Testing by Contract mit C4J

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2013

Autoren: Matthias Hauser, Philip Kaufmann
Betreuer: Prof. Hans Rudin

Aufgabenstellung Studienarbeit
für Herrn Matthias Hauser und Herrn Philip Kaufmann

TbC – Testing by Contract mit Contracts for Java (C4J)

1. Studierende

Diese Arbeit wird als Studienarbeit an der Abteilung Informatik durchgeführt von

- Matthias Hauser, mhauser@hsr.ch
- Philip Kaufmann, pkaufman@hsr.ch

2. Betreuer

Bei dieser Arbeit handelt es sich um eine HSR-interne Arbeit zum Aufbau von Know-how und zur Unterstützung der Software Engineering Module.

Auftraggeber/Betreuer:

- Prof. Hans Rudin, HSR, IFS hrudin@hsr.ch +41 55 222 49 36

3. Ausgangslage

Design by Contract (DbC) ist eine Softwareentwurfsmethode, welche klare Verantwortlichkeiten der Klassen schafft, in dem sie für diese Verträge (Contracts) in Form von Preconditions, Postconditions und Class Invariants definiert. Diese Methode wurde von Bertrand Meyer entwickelt [Meyer92] und in seiner Programmiersprache Eiffel findet sich eine direkte Unterstützung. Systematisch angewandt, kann eine Überprüfung der Contracts Bugs zur Laufzeit finden und lokalisieren. Obwohl das Konzept auf breite Zustimmung stiess, hat es sich in der Praxis nicht systematisch durchgesetzt. Verbreitete Programmiersprachen wie Java oder C# hatten ursprünglich keine direkte Unterstützung für Contracts. Verschiedene Versuche eine solche Unterstützung in Form von Bibliotheken zur Verfügung zu stellen hatten keinen Erfolg. In der Praxis hat sich statt dessen Unit Testing mit JUnit und Derivaten für praktisch alle Programmiersprachen durchgesetzt.

In einem Referat am Nordic Workshop on Software Development [Madsen02] prägte Per Madsen den Begriff „Testing By Contracts“, mit dem er Unit Testing und Design By Contract kombinierte.

In der Zwischenzeit steht auch für Java mit *C4J – Contracts for Java* [C4J] eine leistungsfähige und leicht anzuwendende Bibliothek (Eclipse Plugin) zur Verfügung. An der OOP 2013 hat Jonas Bergström, der Entwickler von C4J, eine eindrückliche praktische Anwendung von *Testing by Contract* mit C4J vorgestellt [Bergström13]. Diese Präsentation gab den Anlass für diese Studienarbeit, mit welcher das Thema „Testing by Contract“ aufgearbeitet und für den Unterricht in den Software Engineering Modulen der HSR aufbereitet werden soll.

4. Aufgabenstellung

Das Ziel dieser Arbeit ist eine Aufarbeitung des Themas „Testing by Contract“ für den Unterricht in den Software Engineering Modulen der HSR.

Als konkrete Resultate werden erstens eine Studie und zweitens eine Beispielapplikation mit C4J zu „Testing by Contract“ erwartet:

1. Die **Studie** soll das Thema „Testing by Contract“ darlegen und die Zusammenhänge mit DbC, Unit Testing, Test Driven Development (TDD) eventuell Behavior Driven Development (BDD) beleuchten und den Nutzen für die praktische Anwendung herausarbeiten.
2. Die **Beispielapplikation** soll die Anwendung von „Testing by Contract“ demonstrieren.

Der genaue Inhalt der Studie und der Beispielapplikation wird von den Studierenden und dem Betreuer im Verlaufe der Arbeit gemeinsam festgelegt. Die Beispielapplikation dient auch der Einarbeitung der Studierenden und die dabei gemachten Erfahrungen sollen in die Studie einfliessen. Bei der Beispielapplikation kann es sich um eine bestehende Applikation handeln, die mit C4J Contracts ergänzt wird, oder um eine neu zu entwickelnde Applikation.

Es sind weitere Resultate aus dieser Studienarbeit denkbar, wie eine Skriptbeilage für den Unterricht, eine Übungsaufgabe TbC mit Musterlösung, Evaluation weiterer Werkzeuge, usw. Diese weiteren Resultate werden gegebenfalls zwischen Studierenden und dem Betreuer vereinbart.

