe Softwaretest und Unit-Test. Dies muss unterschieden werden von einer formalen Verifizierung, also einem Beweis der Korrektheit eines Programmes.“<sup>1</sup> In unserem speziellen Falle ist das also die Überprüfung, ob sich Vor- und Nachbedingungen einer Methode mit deren theoretischer Ausarbeitung decken. Die Gruppe Wolfskin hat sich entschieden, alle hierfür nötigen Tests mithilfe von JUnit Tests zu realisieren. Diese ermöglichen auf übersichtliche Weise das Setzen verschiedener Rahmenbedingungen. Das Ergebnis der Tests gibt gleich Aufschluß darüber, ob das erwartete Ergebnis erhalten wurde, oder ob der Code noch Fehler enthält.

---

<sup>1</sup>Quelle: <http://de.wikipedia.org/wiki/Testen>

## 2 Werkzeuge

Dieses Kapitel beschreibt die von der Gruppe Wolfskin benutzten Werkzeuge zum Testen des Codes der im Laufe der Entwicklung von dem FieStA entstanden ist.

### 2.1 JUnit

Die in diesem Dokument durchgeführten Tests wurden mit einem Unit Framework überprüft. Hierbei handelt es sich um ein Testframework mit dessen Hilfe Modultest durchgeführt werden können. Im besonderen stellt es eine setUp Methode zur Verfügung in der Vorbedingungen definiert werden können und ein tearDown Methode für die Nachbedingungen. Desweiteren stellt das Framework Vergleichsmethoden zur Verfügung die in den Methoden der Testklasse aufgerufen werden können um einen einfachen Vergleich zu ermöglichen.

Das bei der Entwicklung von FieStA eingesetzte Unit Framework ist spezielle auf Java zugeschnitten und heißt deshalb JUnit. Mit diesem Framework wurden alle Klassen-Test durchgeführt und jeder Methode der Klassen im einzelnen getestet.

Bei dem Use-Case-Test stellt sich jedoch ein Problem, da ein Tool genutzt werden sollte, dass direkt auf der graphischen Oberfläche arbeitet. Da dies aber noch sehr umständlich sind, hat sich das Team Wolfskin dafür entschieden die Use-Case Tests ebenfalls mit JUnit zu testen. Da JUnit nicht die graphische Oberfläche testen kann, werden die Use-Case Test nicht direkt auf der Oberfläche ausgeführt, sondern testen den Code der hinter der Oberfläche liegt.

## 3 Testmethoden

In diesem Kapitel werden die benutzten Testmethoden beschrieben und erläutert.

### 3.1 Zweigabdeckung

Bei der Zweigabdeckung wird jede Zweig einer Bedienung mindestens einmal durchlaufen und somit der darin enthalten Code ausgeführt. Mit diesem Test wird sichergestellt, dass wenn ein Zweig aufgerufen wird, dieser auch so arbeitet wie es vorgesehen ist. Mit dieser Testmethode werden die Klassentest überprüft.

### 3.2 Use-Case basiert

Bei einem Use-Case basierten Test handelt es sich um einen Test, bei dem nicht wie bei der Zweigabdeckung jeder Bedingung geprüft wird, sondern lediglich die in dem Use-Case definierten Vorbedingungen in den Test eingegeben werden und im Anschluss geschaut wird, ob unter den gegebenen Vorbedingungen auch die geforderten Nachbedingungen auftreten. Diese Testmethode wird in den Use-Case Test benutzt.

### 3.3 Black-Box-Test

Der Black-Box-Test überprüft ähnlich wie die Zweigabdeckung Methoden, jedoch kennt der Black-Box-Test nur die Aufgabe der Methode und nicht deren Implementierung und ist somit unabhängig von dieser und überprüft lediglich ob die Methode das zurück liefert was erwartet wird.

Der Black-Box-Test wird bei einigen Klassen-Test's hinzugefügt um besonders wichtige Methoden unabhängig von der Implementierung auf Randbedingungen zu testen. Die Hauptbenutzung des Black-Box-Test erfolgt jedoch im Rahmen der Use-Case basierten Tests, da diese ohne Kenntnis der Implementierung durchgeführt werden.

## 4 Use-Case Tests

Dieses Kapitel beschreibt die Tests der verschiedenen Use-Cases, die mit Hilfe von JUnit durchgeführt werden. Da mit der Hilfe von JUnit nicht die graphische Oberfläche selbst getestet werden kann, setzen die Use-Case basierten Tests direkt unter der graphischen Oberfläche an und überprüfen von dort an den Ablauf der Use-Cases.

Jeder Use-Case basierte Test wird als Black-Box-Test durchgeführt und richtet sich nur nach den Vorbedingungen die laut Pflichtenheft gegeben sein müssen um die Aktion auszuführen. Des weiteren wurden eben jene Bedingungen dahingehend geändert, sodaß die Vorbedingungen nicht mehr erfüllt sind. Auf diese Weise wird festgestellt, ob das Programm auch auf alle nötigen Bedingungen abprüft. Die UseCase-Tests verifizieren also nicht nur, ob die Aktion das richtige Ergebnis liefert, sondern gehen zudem der Frage nach, ob sie falsche Ergebnisse oder zu Programmabstürzen führt, wenn nicht alle Vorbedingungen erfüllt sind.

In Absprache mit Herrn Grieser als Auftraggeber wurden für diese Tests zwei der im Pflichtenheft angeführten Aktionen entschieden. Zum einen Überprüfen wir im folgenden Abschnitt das Erstellen eines Zustandes. Der zweite UseCase, der überprüft wird ist die Undo-Funktion. Hierbei handelt es sich um eine komplexere Funktion von FieStA , die jedoch durch die Verwendung des Command-Patterns optimal reduziert wurde.

Damit durchgeführte Testläufe und aufgetretene Fehler eindeutig zugeordnet werden können, bekommt jede Use-Case basierte Test seine eigene ID, die sich aus den Buchstaben "UC" für "Use Case" und einer dreistelligen Zahl zusammen setzt. Diesem wird dann eine ID für den Testlauf und eine ID für eventuell gefundene Fehler angehängt. Damit ist jeder Testlauf und jeder Fehler genau einer Klasse und einer speziellen Methode zugeordnet und ermöglicht ein einfaches zurückverfolgen.

## 4.1 Undo Test

TEST ID:	ERSTELLER:	DATUM:
UC001	Ingo Reimund	10.03.2006

---

---

### **Beschreibung:**

Bei diesem Test wird die Undo Funktion des Editor getestet. Als Vorbedingung des Use-Cases ist ein geöffnetes Editor und mindestens eine ausgeführte Aktion gegeben. Als Nachbedingung muss die letzte Aktion rückgängig gemacht werden.

### **Traceability:**

Der Undo Test wird in drei Teile aufgebrochen. Als erstes wird ein Editor erzeugt dem dann drei Zustände und zwei Zustandsübergängen hinzugefügt werden. Anschließend wird ein Undo auf dem Automaten ausgeführt was auch von den Undo Button oder dem Menu-Eintrag erzeugt und ausgeführt werden würde. Anschließend wird überprüft ob das Undo im Protokoll enthalten ist und ob der Automat die als letztes hinzugefügt Kante entfernt hat.

Der zweite Test wird auf einem leeren Automaten ausgeführt und hat keinerlei Änderungen zur Folge, auch wird das Undo nicht im Protokoll aufgeführt.

Der letzte Test wird nun weggelassen werden, da es sich hierbei um einen Test bei einem geschlossenen Editor handelt. Da aber der Editor die Undo Funktion bereitstellt gibt es hier nichts zu testen.

