Dietrich Boles

TDD-Hamster-Simulator

Testgetriebene Entwicklung mit dem Java-Hamster-Modell

Version 1.0

Benutzungshandbuch

18.08.2016

Inhaltsverzeichnis

1 Einleitung

Programmieranfänger haben häufig Schwierigkeiten damit, dass sie beim Programmieren ihre normale Gedankenwelt verlassen und in eher technisch-orientierten Kategorien denken müssen, die ihnen von den Programmiersprachen vorgegeben werden. Gerade am Anfang strömen oft so viele inhaltliche und methodische Neuigkeiten auf sie ein, dass sie das Wesentliche der Programmierung, nämlich das Lösen von Problemen, aus den Augen verlieren.

1.1 Das Hamster-Modell

Das Hamster-Modell ist mit dem Ziel entwickelt worden, dieses Problem zu lösen. Mit dem Hamster-Modell wird Programmieranfängern ein einfaches, aber mächtiges Modell zur Verfügung gestellt, mit dessen Hilfe Grundkonzepte der imperativen und objektorientierten Programmierung auf spielerische Art und Weise erlernt werden können. Programmierer entwickeln so genannte Hamster-Programme, mit denen sie virtuelle Hamster durch eine virtuelle Landschaft steuern und bestimmte Aufgaben lösen lassen. Die Anzahl der gleichzeitig zu berücksichtigenden Konzepte wird im Hamster-Modell stark eingeschränkt und nach und nach erweitert.

Prinzipiell ist das Hamster-Modell programmiersprachenunabhängig. Zum praktischen Umgang mit dem Modell wurde jedoch bewusst die Programmiersprache Java als Grundlage gewählt. Java – auch als „Sprache des Internet“ bezeichnet – ist eine moderne Programmiersprache, die sich in den letzten Jahren sowohl im Ausbildungsbereich als auch im industriellen Umfeld durchgesetzt hat.

Zum Hamster-Modell existieren drei Bücher. In dem ersten Buch „Programmieren spielend gelernt mit dem Java-Hamster-Modell“ werden allgemeine Grundlagen der Programmierung erläutert sowie Konzepte der imperativen Programmierung (Anweisungen, Schleifen, Prozeduren, Typen, Variablen, Parameter, Rekursion, ...) eingeführt. Darauf aufbauend behandelt das zweite Buch „Objektorientierte Programmierung spielend gelernt mit dem Java-Hamster-Modell“ alle wichtigen Konzepte der objektorientierten Programmierung (Objekte, Klassen, Vererbung, Polymorphie, Interfaces, Exceptions, Zugriffsrechte, Pakete, ...). Im dritten Band „Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell“ geht es um die parallele Programmierung mit Java-Threads (Threads, Kommunikation, Synchronisation, Deadlocks, ...). Die Bücher sind im Springer-Vieweg-Verlag erschienen und können im Buchhandel erworben werden. Alle drei Bücher sind insbesondere für Schüler und Studierende ohne Programmiererfahrung empfehlenswert. Die Bücher sind als Grundlage für Programmierkurse sowie zum Selbststudium geeignet. Dazu enthalten sie viele Beispielprogramme und Übungsaufgaben.

Zum Hamster-Modell gibt es unter folgendem URL die sogenannte Hamster-Website: **http://www.java-hamster-modell.de.** Die Hamster-Website enthält viele Informationen und Materialien rund um das Hamster-Modell. Insbesondere stehen auf der Hamster-Website auch alle drei Bücher in einer kostenlosen PDF-Version zum Download bereit.

1.2 Der Hamster-Simulator

Auf der Hamster-Website steht auch der „Hamster-Simulator“ kostenlos zur Verfügung; ein Programm, mit dem Hamster-Programme erstellt und ausgeführt werden können. Neben den drei Büchern kommt dem Hamster-Simulator dabei eine ganz wichtige Bedeutung zu, denn Programmieren lernt man nicht durch lesen. Man muss üben, üben, üben. Und genau dazu dient der Simulator.

Für diejenigen, die noch nie mit dem Hamster-Modell in Berührung gekommen sind, sei angeraten, vor der testgetriebenen Entwicklung mit dem TDD-Hamster-Simulator zunächst einmal erste Schritte mit dem Original-Hamster-Simulator zu unternehmen, um das Hamster-Modell kennenzulernen.

1.3 Der TDD-Hamster-Simulator

Der TDD-Hamster-Simulator ist ein spezieller Hamster-Simulator, mit dem wir erproben möchten, inwieweit Programmieranfänger bereits mit der sogenannten testgetriebenen Entwicklung von Programmen konfrontiert werden können. Wenn die Erprobung erfolgreich war, werden wir die Konzepte in den Original-Hamster-Simulator integrieren. In diesem Dokument wird die Benutzung des TDD-Hamster-Simulators beschrieben.

Grundlage des TDD-Hamster-Simulators ist das imperative Hamster-Modell (Band 1 der Hamster-Bücher). Objektorientierte und parallele Hamster-Programme werden aktuell nicht unterstützt.

Informationen und Materialien zur testgetriebenen Entwicklung mit dem TDD-Hamster-Simulator können Sie über folgende Unterwebsite der Hamster-Website abrufen: http://www.java-hamster-modell.de/tdd.html.