**Spezifikation:**

```
public class UndoUseCaseTest extends TestCase {

    private Editor edit;

    private Application app;

    private DataManager dm;

    private Command undo = new UndoCommand();

    protected void setUp() throws Exception {
        super.setUp();

        //Application wird erstellt
        Application.main(new String [0]);

        // Ein Editor wird mit allem noetigen Erstellt
        app = Application.getInstance();
        dm = app.getDataFactory().createDataManager();
        edit = app.getEditorFactory().createEditor();
        ProtocolManager prot = new ProtocolManager();
        UndoManager uMan = new UndoManager();

        edit.setDataManager(dm);
        edit.setProtocollManager(prot);
        edit.setUndoManager(uMan);
        edit.addObserver(prot);
        edit.addObserver(uMan);
    }

    public void testActionPerformed() {
        //Es werden 3 Zustaende und 2 Zustandsuebergaenge erstellt
        Command exec;

        //Zustaende
        exec = new CreateStateCommand(dm, new Point(20,20), "A");
        exec.Execute();
        edit.commandExecuted(exec);

        exec = new CreateStateCommand(dm, new Point(60,20), "B");
        exec.Execute();
        edit.commandExecuted(exec);

        exec = new CreateStateCommand(dm, new Point(20,60), "C");
        exec.Execute();
        edit.commandExecuted(exec);

        //Zustandsuebergaenge
        exec = new CreateTransitionCommand(dm, dm.getState(0), dm.getState(1), "a");
        exec.Execute();
        edit.commandExecuted(exec);

        exec = new CreateTransitionCommand(dm, dm.getState(1), dm.getState(2), "b");
        exec.Execute();
        edit.commandExecuted(exec);

        // Test ob alle Transitions vorhanden sind
        Iterator it = edit.getDataManager().getAutomaton().getTransitionIterator();
        assertNotNull(it.next());
    }
}
```

```
        assertNotNull(it.next());
        assertFalse(it.hasNext());

        // Test das Undo noch nicht ausgeführt wurde
        assertFalse(edit.getProtocollManager().getList().get() instanceof UndoCommand);

        //undo wird ausgeführt
        edit.commandExecuted(undo);

        // ein Zustandsuebergang muss nun fehlen
        it = edit.getDataManager().getAutomaton().getTransitionIterator();
        assertNotNull(it.next());
        assertFalse(it.hasNext());

        // Undo muss im Protokoll vorhanden sein
        assertTrue(edit.getProtocollManager().getList().get() instanceof UndoCommand);
    }

    public void testnNoActionPerformed() {

        // keine Zustandsuebergaenge
        Iterator it = edit.getDataManager().getAutomaton().getTransitionIterator();
        assertFalse(it.hasNext());

        // keine Zustaeende
        it = edit.getDataManager().getAutomaton().getStateIterator();
        assertFalse(it.hasNext());

        // keien ausgefuhrten Befehle
        assertNull(edit.getProtocollManager().getList().get());

        //undo wird ausgeführt
        edit.commandExecuted(undo);

        // keine Zustandsuebergaenge
        it = edit.getDataManager().getAutomaton().getTransitionIterator();
        assertFalse(it.hasNext());

        // Keine ausgefuhrten Befehle vorhanden
        assertNull(edit.getProtocollManager().getList().get());

        // keine Zustaeende vorhanden
        it = edit.getDataManager().getAutomaton().getStateIterator();
        assertFalse(it.hasNext());
    }
}
```

## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
UC001-001	Ingo Reimund	0.8	11.03.2006

---

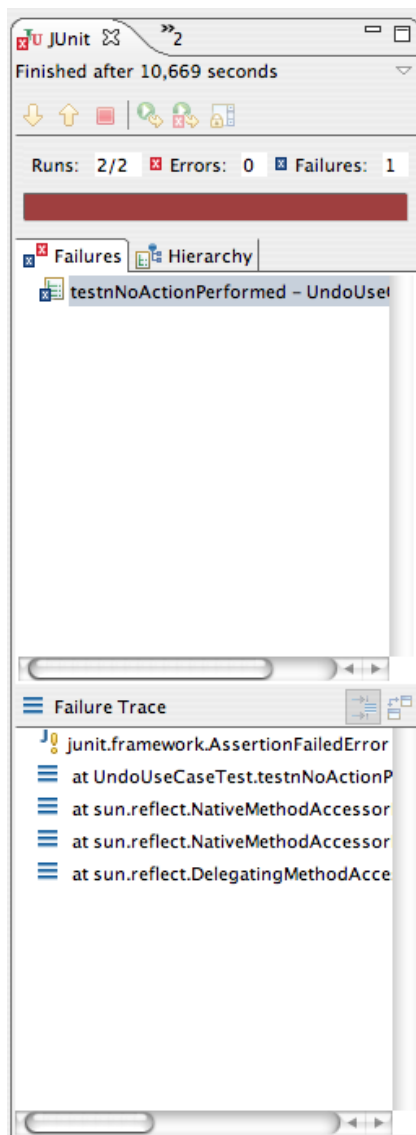
## Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001-001/01	Ingo Reimund	behoben	11.03.2006

---

### BESCHREIBUNG:

Undos werden zu dem Protokoll hinzugefügt, auch wenn noch keine Aktion zuvor ausgeführt wurde.

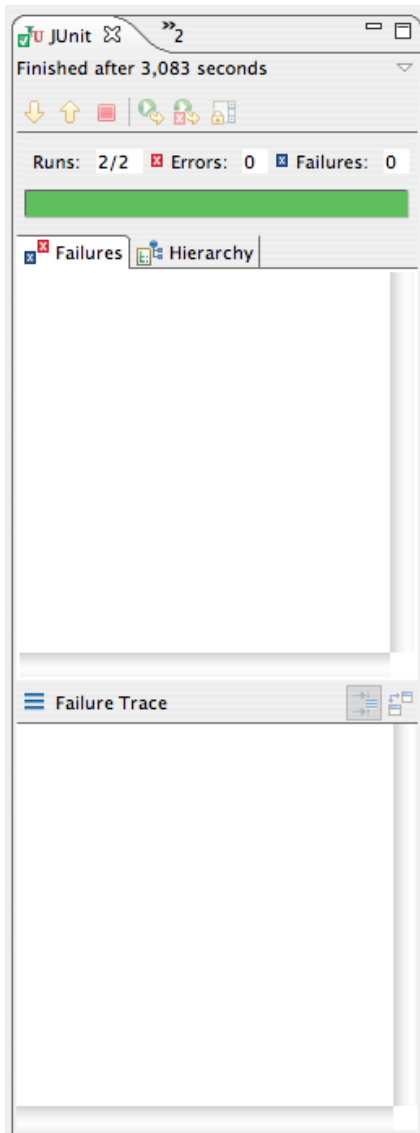


## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
UC001-002	Ingo Reimund	0.8	11.03.2006

---

*Es wurde kein Fehler gefunden.*



## 4.2 Vereinigen von Zuständen Test

TEST ID:

ERSTELLER:

DATUM:

UC002

Ingo Reimund

20.03.2006

---

---

### **Beschreibung:**

Testet das Vereinigen von zwei Knoten. Die von dem Pflichtenheft vorgegeben Vorbedingungen sind ein geöffneter Editor und zwei vorhanden Knoten. Als Nachbedingungen ist ein vereinigter Knoten gegeben.

### **Traceability:**

Dieser Test wird in drei Teile aufgebrochen. Zunächst wird einem leeren Editor 3 Zustände hinzugefügt. Anschließend wird ein Zustandsübergang von dem ersten Zustand zu dem zweiten und einer von dem ersten zu dem dritten Zustand erstellt. Schließlich folgt noch ein Zustandsübergang von dem zweiten zu dem dritten Zustand. Anschließend werden die ersten beiden Knoten vereinigt.

Der zweite Teil ist ein leerer Editor dem lediglich ein Zustands hinzugefügt wird. Bei dieser Ausgangssituation darf nun nichts verändert werden.

Nur ein leerer Editor kann nicht getestet werden, da der Befehl Vereinigen auf einen Zustand angewand werden muss.

**Spezifikation:**

```
public class MergeStateUseCaseTest extends TestCase {

    private Editor edit;

    private Application app;

    private DataManager dm;

    protected void setUp() throws Exception {
        super.setUp();

        // init the Application
        Application.main(new String[0]);

        // create all needed
        app = Application.getInstance();
        dm = app.getDataFactory().createDataManager();
        edit = app.getEditorFactory().createEditor();
        ProtocolManager prot = new ProtocolManager();
        UndoManager uMan = new UndoManager();

        // put it together
        edit.setDataManager(dm);
        edit.setProtocollManager(prot);
        edit.setUndoManager(uMan);
        edit.addObserver(prot);
        edit.addObserver(uMan);
    }

    public void test1State() {
        // create needed data-structure
        Command exec;
        Iterator iter;
        State before;
        State after;

        // 3 states
        exec = new CreateStateCommand(dm, new Point(20, 20), "A");
        exec.Execute();
        edit.commandExecuted(exec);

        iter = edit.getDataManager().getAutomaton().getStateIterator();
        before = (State) iter.next();

        MergeStatesSuperCommand merge = new MergeStatesSuperCommand(edit);
        merge.actionPerformed(null);

        MouseEvent clicked = new MouseEvent(edit.getGraphicalManager(), 0, 0,
            0, 0, 0, 0, false);
        clicked.setSource(dm.getState(0));

        merge.mouseClicked(clicked);
        merge.mouseClicked(clicked);

        merge.actionPerformed(null);

        iter = edit.getDataManager().getAutomaton().getStateIterator();
        assertTrue(iter.hasNext());
        after = (State) iter.next();
        assertFalse(iter.hasNext());
    }
}
```

```
    assertEquals(before.getId(), after.getId());
}

public void test3States() {
    // create needed data-structure
    Command exec;
    // 3 states
    exec = new CreateStateCommand(dm, new Point(20, 20), "A");
    exec.Execute();
    edit.commandExecuted(exec);

    exec = new CreateStateCommand(dm, new Point(60, 20), "B");
    exec.Execute();
    edit.commandExecuted(exec);

    exec = new CreateStateCommand(dm, new Point(20, 60), "C");
    exec.Execute();
    edit.commandExecuted(exec);

    State before1, before2, third, after;

    before1 = dm.getState(2);
    before2 = dm.getState(1);
    third = dm.getState(3);

    // 3 transitions
    exec = new CreateTransitionCommand(dm, before1, before2, "a");
    exec.Execute();
    edit.commandExecuted(exec);

    exec = new CreateTransitionCommand(dm, before2, third, "a");
    exec.Execute();
    edit.commandExecuted(exec);

    exec = new CreateTransitionCommand(dm, before1, third, "b");
    exec.Execute();
    edit.commandExecuted(exec);

    MergeStatesSuperCommand merge = new MergeStatesSuperCommand(edit);
    merge.actionPerformed(null);

    MouseEvent clicked = new MouseEvent(edit.getGraphicalManager(), 0, 0,
        0, 0, 0, 0, false);

    clicked.setSource(before1);
    merge.mouseClicked(clicked);

    clicked.setSource(before2);
    merge.mouseClicked(clicked);

    merge.actionPerformed(null);

    after = dm.getState(4);

    boolean found = false;
    Transition trans = null;
    String alpha = "";
    boolean incomingCorrect = true;
    boolean outgoingCorrect = true;

    for (int i = 0; i < before1.incomingTransitions().length
        && incomingCorrect; i++) {

        found = false;
```

```
trans = before1.incomingTransitions()[i];
alpha = trans.getAlphabet();

for (int j = 0; j < after.incomingTransitions().length && !found; j++) {

    String[] raw = alpha.split((String) Application.getInstance()
        .getPreferences().get("alpha_seperator"));
    for (int k = 0; k < raw.length; k++) {
        if (after.incomingTransitions()[i]
            .containsCharacter(raw[k])) {
            found = true;
            break;
        }
    }
}

incomingCorrect = found;
}

for (int i = 0; i < before1.outgoingTransitions().length
    && outgoingCorrect; i++) {

    found = false;
    trans = before1.outgoingTransitions()[i];
    alpha = trans.getAlphabet();

    for (int j = 0; j < after.outgoingTransitions().length && !found; j++) {

        String[] raw = alpha.split((String) Application.getInstance()
            .getPreferences().get("alpha_seperator"));
        for (int k = 0; k < raw.length; k++) {
            if (after.outgoingTransitions()[i]
                .containsCharacter(raw[k])) {
                found = true;
                break;
            }
        }
    }

    outgoingCorrect = found;
}

assertTrue(incomingCorrect);
assertTrue(outgoingCorrect);

assertEquals(after.getName(), before1.getName() + "/"
    + before2.getName());
}
}
```

## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
UC002-001	Ingo Reimund	0.9	20.03.2006

---

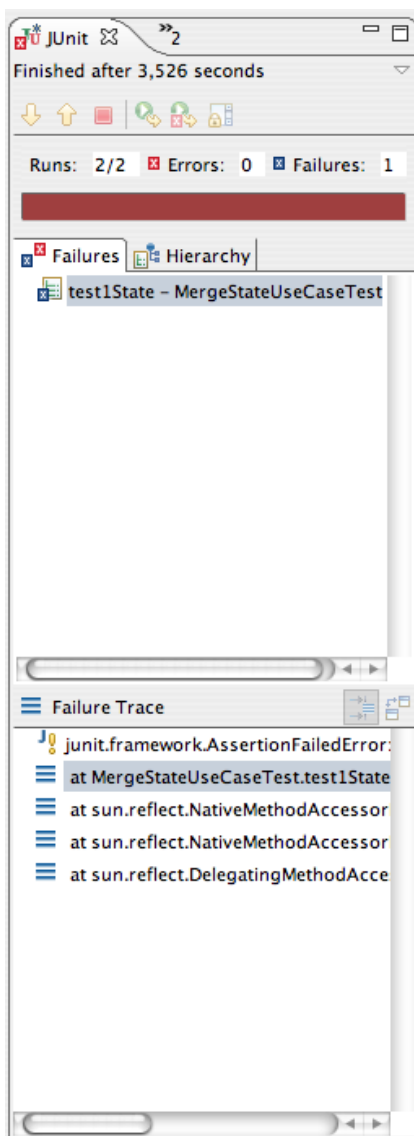
### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002-001/01	Ingo Reimund	behoben	20.03.2006

---

#### BESCHREIBUNG:

Vereinigen eines Zustandes mit sich selbst.

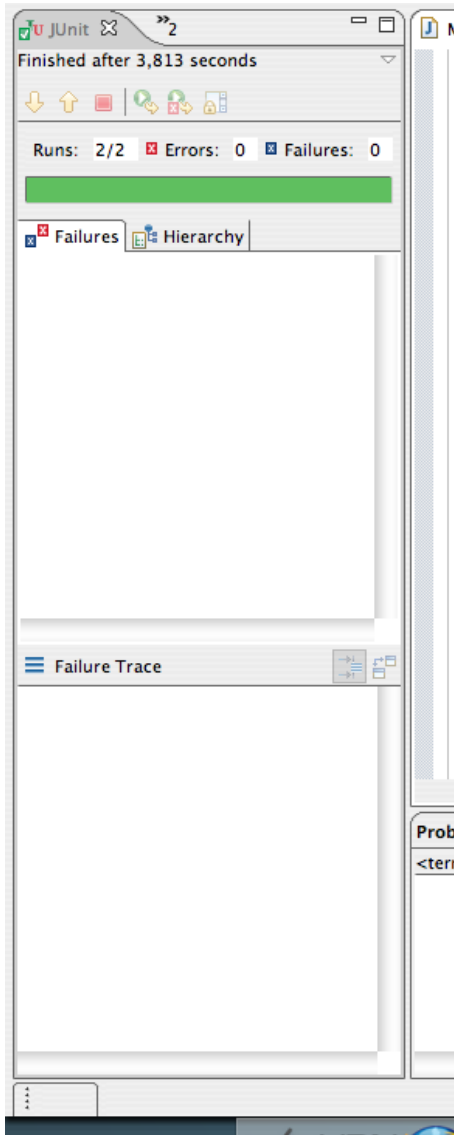


## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
UC002-002	Ingo Reimund	0.9	21.03.2006

---

*Es wurde kein Fehler gefunden.*



## 5 Klassen Tests

Dieses Kapitel beschreibt die Tests von Klassen, die ebenfalls wie die Use-Case basierten Tests mit JUnit durchgeführt werden. Der Unterschied zu den Use-Case basierten Tests liegt darin, dass nicht das Zusammenspiel von Klassen für eine bestimmte Benutzerinteraktion getestet wird, sondern die Methoden die eine Klasse zur Verfügung stellt. Dabei wird jede Methode einer Zweigabdeckung unterworfen um so Fehler in den verschiedenen Bedingungen einer Methode aufzudecken.

Als Anschauliches Beispiel wurde hier der JUnit Test der Automaton-Klasse gewählt. Es sei an dieser Stelle jedoch erneut darauf hingewiesen, daß fast alle Klassen in einem Test-Driven-Development Verfahren erstellt werden. Das bedeutet, daß der Klassentest nicht speziell für dieses Dokument erstellt wurde, sondern daß für fast alle Klassen dieses Projektes ähnlich umfangreiche Tests aufgestellt werden, noch bevor der eigentliche Code entsteht. In diesem Dokument wird der Test der Automaton-Klasse aufgeführt, da die Objekte des Typs Automaton alle Essentiellen Daten halten, die benötigt werden um den aktuell geöffneten Automaten darzustellen. Diese Klasse ist demnach eines der wichtigsten Elemente aus der Datenstruktur. Alle Manipulationen, die der Benutzer im Editor durchführt betreffen direkt die Daten, die im Automaton gespeichert werden, womit ist das Automaton-Objekt also das ist, was der Benutzer über die UI angezeigt bekommt.

Jeder Klassen Test wird in die einzelnen Methodentests aufgeteilt und diese werden einzeln beschrieben. Da JUnit bei jedem Test eine setUp Methode aufruft um eine gleiche Ausgangssituation für diese zu schaffen, wird die setUp Methode bei der Spezifikation der Klasse näher erklärt.

Damit durchgeführte Testläufe und aufgetretene Fehler eindeutig zugeordnet werden können, bekommt jeder Klassen Test seine eigene ID, die sich aus den Buchstaben "CT" für "Class Test" und einer dreistelligen Zahl zusammen setzt. Diesem wird dann eine ID für den speziellen Methodentest angehängt, sowie eine ID für den Testlauf und eine ID für eventuell gefundene Fehler. Damit ist jeder Testlauf und jeder Fehler genau einer Klasse und einer speziellen Methode zugeordnet und ermöglicht ein einfaches zurückverfolgen.

## 5.1 Automaten Test

TEST ID:	ERSTELLER:	DATUM:
CT001	Ingo Reimund	10.02.2006

---

### Beschreibung:

In diesem Test werden die Methoden der Klasse Automaten getestet. Der Schwerpunkt liegt dabei auf einer 100% Zweigabdeckung um die Funktionsweise der Methoden dieser Klasse zu garantieren, da diese bei der Konstruktion des Automaten einen wichtigen Teil der Arbeit übernimmt.