1.4 Ihre Meinung

Wenn Sie den TDD-Hamster-Simulator einsetzen, sind wir sowohl an positiven als auch negativen Meinungen sehr interessiert. Schreiben Sie einfach eine Email an dietrich@boles.de. Vielen Dank!

2 Installation und Starten des TDD-Hamster-Simulators

Der TDD-Hamster-Simulator läuft zurzeit auf Windows-, Macintosh-, Linux- und Solaris-Rechnern. Da er in Java geschrieben ist, müsste er eigentlich auch auf allen anderen Rechnern laufen, für die eine Java JVM und ein JDK existiert.

2.1 Laden und Installation eines JDK

Der TDD-Hamster-Simulator ist ein in Java geschriebenes Programm. Um es ausführen zu können, muss auf Ihrem Rechner eine Java-Laufzeitumgebung installiert werden. Generell gibt es in Java zwei Varianten von Java-Laufzeitumgebungen: ein JDK (Java SE Development Kit) bzw. ein JRE (Java SE Runtime Environment). Für die Nutzung des TDD-Hamster-Simulators ist ein JDK mindestens der Version 8 erforderlich. Gegenüber einem JRE, das deutlich kleiner ist als ein JDK, beinhaltet ein JDK noch weitere Werkzeuge zur Entwicklung von Java-Programmen, wie bspw. einen Compiler. Dieser wird programmintern benötigt. Die jeweils aktuelle Version des JDK kann über den folgenden URL kostenlos aus dem WWW geladen werden: http://www.oracle.com/technetwork/java/javase/downloads/index.html

Das JDK wird standardmäßig von der Firma Oracle für die Betriebssysteme Windows, Linux und Solaris zur Verfügung gestellt, leider nicht für Macintosh-Betriebssysteme.

Nachdem Sie ein JDK auf Ihren Rechner geladen haben, müssen Sie es installieren. Das geschieht normalerweise durch Ausführen der geladenen Datei (in Windows Doppelklick auf die `.exe`-Datei). Sie werden dann durch die weitere Installation geführt. Videos zur Installation eines JDK gibt es zuhauf auf Youtube (https://www.youtube.com).

2.2 Laden und Installation des TDD-Hamster-Simulators

Von der Hamster-Website können Sie über folgenden Link den TDD-Hamster-Simulator herunterladen: http://www.java-hamster-modell.de/tdd.html. Die Datei „tdd-hamstersimulator.zip“ müssen Sie auf Ihren Rechner laden und anschließend entpacken. Vermutlich haben Sie bereits ein entsprechendes Programm zum Entpacken auf Ihrem Rechner. Ansonsten können Sie bspw. von der folgenden Website eines laden und installieren: http://www.chip.de/Downloads_13649224.html?tid1=38986&tid2=0

2.5 Starten des TDD-Hamster-Simulators

Nachdem Sie ein JDK sowie den TDD-Hamster-Simulator wie oben beschrieben auf Ihrem Rechner installiert haben, können Sie den Simulator starten. Dies geschieht folgendermaßen.

- Unter Windows: Führen Sie mit der Maus einen Doppelklick auf die Datei `tdd-hamstersimulator.jar` oder die Datei `tdd-amstersimulator.bat` aus. [1]
- Unter Linux und Solaris: Rufen Sie in dem Ordner, in dem sich die Datei `tdd-hamstersimulator.jar` befindet, folgenden Befehl auf: `java -jar tdd-hamstersimulator.jar`
- Unter Macintosh (OS X): Führen Sie mit der Maus einen Doppelklick auf die Datei `tdd-hamstersimulator.jar` aus.

Herzlichen Glückwunsch! Sie können mit der testgetriebenen Entwicklung von Hamster-Programmen beginnen!

Hinweis:

Wenn es Probleme mit dem Doppelklick unter Windows gibt, kann es sein, dass die Endung .jar dem falschen Programm bzw. Java-Interpreter zugeordnet ist. Testen Sie dies, indem Sie in einem Eingabeaufforderungsfenster folgendes eingeben und in etwa die folgenden Ergebnisse angezeigt bekommen:

Eingabe: `assoc .jar`

Ausgabe: `.jar=jarfile`

Eingabe: `ftype jarfile`

Ausgabe: `jarfile="C:\Program Files\Java\jdk\bin\javaw.exe" -jar "%1" %*`

Den Java-Interpreter, der standardmäßig beim Doppelklick auf eine .jar-Datei aufgerufen werden soll, können Sie durch folgende Eingabe in einem Eingabeaufforderungsfenster ändern:

```
ftype jarfile="pfad\javaw.exe" -jar "%1" %*
```

wobei „pfad“ entsprechend Ihrer JDK-Installation geändert werden muss, bspw. so:

```
jarfile="C:\Program Files\Java\jdk1.8.0\bin\javaw.exe" -jar "%1" %*
```

[1] Eine weitere Alternative besteht darin, ein Eingabeaufforderung-Fenster zu öffnen, sich in den Ordner zu begeben, in dem sich die Datei `tdd-hamstersimulator.jar` befindet, und dort folgenden Befehl einzugeben: `java -jar tdd-hamstersimulator.jar`.