### Traceability:

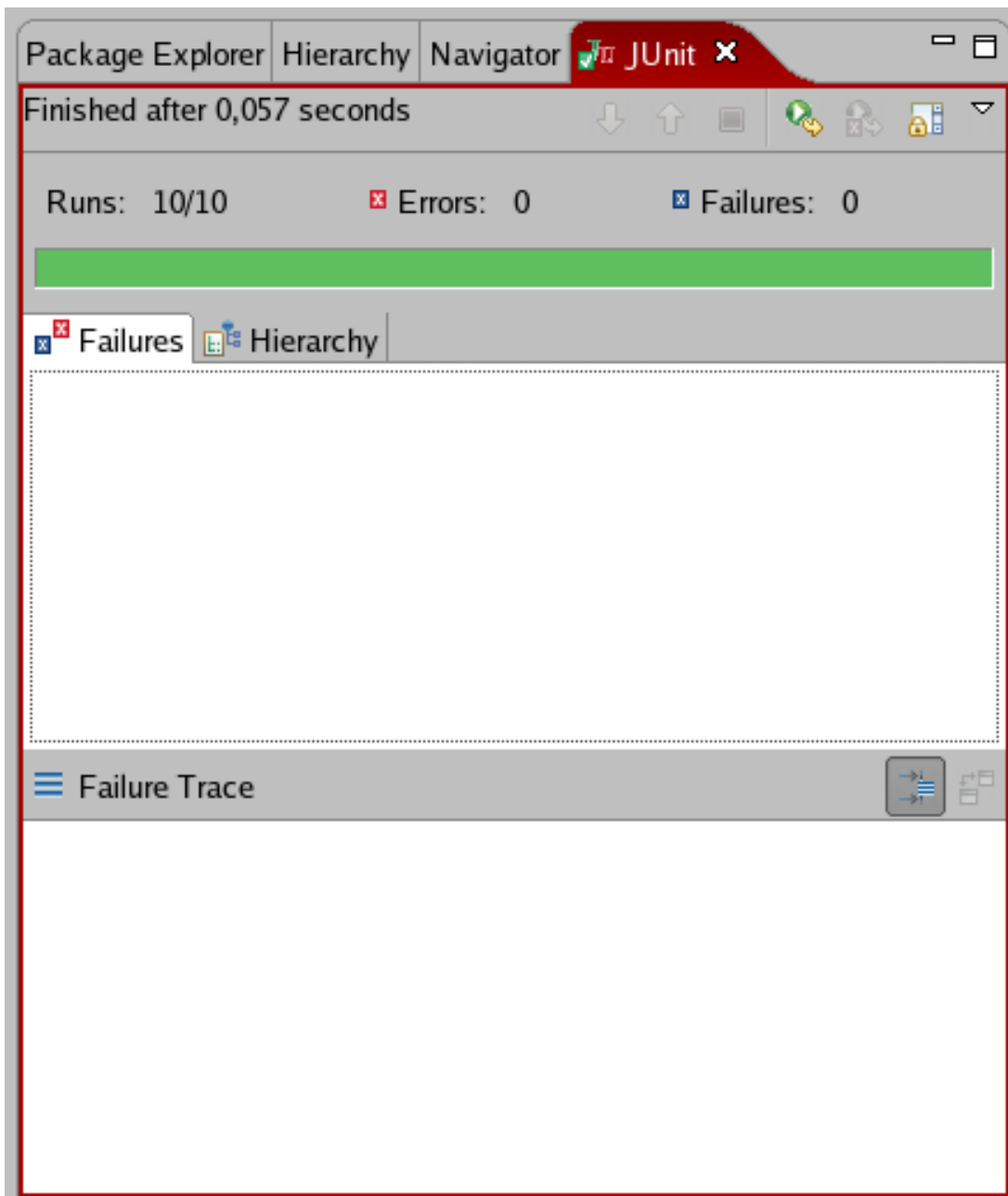
Bevor eine Methode durch JUnit getestet wird, erfolgt zunächst ein Aufruf der setUp Methode um ein einheitliche Ausgangssituation für alle Tests zu schaffen. Bei dem Automaten Test wird zunächst ein Automat erzeugt und diesem dann zwei in der ganzen Testklasse verfügbaren Zustände hinzugefügt sowie ein Zustandsübergang der im ersten Zustand startet und im zweiten endet.

### Spezifikation:

```
// Automat der getestet werden soll.
private Automaton automaton;
// Erster Zustand fuer das Testen.
private State state0 = new State();
// Zweiter Zustand fuer das Testen.
private State state1 = new State();
// Zustandsuebergang der bei Zustand state0 startet und bei
// state1 endet
private Transition transition0 = new Transition(this.state0, this.state1);

/**
 * Erstellt bei jedem aufgerufen Test in der AutomatenTest Klasse einen
 * neuen Automaten und fuegt diesem 2 Zustande hinzu, sowie einen
 * Zustandsuebergang und das Alphabet "a, b, c".
 */
public void setUp() throws Exception {
    this.automaton = new Automaton();
    this.automaton.add(this.state0);
    this.automaton.add(this.state1);
    this.automaton.add(this.transition0);
    this.automaton.setAlphabet("a,_b,_c");
}
```

**Letzter Testlauf:**



Screenshot des letzten Testlauf über den ganzen Automaten.

### 5.1.1 Hinzufügen von Zuständen

TEST ID:	ERSTELLER:	DATUM:
CT001/01	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Testet das Hinzufügen von Zuständen zu dem Automaten.

#### Traceability:

Bei dem Hinzufügen von Zuständen muss darauf geachtet werden, dass keine zwei Zustände die selbe ID besitzen, da sonst nur auf den zuletzt hinzugefügte Zustand zugegriffen werden kann.

#### Spezifikation:

```
public void testAddState() {
    // Fuegt dem Automaten einen Zustand hinzu, der die
    // gleiche ID hat wie ein bereits existierender Zustand.
    // Erwartet wird, dass der neue Zustand nicht hinzugefuegt
    // wird.
    State state = new State(this.state0.getId());
    this.automaton.add(state);
    assertNotSame(state, this.automaton.getState(state.getId()));
    assertEquals(this.state0, this.automaton.getState(state.getId()));

    // Fuegt dem Automaten einen neuen Zustand mit einer neuen
    // ID hinzu
    // Erwartet wird, dass der neue Zustand hinzugefuegt wird.
    state = new State();
    this.automaton.add(state);
    assertEquals(state, this.automaton.getState(state.getId()));
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/01-001	Ingo Reimund	0.7	12.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/01-002	Ingo Reimund	0.7	13.02.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/01-002/01	Ingo Reimund	behoben	16.02.2006

---

## BESCHREIBUNG:

Nach dem Hinzufügen eines Zustandes, hat der betroffenen Zustand noch Zustandsübergänge die nicht im Automaten enthalten sind.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/01-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/01-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

## 5.1.2 Hinzufügen von Zustandsübergängen

TEST ID:  
CT001/02

ERSTELLER:  
Ingo Reimund

DATUM:  
10.02.2006

---

### Beschreibung:

Testet das Hinzufügen von Zustandsübergängen zu dem Automaten.

### Traceability:

Bei dem Hinzufügen von Zustandsübergängen muss darauf geachtet werden, dass keine zwei Zustandsübergänge mit der selben ID hinzugefügt werden dürfen. Ebenso darf nur ein Zustandsübergang hinzugefügt werden, dessen Start- und Endzustand im Automaten vorhanden sind, das Alphabet des Zustandsübergangs vom Automaten akzeptiert wird und es keinen anderen Zustandsübergang mit dem selben Start- und dem selben Endzustand gibt. Die beiden letzten Bedingungen werden in gesonderten Methoden abgearbeitet und ebenfalls gesondert getestet.

### Spezifikation:

```
public void testAddTransition() {
    // Fuegt dem Automaten einen Zustandsuebergang hinzu, der die
    // gleiche ID hat wie ein bereits existierender Zustandsuebergang.
    // Erwartet wird, dass der neue Zustandsuebergang nicht
    // hinzugefuegt wird.
    Transition transition = new Transition(this.transition0.getId(),
        this.state0, this.state0);
    this.automaton.add(transition);
    assertNotSame(transition, this.automaton.getTransition(transition
        .getId()));
    assertEquals(this.transition0, this.automaton.getTransition(transition
        .getId()));

    // Fuegt einen Zustandsuebergang hinzu, bei dem ein Zustand
    // nicht in dem Automaten existiert.
    // Erwartet wird, dass der Zustandsuebergang nicht
    // hinzugefuegt wird.
    transition = new Transition(state0, new State());
    this.automaton.add(transition);
    assertNull(this.automaton.getTransition(transition.getId()));

    // Fuegt einen Zustandsuebergang hinzu, mit dem gleichen Start- und
    // Endzustand von einem bereits hinzugefuegtem Zustandsuebergang.
    // Erwartet wird, dass der Zustandsuebergang nicht hinzugefuegt
    // wird.
    transition = new Transition(this.state0, this.state1);
    this.automaton.add(transition);
    assertNull(this.automaton.getTransition(transition.getId()));

    // Fuegt einen Zustandsuebergang hinzu
    // Erwartet wird, dass der Zustandsuebergang hinzugefuegt wird.
    transition = new Transition(this.state1, this.state1);
    this.automaton.add(transition);
    assertEquals(transition, this.automaton.getTransition(transition
        .getId()));
}
```

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/02-001	Ingo Reimund	0.7	12.02.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/02-001/01	Ingo Reimund	abgelehnt	12.02.2006

---

## BESCHREIBUNG:

Zustandsübergänge werden mehrfach hinzugefügt und es werden nicht die Zeichen der Zustandsübergänge beachtet. Der Fehler wurde abgelehnt da die Struktur nicht verständlich war und die Überprüfungen wurden in gesonderte Methoden ausgelagert

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/02-002	Ingo Reimund	0.7	13.02.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/02-002/01	Ingo Reimund	behoben	14.02.2006

---

## BESCHREIBUNG:

Zustandsübergänge dessen Start oder EndZustands nicht im Automaten liegen werden hinzugefügt.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/02-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/02-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.3 Akzeptieren von Zustandsübergängen mit Zeichen

TEST ID:  
CT001/03

ERSTELLER:  
Ingo Reimund

DATUM:  
10.02.2006

---

#### Beschreibung:

Überprüft ob die Zeichen des Zustandsübergangs auch von dem Automaten akzeptiert wird.

#### Traceability:

Diese Testmethode übernimmt eine kleine Teilaufgabe bei dem Hinzufügen von Zustandsübergängen. Sie überprüft ob der Automat auch die Zeichen des Zustandsübergang akzeptiert. Wenn der Automat jeden Zustandsübergang akzeptieren soll, dann werden die Zeichen des Zustandsübergangs dem Alphabet des Automaten hinzugefügt.

#### Spezifikation:

```
public void testAcceptTransitionAlphabet() {
    // akzeptiert alle Zustandsuebergaenge
    this.automaton.acceptAllTransitions(true);

    // fuegt einen neuen Zustandsuebergang hinzu, der ein neues
    // Zeichen, das noch nicht im Alphabet des Automaten vorhanden ist.
    // Erwartet wird, dass das Zeichen dem Automaten hinzugefuegt
    // wird.
    // und der Zustandsuebergang ebenfalls.
    Transition transition = new Transition(state0, state0);
    transition.addCharacterToAlphabet("d");
    this.automaton.add(transition);
    assertEquals(0, this.automaton.getAlphabet().compareTo("a,_b,_c,_d"));
    assertNotNull(this.automaton.getTransition(transition.getId()));

    // akzeptiert nur Zustandsuebergaenge mit bereits bekannte Zeichen
    this.automaton.acceptAllTransitions(false);

    // Fuegt einen neuen Zustandsuebergang mit dem Zeichen "g" das dem
    // dem Automaten nicht bekannt ist.
    // Erwartet wird, dass der Zustandsuebergang nicht hinzugefuegt
    // wird.
    // sowie das Zeichen "g" nicht dem Automaten hinzugefuegt wird.
    transition = new Transition(state1, state1);
    transition.addCharacterToAlphabet("g");
    this.automaton.add(transition);
    assertEquals(0, this.automaton.getAlphabet().compareTo("a,_b,_c,_d"));
    assertNull(this.automaton.getTransition(transition.getId()));

    // Fuegt einen Zustandsuebergang ohne Zeichen hinzu.
    // Erwartet wird, dass der Zustandsuebergang nicht hinzugefuegt
    // wird.
    transition = new Transition(state1, state1);
    this.automaton.add(transition);
    assertNull(this.automaton.getTransition(transition.getId()));

    // Fuegt einen Zustandsuebergang hinzu, mit einem Zeichen das der
    // Automat bereits besitzt.
    // Erwartet wird, dass das der Zustandsuebergang hinzugefuegt
    // wird.
}
```

```

transition = new Transition(state1, state1);
transition.addCharacterToAlphabet("d");
this.automaton.add(transition);
assertEquals(0, this.automaton.getAlphabet().compareTo("a,_b,_c,_d"));
assertNotNull(this.automaton.getTransition(transition.getId()));
}

```

## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/03-001	Ingo Reimund	0.7	13.02.2006

### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/03-001/01	Ingo Reimund	behoben	13.02.2006

#### BESCHREIBUNG:

Der Automat akzeptiert alle Zustandsübergänge und ihm wird ein Zustandsübergang mit dem Zeichen "g" hinzugefügt, jedoch entählt der Automat im Anschluss nur den Zustandsübergang und nicht das Zeichen des neuen Zustandsübergangs.

### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/03-001/02	Ingo Reimund	behoben	14.02.2006

#### BESCHREIBUNG:

Der Automat akzeptiert nur Zustandsübergänge mit bestimmten Zeichen und ihm wird ein Zustandsübergang ohne Zeichen hinzugefügt, jedoch entählt der Automat im Anschluss den Zustandsübergang, obwohl diese nicht hinzugefügt werden darf.

## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/03-002	Ingo Reimund	0.8	28.02.2006

*Es wurde kein Fehler gefunden.*

## Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/03-003	Ingo Reimund	0.9	15.03.2006

*Es wurde kein Fehler gefunden.*

### 5.1.4 Test auf doppelte Zustandsübergänge

TEST ID:	ERSTELLER:	DATUM:
CT001/04	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Test auf einen zweiten Zustandsübergang mit dem selben Start und Endzustand.

#### Traceability:

Diese Testmethode übernimmt eine kleine Teilaufgabe bei dem Hinzufügen von Zustandsübergängen. Sie testet auf einen bereits vorhandenen Zustandsübergang der den selben Start- und den selben Endzustand besitzt wie der Zustandsübergang der neu hinzugefügt werden soll.

#### Spezifikation:

```
public void testDoubleTransition() {
    // Fügt einen Zustandsuebergang hinzu, der den state0 als
    // Startzustand und state1 als Endzustand hat.
    // Erwartet wird, dass der Zustandsuebergang nicht hinzugefügt
    // wird.
    Transition transition = new Transition(this.state0, this.state1);
    this.automaton.add(transition);
    assertNull(this.automaton.getTransition(transition.getId()));

    // Fügt einen Zustandsuebergang hinzu, der den state0 als
    // Startzustand besitzt aber einen anderen Endzustand.
    // Erwartet wird, dass der Zustandsuebergang hinzugefügt wird.
    transition = new Transition(state0, state0);
    this.automaton.add(transition);
    assertEquals(transition, this.automaton.getTransition(transition
        .getId()));

    // Fügt einen Zustandsuebergang hinzu, der den state1 als
    // Startzustand und state0 als Endzustand hat.
    // Erwartet wird, dass der Zustandsuebergang hinzugefügt wird.
    transition = new Transition(state1, state0);
    this.automaton.add(transition);
    assertEquals(transition, this.automaton.getTransition(transition
        .getId()));
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/04-001	Ingo Reimund	0.7	13.02.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/04-001/01	Ingo Reimund	behoben	13.02.2006

---

#### BESCHREIBUNG:

Es gibt mehrere Zustandsübergänge die den selben Start- und Endzustand besitzen.

### **Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/04-002	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

### **Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/04-003	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.5 Entfernen eines Zustandes

TEST ID:  
CT001/05

ERSTELLER:  
Ingo Reimund

DATUM:  
10.02.2006

---

#### Beschreibung:

Testet das Entfernen eines Zustandes.

#### Traceability:

Diese Methode testet das Entfernen eines Zustandes eines bereits bestehenden Automaten. Hierbei ist besonders darauf zu achten, dass eventuell mit dem zu entfernenden Zustands verbundenen Zustandsübergänge ebenfalls entfernt werden.

#### Spezifikation:

```
public void testRemoveState () {  
  
    // Entfernt einen Zustand der keine Zustandsuebergaenge besitzt.  
    // Erwartet wird, dass der Zustand entfernt wird.  
    State state = new State ();  
    this.automaton.add(state);  
    assertNotNull(this.automaton.getState(state.getId()));  
    this.automaton.remove(state);  
    assertNull(this.automaton.getState(state.getId()));  
  
    // Entfernt den state0 und die neu hinzugefuegten  
    // Zustandsuebergang sowie transition0.  
    // Erwartet wird, dass der Zustands entfernt wird und die  
    // Zustandsuebergaenge ebenfalls.  
    Transition transition = new Transition(this.state0, this.state0);  
    this.automaton.add(transition);  
    this.automaton.remove(this.state0);  
    assertNull(this.automaton.getState(state0.getId()));  
    assertNull(this.automaton.getTransition(transition0.getId()));  
    assertNull(this.automaton.getTransition(transition.getId()));  
  
    // Entfernt einen Zustand der nicht im Automaten vorhanden ist.  
    // Erwartet wird keine Veraenderung.  
    state = new State ();  
    this.automaton.remove(state);  
    assertNull(this.automaton.getState(state.getId()));  
}
```

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/05-001	Ingo Reimund	0.7	12.02.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/05-001/01	Ingo Reimund	behoben	13.02.2006

---

## BESCHREIBUNG:

Aus dem Automaten wird ein Zustand gelöscht, jedoch bleiben die Zustandsübergänge dieses Automaten vorhanden.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/05-002	Ingo Reimund	0.7	13.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/05-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/05-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.6 Entfernen eines Zustandsübergang

TEST ID:	ERSTELLER:	DATUM:
CT001/06	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Testet das Entfernen eines Zustandsübergangs.