3 Testgetriebene Entwicklung

Eine schöne Einführung in die testgetriebene Entwicklung bietet das Buch „Testgetriebene Entwicklung mit JUnit & FIT“ von Frank Westpfahl, das über folgenden URL als kostenloses PDF-Dokument geladen werden kann: http://www.frankwestphal.de/ftp/Westphal_Testgetriebene_Entwicklung.pdf

3.1 Was bedeutet „Testen“?

Verallgemeinert geht es beim Testen von Software um die Überprüfung der Erfüllung der für ihren Einsatz definierten Anforderungen und die Messung der Qualität. Konkreter geht es um das Aufspüren von Fehlern und deren Beseitigung. Fehler können dabei vielfältiger Natur sein. Zum einen kann ein Programm in bestimmten Situationen fehlerhafte Ergebnisse produzieren oder sich fehlerhaft verhalten, zum anderen kann ein Programm „abstürzen“, d.h. sich ungewollt beenden.

Welche Fehler kennen wir beispielsweise aus unseren Hamster-Programmen? Zunächst sind natürlich die klassischen Laufzeitfehler zu nennen: Der Hamster läuft gegen eine Mauer, er versucht ein Korn zu nehmen, obwohl gar keines da ist, oder er versucht ein Korn abzulegen, obwohl er keines im Maul hat. Andere Laufzeitfehler können beim Umgang mit Zahlen auftreten, wenn wir beispielsweise versuchen, eine Zahl durch 0 zu dividieren.

Endlosschleifen, d.h. Schleifen, bei denen die Schleifenbedingung niemals falsch wird, charakterisieren einen anderen Typ von Fehlern. Das Programm wird dabei nie beendet. Analoges gilt für Endlosrekursion, d.h. es wird keine Abbruchbedingung für einen rekursiven Funktionsaufruf erreicht. Anders als bei Endlosschleifen führt eine Endlosrekursion aber bedingt durch bestimmte Aktionen des Laufzeitsystems im Allgemeinen zu einem fehlerhaften Programmabbruch.

Ein Hamster-Programm ist auch dann fehlerhaft, wenn der Hamster die ihm gestellte Aufgabe nicht korrekt löst. Eine Aufgabe ist dabei nur dann vollständig korrekt gelöst, wenn sie der Hamster aus allen möglichen und erlaubten Ausgangssituationen heraus und unter Einhaltung aller vorgegebenen Randbedingungen löst.

Testen im Rahmen der Softwareentwicklung bedeutet nun, möglichst alle derartigen Fehler zu finden und anschließend zu beseitigen. Bei Hamster-Programmen ist es dabei noch relativ einfach, Fehler zu finden, denn wir sehen ja im Hamster-Simulator bei jeder Aktion des Hamsters am Zustand des Hamster-Territoriums, ob er korrekt agiert hat oder nicht. Bei komplexer Nicht-Hamster-Software ist Testen jedoch ein umfangreicher Teilprozess der Softwareentwicklung und es gibt sogar Programme für die es unmöglich zu beweisen ist, dass sie vollständig korrekt sind.

3.2 Probleme beim klassischen Testen

Bei der klassischen Softwareentwicklung ist das Testen ein nachgeschalteter Prozess, d.h. zunächst wird die Software entwickelt, anschließend wird getestet. Werden Fehler entdeckt, werden diese beseitigt und es wird erneut getestet. Dabei mögen viele Programmierer das Testen ganz und gar nicht und es wird häufig vernachlässigt. Was sind die Gründe hierfür?

Zunächst sei ein psychologischer Grund zu nennen: Wenn wir nach Fehlern suchen, müssen wir uns ja eingestehen, dass wir als Programmierer nicht perfekt sind. Also suchen wir lieber erst gar nicht oder nur oberflächlich nach Fehlern. Dieser Grund impliziert, dass es sinnvoll ist, seine Programme auch von anderen testen zu lassen.

Ein weiterer Grund ist häufig die Zeit. Wir müssen unser Programm bis zu einem bestimmten Zeitpunkt fertig haben. Da aber das Testen der letzte Teilprozess der Softwareentwicklung ist, bleibt hierfür keine Zeit mehr. Das Testen ist das Erste, was über Bord geht, wenn es zeitlich eng wird.

Getestet wird häufig auch nur wenig systematisch sondern eher ad hoc. Man testet eher spontan mal dies und das, als sich genau zu überlegen, was eigentlich getestet werden muss und nach einem ordentlichen und gut vorbereiteten Plan vorzugehen.

3.3 Was ist Testgetriebene Entwicklung?

Motivation und Ziel der sogenannten „Testgetriebenen Entwicklung“ oder auf Englisch „Test-Driven Development“ – abgekürzt TDD – ist eine höhere Qualität von entwickelter Software durch die Beseitigung der oben genannten Probleme. Bei der TDD wird das Testen in den Mittelpunkt der Softwareentwicklung gestellt und bekommt eine mindestens genauso große Gewichtung im Entwicklungsprozess wie die eigentliche Programmierung. Bereits vor der Entwicklung des eigentlichen Programmcodes macht sich ein Programmierer Gedanken über das Testen. Er schreibt als erstes speziellen Testcode, dessen Ausführung natürlich erst fehlschlägt, da ja der eigentliche Code noch gar nicht existiert. Erst danach schreibt der Programmierer den Code und testet ihn dann unmittelbar mit dem Testcode. Programme werden bei der TDD nicht als Ganzen entwickelt und getestet sondern inkrementell und in kleinen Schritten. Die Vorgehensweise ist dabei sehr systematisch und gut geplant.