#### Traceability:

Entfernt einen Zustandsübergang aus einem bereits vorhanden Automaten.

#### Spezifikation:

```

public void testRemoveTransition() {
    // Entfernt einen Zustandsuebergang.
    // Erwartet wird, dass der Zustandsuebergang entfernt wird.
    assertNotNull(this.state0.outgoingTransitions()[0]);
    assertNotNull(this.state1.incomingTransitions()[0]);
    this.automaton.remove(this.transition0);
    assertNull(this.automaton.getTransition(this.transition0.getId()));
    assertEquals(0, this.state0.outgoingTransitions().length);
    assertEquals(0, this.state1.incomingTransitions().length);

    // Entfernt einen nicht vorhandenen Zustandsuebergang
    // Erwartet wird keine Reaktion.
    this.automaton.remove(new Transition(this.state0, this.state1));
    assertNull(this.automaton.getTransition(this.transition0.getId()));
}

```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/06-001	Ingo Reimund	0.7	12.02.2006

---

*Es wurde kein Fehler gefunden.*

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/06-002	Ingo Reimund	0.7	13.02.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/06-002/01	Ingo Reimund	behoben	13.02.2006

---

#### BESCHREIBUNG:

Aus dem Automaten wird ein Zustandsübergang gelöscht, jedoch zeichen Start- und Endzustands noch immer auf den gelöschten Zustandsübergang

### **Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/06-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

### **Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/06-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.7 Iterator über die Startzustände

TEST ID:	ERSTELLER:	DATUM:
CT001/07	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Testet das Erstellen eines Iterator über die Startzustände.

#### Traceability:

Diese Methode erstellt einen Iterator über alle Startzustände.

#### Spezifikation:

```
public void testStartStateIterator() {
    // keine Startzustaende vorhanden.
    assertFalse(this.automaton.getStartStateIterator().hasNext());

    // Nur state0 ist Startzustand.
    this.state0.setStartstate(true);
    Iterator it = this.automaton.getStartStateIterator();
    assertTrue(it.hasNext());
    assertEquals(this.state0, it.next());

    // Keine Zustaende vorhanden und darum keine Startzustaende.
    this.automaton.remove(this.state0);
    this.automaton.remove(this.state1);
    assertFalse(this.automaton.getStartStateIterator().hasNext());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/07-001	Ingo Reimund	0.7	12.02.2006

---

*Es wurde kein Fehler gefunden.*

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/07-002	Ingo Reimund	0.7	13.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/07-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/07-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.8 Überprüfen auf vorhandenes Zeichen

TEST ID:	ERSTELLER:	DATUM:
CT001/08	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Überprüft auf bereits vorhandene Zeichen.

#### Traceability:

Diese Methode testet ob die Zeichen des Alphabet auch richtig erkannt werden. Besonders kritisch ist dabei das leere Zeichen.

#### Spezifikation:

```
public void testContainsCharacter() {
    // Zeichen "a" im Alphabet des Automaten.
    assertTrue(this.automaton.containsCharacter("a"));

    // Die Zeichen sind nicht im Alphabet
    assertFalse(this.automaton.containsCharacter(""));
    assertFalse(this.automaton.containsCharacter("d"));

    // Setzt ein leeres Alphabet.
    this.automaton.setAlphabet("");
    assertFalse(this.automaton.containsCharacter(""));
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/08-001	Ingo Reimund	0.7	12.02.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/08-001/01	Ingo Reimund	behoben	13.02.2006

---

#### BESCHREIBUNG:

Der Leerstring wird als Zeichen erkannt.

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/08-002	Ingo Reimund	0.7	13.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/08-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/08-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.9 Hinzufügen von Zeichen

TEST ID:	ERSTELLER:	DATUM:
CT001/09	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Testet das Hinzufügen von Zeichen.

#### Traceability:

Diese Methode testet ob neue Zeichen hinzugefügt werden und ob sie auch nur einmal vorhanden sind.

#### Spezifikation:

```
public void testAddCharacterToAlphabet() {

    // Fuegt ein neues Zeichen dem Alphabet des Automaten hinzu.
    this.automaton.addCharacterToAlphabet("d");
    assertEquals(0, this.automaton.getAlphabet().compareTo("a,_b,_c,_d"));
    assertTrue(this.automaton.containsCharacter("d"));

    // Fuegt dem Lehren Automatenalphabet ein neues Zeichen hinzu.
    this.automaton.setAlphabet("");
    this.automaton.addCharacterToAlphabet("a");
    assertEquals(0, this.automaton.getAlphabet().compareTo("a"));
    assertTrue(this.automaton.containsCharacter("a"));

    // Fuegt ein bereits vorhandenes Zeichen hinzu
    this.automaton.addCharacterToAlphabet("a");
    assertTrue(this.automaton.containsCharacter("a"));
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/09-001	Ingo Reimund	0.7	12.02.2006

---

*Es wurde kein Fehler gefunden.*

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/09-002	Ingo Reimund	0.7	13.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/09-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/09-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.1.10 Entfernen eines bereits vorhandenes Zeichens

TEST ID:	ERSTELLER:	DATUM:
CT001/10	Ingo Reimund	10.02.2006

---

#### Beschreibung:

Überprüft das Entfernen von Zeichen.

#### Traceability:

Diese Methode testet ob die Zeichen des Alphabet auch richtig entfernt werden. Besonders getestet wird das Entfernen von Zeichen aus einem leeren Alphabet und das Entfernen von nicht vorhandenen Zeichen.

#### Spezifikation:

```
public void testRemoveCharacterFromAlphabet() {

    // Entfernt ein Zeichen aus dem Alphabet, das auch vorhanden ist.
    this.automaton.removeCharacterFromAlphabet("a");
    assertFalse(this.automaton.containsCharacter("a"));
    assertTrue(this.automaton.containsCharacter("b"));
    assertTrue(this.automaton.containsCharacter("c"));

    // Entfernt ein Zeichen aus einem leeren Alphabet
    this.automaton.setAlphabet("");
    this.automaton.removeCharacterFromAlphabet("a");
    assertFalse(this.automaton.containsCharacter("a"));
    assertEquals(0, this.automaton.getAlphabet().compareTo(""));

    // Entfernt ein nicht vorhandenes Zeichen aus dem Alphabet
    this.automaton.setAlphabet("a");
    this.automaton.removeCharacterFromAlphabet("h");
    assertFalse(this.automaton.containsCharacter("h"));
    assertEquals(0, this.automaton.getAlphabet().compareTo("a"));
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/10-001	Ingo Reimund	0.7	12.02.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT001/10-001/01	Ingo Reimund	behoben	13.02.2006

---

#### BESCHREIBUNG:

Das Zeichen war nach dem Entfernen immer noch vorhanden.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/10-002	Ingo Reimund	0.7	13.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/10-003	Ingo Reimund	0.8	28.02.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT001/10-004	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

## 5.2 CommandList Test

TEST ID:	ERSTELLER:	DATUM:
CT002	Ingo Reimund	10.03.2006

---

### Beschreibung:

In diesem Test werden die Methoden der Klasse CommandList getestet. Die Entscheidung dafür liegt in der Relevanz der CommandList die in dem UndoManager und auch in dem ProtocolManager benutzt werden um die Commands in der ausgeführten Reihenfolge zu halten und einen einfachen Zugriff darauf zu ermöglichen.

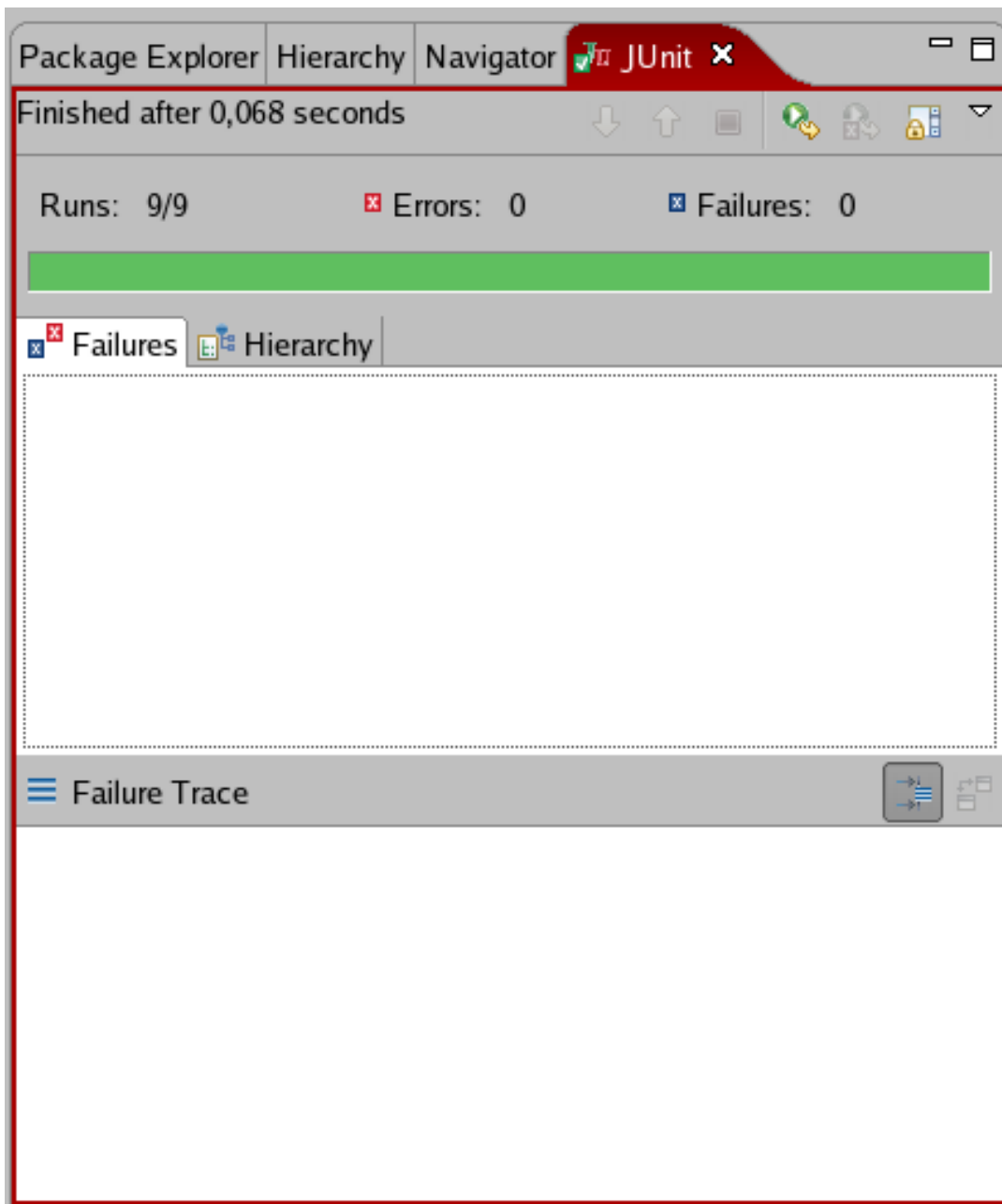
### Traceability:

Bevor eine Methode durch JUnit getestet wird, erfolgt zunächst ein Aufruf der setUp Methode um ein einheitliche Ausgangssituation für alle Tests zu schaffen. Für den Test der CommandList wird dabei eine neue Liste erzeugt und zunächst der command0 und anschließend der command1 hinzugefügt, so dass die Liste lediglich zwei Element besitzt.

### Spezifikation:

```
protected void setUp() throws Exception {
    this.list = new CommandList();
    this.list.add(this.command0);
    this.list.add(this.command1);
}
```

**Letzter Testlauf:**



Screenshot des letzten Testlauf über die CommandList.

### 5.2.1 Hinzufügen von Befehlen

TEST ID:	ERSTELLER:	DATUM:
CT002/01	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Testet das Hinzufügen von neuen Befehlen in die Liste.

#### Traceability:

Wird ein neuer Befehl der Liste hinzugefügt, so wird dieses an die aktuelle Position angehängt und Befehle die nach der aktuellen Position folgen werden überschrieben und fallen komplett weg. Es gibt beim Hinzufügen von Befehlen eine Überprüfung auf "null".

#### Spezifikation:

```

public void testAdd() {

    // Geht vor das erste Element und fuegt ein neues Element hinzu.
    // Das neue Element muss nun das erste und einzige Element sein.
    this.list.stepBackward();
    this.list.stepBackward();
    this.list.add(this.command1);
    assertEquals(this.command1, this.list.get());
    assertFalse(this.list.hasNext());

    // Testet ob eine leere Liste auch "null" zurueck gibt.
    this.list = new CommandList();
    assertNull(this.list.get());

    // Fuegt einen Befehl zu der Liste hinzu.
    this.list.add(this.command0);
    assertEquals(this.command0, this.list.get());

    // Fuegt ein "null" hinzu.
    this.list.add(null);
    assertEquals(this.command0, this.list.get());
}

```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/01-001	Ingo Reimund	0.8	11.03.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/01-002/01	Ingo Reimund	offen	15.03.2006

---

#### BESCHREIBUNG:

"null" wird zu der Liste hinzugefügt.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/01-002	Ingo Reimund	0.9	15.03.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/01-002/01	Ingo Reimund	behoben	15.03.2006

---

## BESCHREIBUNG:

“null“ wird zu der Liste hinzugefügt.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/01-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

## 5.2.2 Test ob ein Element folgt

TEST ID:	ERSTELLER:	DATUM:
CT002/02	Ingo Reimund	10.03.2006

---

### Beschreibung:

Testet auf nächste Element in der Liste

### Traceability:

Dieser Test testet lediglich ob die hasNext() Methode so arbeitet wie es z.B. von einem Iterator erwartet wird. Da es auch möglich ist, dass das erste Element rückgängig gemacht werden kann, so hält die Liste ein null Objekt vor dem ersten Element das den Anfang widerspiegelt.

### Spezifikation:

```

public void testHasNext() {

    // Geht von dem letzten Element Schritt fuer Schritt zum Anfang
    // und testet ob die Element auch immer erkannt werden.
    assertFalse(this.list.hasNext());
    this.list.stepBackward();
    assertTrue(this.list.hasNext());
    this.list.stepBackward();
    assertTrue(this.list.hasNext());

    // Geht or das Element das vor dem ersten Element liegt und
    // testet ob dieser Schritt gemacht wird.
    this.list.stepBackward();
    assertTrue(this.list.hasNext());

    // Testet bei einer leeren Liste.
    this.list = new CommandList();
    assertFalse(this.list.hasNext());
}

```

### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/02-001	Ingo Reimund	0.8	11.03.2006

---

### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/02-001/01	Ingo Reimund	offen	11.03.2006

---

#### BESCHREIBUNG:

Bei einer Liste mit Elementen wird bei dem null Objekt am Anfang der Liste ein "false" zurück gegeben, obwohl weiter Elemente vorhanden sind.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/02-002	Ingo Reimund	0.9	15.03.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/02-002/01	Ingo Reimund	behoben	15.03.2006

---

## BESCHREIBUNG:

Bei einer Liste mit Elementen wird bei dem null Objekt am Anfang der Liste ein "false" zurück gegeben, obwohl weitere Elemente vorhanden sind.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/02-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.3 Test ob ein Element vorhergeht

TEST ID:	ERSTELLER:	DATUM:
CT002/03	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Testet ob ein Element in der Liste vorhergeht.

#### Traceability:

Dieser Test überprüft ob die Methode `hasPrevious()` vorhergehende Befehle erkennt und keinen falsche Aussage liefert, da diese Methode für das Iterieren über die Befehlsliste wichtig ist.

#### Spezifikation:

```
public void testHasPrevious() {
    // Der Zeiger der Liste zeigt auf das zweite Element und erwartet
    // das ein Element davor liegt;
    assertTrue(this.list.hasPrevious());

    // Der Zeiger der Liste wird ein Element nach hinten verschoben und
    // zeigt somit auf das erste Element in der Liste und hat somit
    // keine Vorgaenger.
    this.list.stepBackward();
    assertFalse(this.list.hasPrevious());
    assertEquals(this.command0, this.list.get());

    // Die Liste wird verlassen und es duerfen sich auch hier keine
    // Element davor befinden.
    this.list.stepBackward();
    assertFalse(this.list.hasPrevious());
    assertNull(this.list.get());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/03-001	Ingo Reimund	0.8	11.03.2006

---

*Es wurde kein Fehler gefunden.*

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/03-002	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/03-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.4 Rückgabe eines Befehls

TEST ID:	ERSTELLER:	DATUM:
CT002/04	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Testet ob die get() Methode die richtigen Element zurück gibt

#### Traceability:

Dieser Test, testet ob die get() Methode der CommandList auch die richtigen Elemente zurück gibt. Besonderer Schwerpunkt liegt dabei auf dem ersten Element der Liste da ein Null Element ist und somit "null" zurück gibt.

#### Spezifikation:

```
public void testGet() {
    // Fragt das letzte Element der Liste ab.
    assertEquals(this.commandl, list.get());

    // Testet ob die Rueckgabe in einer leeren Liste richtig ist.
    this.list = new CommandList();
    assertNull(this.list.get());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/04-001	Ingo Reimund	0.8	11.03.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/04-001/01	Ingo Reimund	behoben	11.03.2006

---

#### BESCHREIBUNG:

In einer leeren Liste wird ein Element zurück gegeben das nicht hinzugefügt wurde.

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/04-002	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/04-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.5 Einfrieren der Liste

TEST ID:	ERSTELLER:	DATUM:
CT002/05	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Nach der freeze() Methode darf die Liste nicht mehr verändert werden.