Die TDD unterscheidet zwei Typen von Tests. Unit-Tests überprüfen funktionale Einzelteile eines Programms auf korrekte Funktionalität während mittels Akzeptanztests sichergestellt werden soll, dass das Programm die gestellten Anforderungen erfüllt. Im Mittelpunkt der TDD mit dem Hamster-Modell stehen dabei die Unit-Tests.

3.4 Methodik bei der Testgetriebenen Entwicklung

Im Gegensatz zu vielen Softwareentwicklungsmethoden, bei denen am Ende eines Zyklus beziehungsweise Projektes Tests erstellt bzw. durchgeführt werden, gilt bei der TDD: Test zuerst. Die TDD arbeitet iterativ und lässt sich wie folgt skizzieren:

1. Erstelle einen Test für noch nicht implementierte Funktionalität.
2. Durchlaufe alle Tests und lasse den neu erstellten Test fehlschlagen.
3. Implementiere die vom neu erstellten Test erwartete Funktionalität.
4. Durchlaufe wieder alle Tests und diesmal sollten alle Tests erfolgreich durchlaufen werden.
5. Refaktorisiere den Quellcode (entferne Duplikate, verbessere die Struktur und Lesbarkeit), ohne dass bei einem erneuten Durchführen der Tests einer fehlschlagen würde.
6. Wenn noch weitere Funktionalität implementiert werden soll, beginne bei Schritt 1.

Bevor eine Funktionalität implementiert wird, wird diese durch einen Test spezifiziert (erster Schritt). Danach werden alle Tests durchlaufen und der neu erstellte Test sollte fehlschlagen (zweiter Schritt). Durch den Fehlschlag wird garantiert, dass der Test neue Funktionalität beschreibt und nicht bereits implementierte beziehungsweise getestete Funktionen testet. Der Entwickler wird so zur Implementierung der Funktionalität getrieben (dritter Schritt). Sobald die Funktionalität implementiert wurde, werden wieder alle Tests durchlaufen (vierter Schritt). Würde der neu erstellte Test immer noch fehlschlagen, kann der Entwickler den Fehler auf die zuvor implementierte Funktionalität eingrenzen und beheben. Wenn alle Tests erfolgreich durchlaufen werden, wird dem Entwickler bestätigt, dass die Implementierung korrekt ist. Als nächstes wird der Quellcode wenn nötig refaktorisiert (fünfter Schritt). Es werden Duplikate aus dem Quellcode entfernt und die Struktur sowie Lesbarkeit werden verbessert. Nach der Refaktorisierung sollten weiterhin alle Tests erfolgreich durchlaufen werden. Falls Tests fehlschlagen, weist dieses auf einen Fehler bei der Refaktorisierung hin. Solange noch weitere Funktionalität implementiert werden soll, wird wieder beim ersten Schritt begonnen (sechster Schritt).

3.5 Unit-Tests

Unit-Tests weisen einen bestimmten Aufbau und ein bestimmtes Vorgehen auf. Sie bestehen aus vielen Testfällen, die zu Testklassen und diese wiederum zu Test-Suites zusammengefasst werden.

3.5.1 Testfälle

Durch einen einzelnen Testfall (englisch TestCase) wird eine stark begrenzte Funktionalität, häufig eine einzelne Operation getestet. Testfälle weisen im Allgemeinen folgenden Grundaufbau auf:

- Anfangs wird ein Ausgangszustand initialisiert.
- Es wird die zu testende Funktionalität ausgeführt; bspw. eine Prozedur aufgerufen.

- Das Ergebnis bzw. der erreichte Zustand wird als Ist-Wert mit dem erwarteten Soll-Wert verglichen. Stimmen sie überein, war der Test erfolgreich, andernfalls ist der Test fehlgeschlagen.

Der anfangs initialisierte Ausgangszustand wird auch als Text-Fixture bezeichnet.

3.5.2 Testklassen

Wenn verschiedene Testfälle dieselbe Text-Fixture besitzt, sollten diese in einer Testklasse zusammengefasst werden. Die gemeinsame Text-Fixture kann dann in einen getrennten Bereich ausgelagert werden. Wird eine Testklasse ausgeführt, werden alle enthaltenen Testfälle ausgeführt. Vor jedem Testfall wird dabei die Test-Fixture ausgeführt. Die Testfälle dürfen nicht voneinander abhängig sein, denn es ist nicht vorhersehbar, in welcher Reihenfolge sie ausgeführt werden. Schlägt ein einzelner Testfall fehl, wird sofort der gesamte Testlauf abgebrochen.

3.5.3 Testsuites

Testsuites stellen ein weiteres Strukturierungsmittel in Unit-Tests dar. Sie erlauben es, mehrere Testklassen bzw. andere Testsuites zu einer Gruppe zusammenzufassen. Wird eine Testsuite ausgeführt, werden alle darin enthalten Tests und Testsuites ausgeführt. Auch hier ist die Reihenfolge der Ausführung nicht definiert.

3.5.4 Testrunner

Testrunner sind Werkzeuge, mit denen Unit-Tests ausgeführt werden. Im Allgemeinen enthalten Testrunner heutzutage ein grafisches Frontend. Bekannt geworden sind Testrunner durch ihre Ausgabe von grünen und roten Balken. Schlägt ein Test fehl, werden ein roter Balken angezeigt und weitere Hinweise zum Fehler ausgegeben. War die Ausführung aller Testfälle erfolgreich, wird das durch einen grünen Balken signalisiert.

4 Der TDD-Hamster-Simulator

Abbildung 4.1 zeigt einen Screenshot vom TDD-Hamster-Simulator. Für eine Einführung in seinen Aufbau und seine Nutzung schauen Sie sich bitte das folgende Video an:

http://www.java-hamster-modell.de/tdd/1-Benutzung/1-Benutzung.html

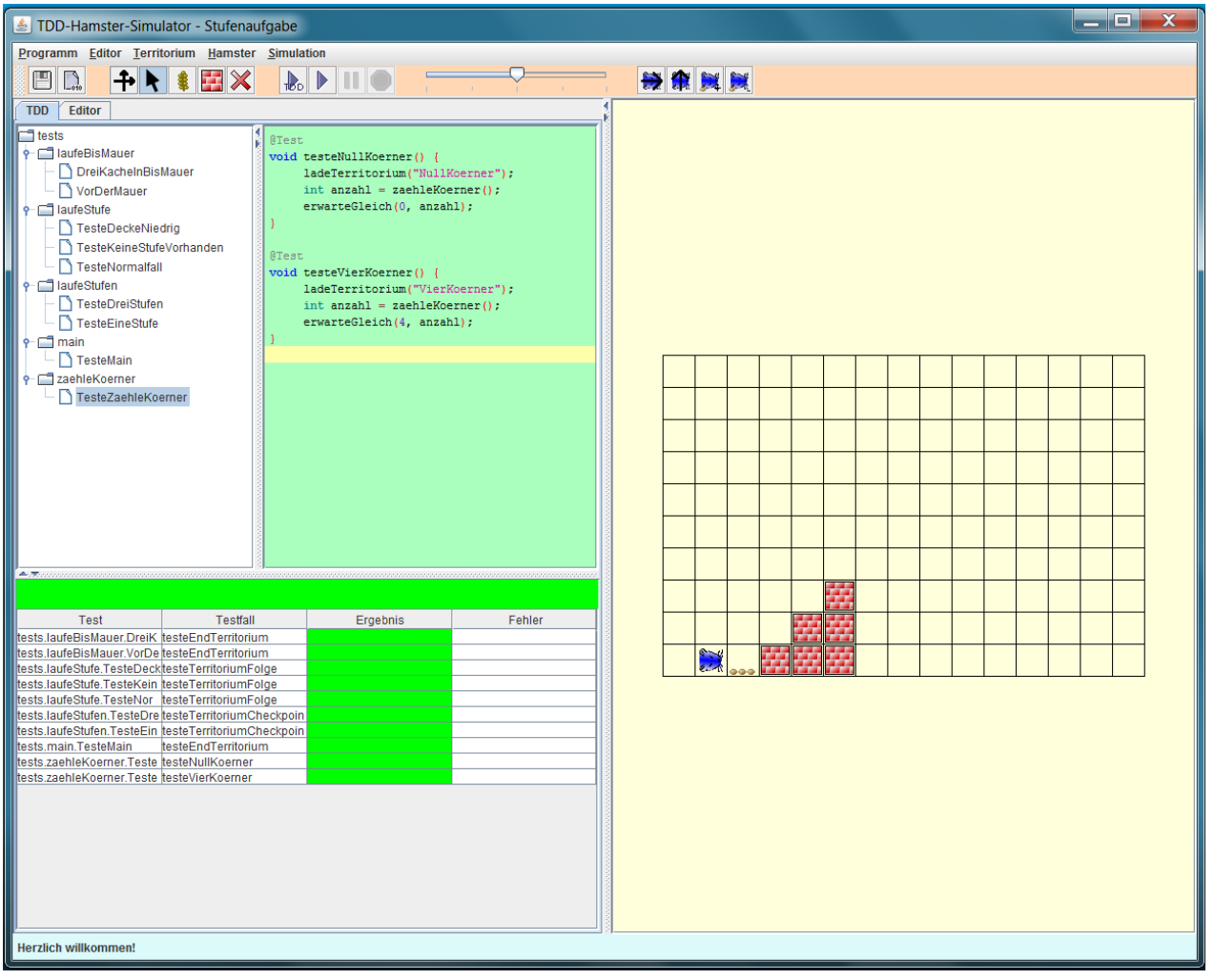

Abbildung 4.1: TDD-Hamster-Simulator

4.1 Motivation für einen TDD-Hamster-Simulator

Im Rahmen seiner Bachelorarbeit wurde von Florian Schmalriede der sogenannte TDD-Hamster-Simulator entwickelt. Der TDD-Hamster-Simulator enthält in vereinfachter Form alle Funktionalität, um wie mit dem Original-Hamster-Simulator Hamster-Programme entwickeln zu können. Er enthält darüber hinaus aber erweiterte Funktionalität, mit der es möglich ist, auf einfache Art und Weise Unit-Tests für

Hamster-Programme zu erstellen und auszuführen. Der TDD-Hamster-Simulator ist auf imperative Java-Hamster-Programme beschränkt.