#### Traceability:

Wird die freeze() Methode ausgeführt, so darf die Liste nicht mehr auf Veränderungen reagieren. Es kann die Liste also noch beliebig durchlaufen werden, aber die add() Methode hat keinen Effekt mehr.

#### Spezifikation:

```
public void testFreeze () {
    this.list.freeze ();

    // Fuegt einen neuen Befehl an die Liste an.
    Command command = new MyStructCommand ();
    assertEquals (this.command1, this.list.get ());
    this.list.add (command);
    assertNotSame (command, this.list.get ());
    assertEquals (this.command1, this.list.get ());

    // Geht einen Schritt in der Liste zurueck und fuegt dann einen Befehl hinzu.
    this.list.stepBackward ();
    this.list.add (command);
    assertNotSame (command, this.list.get ());
    assertEquals (this.command0, this.list.get ());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/05-001	Ingo Reimund	0.8	11.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/05-002	Ingo Reimund	0.9	15.03.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/05-002/01	Ingo Reimund	behoben	15.03.2006

---

## BESCHREIBUNG:

Der hinzugefügte Befehl befindet sich in der Liste, obwohl diese nicht mehr verändert werden darf.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/05-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.6 Einen Schritt vor

TEST ID:	ERSTELLER:	DATUM:
CT002/06	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Geht in der Liste einen Schritt vor.

#### Traceability:

Diese Methode verschiebt lediglich den Pointer von der aktuellen Stelle eine Stelle weiter, wobei bei diesem Schritt die Liste am Ende nicht verlassen werden darf.

#### Spezifikation:

```
public void testStepForward () {

    // Geht vor das erste Element der Liste.
    this.list.stepFront();
    assertNull(this.list.get());

    // Verschiebt den Pointer auf das erste Element der Liste.
    this.list.stepForward();
    assertEquals(this.command0, this.list.get());

    // Verschiebt den Pointer auf das zweite Element der Liste.
    this.list.stepForward();
    assertEquals(this.command1, this.list.get());

    // Verschiebt den Pointer aus der Liste.
    this.list.stepForward();
    assertEquals(this.command1, this.list.get());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/06-001	Ingo Reimund	0.8	11.03.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/06-001/01	Ingo Reimund	behoben	11.03.2006

---

#### BESCHREIBUNG:

Vor dem ersten Element befindet sich ein unbekanntes Element, obwohl null erwartet wird.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/06-002	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/06-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.7 Einen Schritt zurück

TEST ID:  
CT002/07

ERSTELLER:  
Ingo Reimund

DATUM:  
10.03.2006

---

#### Beschreibung:

Geht in der Liste einen Schritt zurück.

#### Traceability:

Diese Methode verschiebt lediglich den Pointer von der aktuellen Stelle eine Stelle zurück, wobei bei diesem Schritt die Liste am Anfang nicht verlassen werden darf.

#### Spezifikation:

```
public void testStepBackward() {
    assertEquals(this.command1, this.list.get());

    // ein Schritt von der letzten Stelle zurueck.
    this.list.stepBackward();
    assertEquals(this.command0, this.list.get());

    // Ein Schritt zurueck, vor das erste Element.
    this.list.stepBackward();
    assertNull(this.list.get());

    // Schritt aus der Liste.
    this.list.stepBackward();
    assertNull(this.list.get());

    // Ein Schritt zu dem ersten Element. Testet das die Liste nicht
    // verlassen wird.
    this.list.stepForward();
    assertEquals(this.command0, this.list.get());
}
```

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/07-001	Ingo Reimund	0.8	11.03.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/07-001/01	Ingo Reimund	behoben	11.03.2006

---

## BESCHREIBUNG:

Vor dem ersten Element befindet sich ein unbekanntes Element, obwohl null erwartet wird.

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/07-001/02	Ingo Reimund	behoben	12.03.2006

---

## BESCHREIBUNG:

Die Liste wird verlassen.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/07-002	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/07-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.8 Sprung an den Anfang

TEST ID:	ERSTELLER:	DATUM:
CT002/08	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Springt an den Anfang der Liste.

#### Traceability:

Diese Methode Springt an den Anfang der Liste. Dabei ist es wichtig, das erst ein Schritt gemacht wird, bevor ein Element geholt wird. Dies begründet sich darin, dass das erste Element der Liste ein "null" Element ist und lediglich den Anfang der Liste signalisiert.

#### Spezifikation:

```
public void testStepFront() {

    // Sprung vor das erste Element der Liste.
    this.list.stepFront();
    assertNull(this.list.get());
    assertTrue(this.list.hasNext());
    assertFalse(this.list.hasPrevious());

    // Test ob das Erste Element das n"achste ist.
    this.list.stepForward();
    assertEquals(this.command0, this.list.get());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/08-001	Ingo Reimund	0.8	11.03.2006

---

#### Fehlerbericht:

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/08-001/01	Ingo Reimund	behoben	11.03.2006

---

#### BESCHREIBUNG:

Der Sprung führt direkt zum ersten Element und nicht davor.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/08-002	Ingo Reimund	0.9	15.03.2006

---

*Fehlerbericht:*

FEHLER ID:	AUTOR:	STATUS:	BEHEBUNG:
CT002/08-002/01	Ingo Reimund	behoben	15.03.2006

---

## BESCHREIBUNG:

Vor dem ersten Element befindet sich ein unbekanntes Element, obwohl null erwartet wird.

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/08-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

### 5.2.9 Klonen der Liste

TEST ID:	ERSTELLER:	DATUM:
CT002/08	Ingo Reimund	10.03.2006

---

#### Beschreibung:

Erstellt eine Kopie der Liste.

#### Traceability:

Diese Methode erstellt eine Kopie der Liste mit den Befehlen. Die Befehle selbst werden aber nicht kopiert. Dabei ist es besonders wichtig, dass alle Element in der gleichen Reihenfolge liegen wie in der original Liste. Ebenfalls dürfen nachträgliche Änderungen an der Kopie nur die Kopie betreffen und nicht das original.

#### Spezifikation:

```
public void testClone() {
    CommandList clone = (CommandList) this.list.clone();

    // Spring an den Anfang und testet das die Element in der
    // richtigen Reihenfolge liegen.
    clone.stepFront();
    assertEquals(this.command0, clone.next());
    assertEquals(this.command1, clone.next());

    // Vergleicht das erste Element der beide Listen
    clone.stepBackward();
    this.list.stepBackward();
    assertEquals(this.command0, this.list.get());
    assertEquals(this.command0, clone.get());

    // Fügt der Kopie ein neues Element hinzu, dieses darf
    // nicht im original vorhanden sein.
    Command command = new MyStructCommand();
    clone.add(command);
    this.list.stepForward();
    assertEquals(command, clone.get());
    assertNotSame(command, this.list.get());
}
```

#### Testlauf

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/08-001	Ingo Reimund	0.8	11.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/08-002	Ingo Reimund	0.9	15.03.2006

---

*Es wurde kein Fehler gefunden.*

**Testlauf**

TESTLAUF ID:	TESTER:	VERSION:	DATUM:
CT002/08-003	Ingo Reimund	0.9	16.03.2006

---

*Es wurde kein Fehler gefunden.*

## A Glossar

### B

---

#### **Black-Box-Test**

Bei einem Black-Box-Test handelt es sich um einen Test der ohne nähere Kenntnis der Implementierung durchgeführt wird.

### F

---

#### **Framework**

Unter einem Framework (deutsch: Rahmenkonstrukt) versteht man ein Programm, das eine Art Rahmen bereitstellt, in das man andere Programmbestandteile reinladen kann. Diese Programmbestandteile bieten die eigentliche Funktionalität des Programmes. Jedes Programmbestandteil kann auf die gleichen Funktionen innerhalb des Frameworks zugreifen und bietet nach außen dadurch eine ähnliche oder gleiche Funktionsweise. Ein bekanntes Beispiel ist das Eclipse Framework oder die Unit Testframeworks.

#### **Weiter Informationen:**

<http://de.wikipedia.org/wiki/framework>

### J

---

#### **JUnit**

JUnit ist ein Test Framework, mit dessen Hilfe das Testen von Teilkomponenten eines Systems ermöglicht wird. Ist ein Test mit JUnit geschrieben und läuft dieser durch, so kann an den zutestenden Klassen gearbeitet und diese immer wieder auf ihre Funktionalität überprüft werden.

#### **Weiter Informationen:**

<http://www.junit.org>

### U

---

#### **Use-Case-basiert**

Bei Use-Case-basiert handelt es sich um einen auf Use-Cases aufbauende Test der einen Use-Case als Ausgangssituation nimmt und die Randbedingungen der Use-Cases betrachtet und darauf aufbauen einen über mehrere Klassen verlaufenden Black-Box-Test erstellt.

## **Z**

---

### **Zweigabdeckung**

Bei der Zweigabdeckung handelt es sich um ein Testverfahren, das eine genaue Kenntnis über die Implementierung besitzt und jeden Zweig der bei einer Bedingung entsteht auf das richtige arbeiten überprüft.