Sinn und Zweck des TDD-Hamster-Simulators ist es, Programmieranfänger frühzeitig Gelegenheit dazu zu geben, die Konzepte und die Methodik der testgetriebene Entwicklung kennenzulernen und zu erproben. Die hier erlernten Konzepte können dann später relativ einfach auf bspw. die TDD mit Java und JUnit übertragen werden. JUnit ist das meist-genutzte Framework für Unit-Tests von Java-Programmen.

4.2 Hamster-Unit-Tests

Bei Hamster-Unit-Tests werden im Allgemeinen Zustände von Hamster-Territorien miteinander verglichen. Der Zustand eines Hamster-Territoriums wird dabei durch folgende Eigenschaften gebildet:

- Die Größe des Territoriums (Anzahl an Reihen und Spalten)
- Belegung der einzelnen Kacheln des Territoriums durch Mauern bzw. eine bestimmte Anzahl an Körnern
- Positionierung des Hamsters im Territorium
- Blickrichtung des Hamsters im Territorium
- Anzahl an Körnern im Maul des Hamsters

Der Zustand zweier Territorien ist gleich, wenn alle diese Eigenschaften gleich sind.

Im TDD-Hamster-Simulator gibt es vier verschiedene Typen von Tests: End-Territorium-Test, Territorium-Folge-Tests, Territorium-Checkpoint-Tests und Anwender-Tests. Die Anwender-Tests sind am ehesten mit JUnit-Tests zu vergleichen. Sie erfordern die explizite Entwicklung von Quellcode für das Testen. Die anderen drei Testtypen nutzen grafisch-interaktive Hilfsmittel zum Entwickeln von Tests und erleichtern damit Programmieranfängern die Arbeit.

4.3 End-Territorium-Test

Mit Hilfe eines End-Territorium-Tests kann überprüft werden, ob nach der Ausführung einer Prozedur ein bestimmtes End-Territorium erreicht wird. Zunächst wird bei einem End-Territorium-Test ein Start-Territorium angegeben. Anschließend folgt ein Prozeduraufruf. Zum Schluss wird ein End-Territorium festgelegt.

Bei der Ausführung eines End-Territorium-Tests wird zunächst der Zustand des Start-Territoriums hergestellt. Danach folgt die Ausführung der Prozedur. Der Test ist genau erfolgreich, wenn nach erfolgreich beendeter Prozedurausführung der erreichte Zustand gleich dem Zustand des angegebenen End-Territoriums ist.

Als Prozedur kann natürlich auch die main-Prozedur aufgerufen werden und somit ein komplettes Programm getestet werden. Ebenfalls ist es möglich, eine Folge von Prozeduraufrufen oder andere Anweisungen anzugeben und zu testen.

4.4 Territorium-Folge-Tests

Bei einem Territorium-Folge-Test wird nicht nur der erreichte Endzustand eines Prozeduraufrufes mit einem erwarteten Endzustand des Territoriums verglichen, vielmehr erfolgt eine Überprüfung nach jeder einzelnen Zustandsänderung durch ein Programm. Ein Territorium-Folge-Test besteht dazu aus einem anzugebenen Start-Territorium, einem Prozeduraufruf und einer Folge von Territorien.

Bei der Ausführung eines Territorium-Folge-Tests wird zunächst der Zustand des Start-Territoriums hergestellt. Danach folgt die Ausführung der Prozedur. Dabei wird nach jeder Zustandsänderung des Territoriums überprüft, ob der erreichte Zustand dem Zustand des nächsten Territoriums in der Folge der Territorien entspricht. Ist dies nicht der Fall, wird der Test unmittelbar abgebrochen und ein Fehlschlag signalisiert. Der Test ist erfolgreich, wenn das Programm erfolgreich beendet wurde und alle Territorien der Folge in der entsprechenden Reihenfolge erreicht wurden.

4.5 Territorium-Checkpoint-Tests

Während bei einem Territorium-Folge-Test alle durch ein Programm bewirkten Zustandsänderungen mit erwarteten Zuständen abgeglichen werden, werden bei einem Territorium-Checkpoint-Test nur bestimmte Folgezustände überprüft. Auch bei einem Territorium-Checkpoint-Test werden zunächst ein Start-Territorium und ein Prozeduraufruf angegeben. Danach folgt eine Folge von Territorien, welche bei der Ausführung der Prozedur in der entsprechenden Reihenfolge erreicht werden müssen. Zusätzlich lassen sich noch weitere Territorien auf einer sogenannten Blacklist angegeben. Territorien der Blacklist dürfen zu keinem Zeitpunkt der Ausführung der Prozedur erreicht werden, sonst schlägt der Test direkt fehl.

Bei der Ausführung eines Territorium-Folge-Tests wird zunächst der Zustand des Start-Territoriums hergestellt. Danach folgt die Ausführung der Prozedur. Wurden nach erfolgreich beendeter Ausführung der Prozedur alle angegebenen Territorien in der entsprechenden Reihenfolge erreicht und wurde kein Territorium der Blacklist erreicht, so war der Test erfolgreich.

4.6 Anwender-Tests

Anwender-Tests im TDD-Hamster-Simulator entsprechen vom Aufbau und der Art und Weise ihrer Anwendung Tests in JUnit, dem bekanntesten Test-Framework für Java.

4.6.1 Aufbau von Anwender-Tests

Ein einzelner Testfall wird durch eine Prozedur repräsentiert, der die Annotation `@Test` vorangestellt wird. In einer speziellen Test-Datei, die eine Testklasse repräsentiert, lassen sich mehrere Testfälle, also Prozeduren zusammenfassen. Zum Aufbau einer

gemeinsamen Test-Fixture für die Testfälle einer Testklasse kann eine spezielle Prozedur definiert, der die Annotation `@Einrichtung` vorangestellt wird.

Als einfaches Beispiel sollen im Folgenden mal die Prozeduren `kehrt` und `rechtsUm` getestet werden:

```
void kehrt() {
    linksUm();
    linksUm();
}

void rechtsUm() {
    linksUm();
    kehrt();
}
```

Dazu legen wir folgende Testklasse an (die lade- und erwarte-Prozeduren werden weiter unten beschrieben):

```
@Einrichtung
void richteEin() {
    ladeTerritorium("NachOsten");
}

@Test
void testeKehrt() {
    kehrt();
    erwarteBlickrichtungWest();
}

@Test
void testeRechtsUm() {
    rechtsUm();
    erwarteBlickrichtungSued();
}
```

4.6.2 Ausführung von Anwender-Tests

Bei der Ausführung eines Anwender-Tests werden die einzelnen Testfälle, d.h. @Test-Prozeduren, der jeweiligen Testklasse ausgeführt. In welcher Reihenfolge dies geschieht, ist undefiniert. Die Testfälle sollten also unabhängig voneinander sein. Jeweils unmittelbar vor der Ausführung jeder @Test-Prozedur wird dabei, falls vorhanden, automatisch die @Einrichtung-Prozedur aufgerufen. Schlägt ein Testfall fehl, wird unmittelbar der gesamte Anwender-Test abgebrochen.

4.6.3 Verwalten von Territorien

Über das Territorium-Menü im TDD-Hamster-Simulator lassen sich Territorien speichern und laden. Dazu muss jeweils eine Datei angegeben werden. Durch die Prozedur `ladeTerritorium(String dateiname)`, die in Testklassen aufgerufen werden kann,

kann ein entsprechend abgespeichertes Territorium geladen werden. Im obigen Beispiel wird in der @Einrichtung-Prozedur richteEin jeweils vor der Ausführung der beiden @Test-Prozeduren das in der Datei `namens` „NachOsten“ abgespeicherte Territorium geladen.

4.7 Erwartungen

Der TDD-Hamster-Simulator ermöglicht es, bestimmte Eigenschaften und Werte zu überprüfen, die erfüllt sein müssen, damit ein Test erfolgreich durchläuft. Dazu wird eine Vielzahl an speziellen Prozeduren zur Verfügung gestellt, die in Testklassen aufgerufen werden können. Diese Prozeduren beginnen alle mit dem Präfix „erwarte“.

Im obigen Beispiel werden bspw. die Prozeduren `erwarteBlickrichtungWest` und `erwarteBlickrichtungSued` aufgerufen, in denen überprüft wird, ob die aktuelle Blickrichung des Hamsters Westen bzw. Süden ist.

Wird eine solche Erwartung nicht erfüllt, werden der Testfall und auch der gesamte Test unmittelbar abgebrochen. Der Test ist fehlgeschlagen, ein roter Balken erscheint und es werden Hinweise zum möglichen Fehler ausgegeben.

4.7.1 Hamster-Erwartungen

Der TDD-Hamster-Simulator stellt folgende `erwarte…`-Prozeduren zur Verfügung, mit denen Erwartungen an den aktuellen Zustand des Hamsters ausgedrückt werden können:

```
void erwarteBlickrichtungNord()
void erwarteBlickrichtungNichtNord()
void erwarteBlickrichtungWest()
void erwarteBlickrichtungNichtWest()
void erwarteBlickrichtungSued()
void erwarteBlickrichtungNichtSued()
void erwarteBlickrichtungOst()
void erwarteBlickrichtungNichtOst()
void erwarteVornFrei()
void erwarteNichtVornFrei()
void erwarteKornDa()
void erwarteNichtKornDa()
void erwarteMaulLeer()
void erwarteNichtMaulLeer()
void erwarteReiheGleich(int reihe)
void erwarteReiheUngleich(int reihe)
void erwarteSpalteGleich(int spalte)
void erwarteSpalteUngleich(int spalte)
void erwartePositionGleich(int reihe, int spalte)
void erwartePositionUngleich(int reihe, int spalte)
void erwarteAnzahlKoernerImMaulGleich(int anzahl)
void erwarteAnzahlKoernerImMaulUngleich(int anzahl)
```

Die Namen der Prozeduren sprechen eigentlich für sich. Bspw. schlägt die Prozedur `erwarteVornFrei` fehl, wenn vor dem Hamster eine Mauer ist, die Prozedur `erwarteNichtVornFrei` schlägt fehl, wenn vor dem Hamster keine Mauer ist.

Bei einigen Prozeduren müssen als Parameter die erwartete Reihe bzw. Spalte des Hamsters im Territorium übergeben werden. Die Reihennummerierung erfolgt dabei von oben nach unten und die Spaltennummerierung von links nach rechts. Sie beginnt jeweils bei 0, d.h. wenn erwartet wird, dass der Hamster in der obersten Reihe steht, muss dazu folgender Prozeduraufruf erfolgen: erwarteReiheGleich(0);

4.7.2 Territorium-Erwartungen

Der TDD-Hamster-Simulator stellt folgende `erwarte…`-Prozeduren zur Verfügung, mit denen Erwartungen an den aktuellen Zustand des Territoriums ausgedrückt werden können:

```
void erwarteTerritoriumGleich(String name)
void erwarteAnzahlReihenGleich(int anzahl)
void erwarteAnzahlReihenUngleich(int anzahl)
void erwarteAnzahlSpaltenGleich(int anzahl)
void erwarteAnzahlSpaltenUngleich(int anzahl)
void erwarteGroeszeGleich(int reihen, int spalten)
void erwarteGroeszeUngleich(int reihen, int spalten)
void erwarteAnzahlKoernerAufKachelGleich(int reihe, int spalte,
                                         int anzahl)
void erwarteAnzahlKoernerAufKachelUngleich(int reihe, int spalte,
                                           int anzahl)
void erwarteKoernerAufKachel(int reihe, int spalte)
void erwarteKeineKoernerAufKachel(int reihe, int spalte)
void erwarteAnzahlKoernerInReiheGleich(int reihe, int anzahl)
void erwarteAnzahlKoernerInReiheUngleich(int reihe, int anzahl)
void erwarteKoernerInReihe(int reihe, int anzahl)
void erwarteKeineKoernerInReihe(int reihe)
void erwarteAnzahlKoernerInSpalteGleich(int spalte, int anzahl)
void erwarteAnzahlKoernerInSpalteUngleich(int spalte, int anzahl)
void erwarteKoernerInSpalte(int spalte)
void erwarteKeineKoernerInSpalte(int spalte)
void erwarteAnzahlKoernerAufTerritoriumGleich(int anzahl)
void erwarteAnzahlKoernerAufTerritoriumUngleich(int anzahl)
void erwarteKoernerAufTerritorium()
void erwarteKeineKoernerAufTerritorium()
void erwarteMauerAufKachel(int reihe, int spalte)
void erwarteKeineMauerAufKachel(int reihe, int spalte)
```

Auch hier sprechen die Namen der Prozeduren für sich. Bei der Prozedur `erwarteTerritoriumGleich` muss der Name eines abgespeicherten Territoriums als Parameter übergeben werden.

4.7.3 Allgemeine Erwartungen

Der TDD-Hamster-Simulator stellt folgende `erwarte…`-Prozeduren zur Verfügung, mit denen Erwartungen an allgemeine Werte bzw. Eigenschaften ausgedrückt werden können:

```
void erwarteGleich(int erwartet, int ist)
void erwarteUngleich(int unerwartet, int ist)
void erwarteWahr(boolean ist)
void erwarteFalsch(boolean ist)
```

Mit der Prozedur erwarteGleich kann bspw. de erwartete Wert einer Variablen überprüft werden: int anzahl = anzahlKoerner(); erwarteGleich(4, anzahl);

4.7.4 Fehlschlag-Erwartung

Weiterhin gibt es noch eine sogenannte Fehlschlag-Erwartungsprozedur

```
void erwarteFehlschlag()
```

Wenn diese in einem Testfall tatsächlich erreicht und ausgeführt wird, führt dies zu einem unmittelbaren Fehlschlagen des Testfalls.

4.7.5 Erwartungsprozeduren mit Fehlermeldungen

Alle oben angeführten Erwartungsprozeduren gibt es noch in einer überladenen Version, bei denen als erster Parameter eine Zeichenkette erwartet wird. Diese Zeichenkette sollte eine aussagekräftige Fehlermeldung sein, da sie beim tatsächlichen Eintreten des entsprechenden Fehlers im TDD-Hamster-Simulator ausgegeben wird. Beispiel:

```
int anzahl = anzahlKoerner();
erwarteGleich("Eigentlich sollten es 4 Körner sein! ", 4, anzahl);
```

5 Beispiele und Videos

Im TDD-Hamster-Simulator ist unter dem Namen „Stufenaufgabe“ ein komplettes Beispiel enthalten, dass alle Testarten integriert. Zur Vorstellung des TDD-Hamster-Simulators wurden Videos angefertigt, die die Benutzung des Simulators und den Einsatz der einzelnen Testarten demonstrieren:

1. Allgemeine Benutzung:
   http://www.java-hamster-modell.de/tdd/1-Benutzung/1-Benutzung.html
2. Territorium-End-Tests:
   http://www.java-hamster-modell.de/tdd/2-EndTests/2-EndTests.html
3. Territorium-Folge-Tests:
   http://www.java-hamster-modell.de/tdd/3-FolgeTests/3-FolgeTests.html
4. Territorium-Checkpoint-Tests:
   http://www.java-hamster-modell.de/tdd/4-CheckpointTests/4-CheckpointTests.html

5. Anwender-Tests:
   http://www.java-hamster-modell.de/tdd/4-AnwenderTests/4-AnwenderTests.html