5. Zur Literatur und Links

[Bergström13] Jonas Bergström und Hagen Buchwald: *Testing by Contract – eine Fallstudie aus Schweden*
OOP München, Januar 2013
http://www.sigs-datacom.de/oop2013/oop2013.html

[C4J] Homepage: http://c4j-team.github.io/C4J/index.html

[Madsen02] Per Madsen: *Testing by Contract – Combining Unit Testing and Design by Contract*
Proc. Nordic Workshop on Software Development Tools and Techniques,
Copenhagen, August 18-20, 2002
http://www.it-c.dk/people/kasper/NWPER2002/papers/madsen.pdf

[Meyer92] Bertrand Meyer: *Applying „Design by Contract“*
IEEE Computer, October 1992
http://se.ethz.ch/~meyer/publications/computer/contract.pdf

6. Zur Durchführung

Mit dem Betreuer finden wöchentliche Besprechungen statt. Mit dem Auftraggeber finden Besprechungen nach Bedarf statt. Zusätzliche Besprechungen können durch die Studierenden veranlasst werden.
Ausser dem Kick-off Meeting sind alle Besprechungen von den Studierenden mit einer Traktandenliste vorzubereiten, die Besprechung ist durch die Studierenden zu leiten und die Ergebnisse sind in einem Protokoll festzuhalten, das dem Betreuer und dem Auftraggeber per E-Mail zugestellt wird.
Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. An Meilensteinen gemäss Projektplan sind einzelne Arbeitsresultate in vorläufigen Versionen abzugeben. Über die abgegebenen Arbeitsresultate erhalten die Studierenden ein vorläufiges Feedback. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

7. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien der Abteilung Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.
Alle Resultate sind vollständig auf CD/DVD in 3 Exemplaren abzugeben. Der Bericht ist nur auf Wunsch in gedruckter Form abzugeben.

8. Termine

Siehe auch Terminplan auf https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html?&L=0

Montag, den 16.09. 2013	Beginn der Studienarbeit, Ausgabe der Aufgabenstellung durch die Betreuer
16.12.2013	Die Studierenden senden folgende Dokumente der Arbeit per Mail zur Prüfung an ihre Betreuer: - Kurzfassung - A0-Poster Vorlagen sowie eine ausführliche Anleitung betreffend Dokumentation stehen unter den allgemeinen Infos Diplom-, Bachelor- und Studienarbeiten zur Verfügung.
20.12.2013	Die Studierenden senden die vom Betreuer abgenommene und freigegebene Kurzfassung als Word-Dokument an das Studiengangsekretariat (cfurrer(at)hsr.ch).
20. 12.2013	Abgabe des Berichtes an den Betreuer bis 17.00 Uhr.

Allfällige weitere Termine sind am Sekretariat der Abteilung Informatik zu erfragen und sollten entsprechend in einem Sitzungsprotokoll dokumentiert werden.

9. Beurteilung

Eine erfolgreiche Studienarbeit zählt 8 ECTS-Punkte pro Studierenden. Für 1 ECTS Punkt ist eine Arbeitsleistung von ca. 25 bis 30 Stunden budgetiert. Entsprechend sollten ca. 240h Arbeit für die Studienarbeit aufgewendet werden. Dies entspricht ungefähr 17h pro Woche (auf 14 Wochen) und damit ca. 2 Tage Arbeit pro Woche.

Für die Beurteilung ist der HSR-Betreuer verantwortlich.

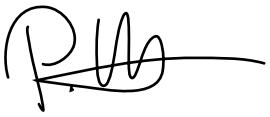
Rapperswil, den 15. September 2013

Prof. Hans Rudin
Institut für Software
Hochschule für Technik Rapperswil

Erklärung

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Ort, Datum	Unterschrift
Rapperswil, 16.12.2013	
	Matthias Hauser
	
	Philip Kaufmann

Abstract

Während Test-Driven Development als agile Entwicklungsmethode der Software Community schon lange kein Fremdwort mehr ist, fristet Design by Contract noch eher ein Schattendasein. Dem deutlich robusteren Code, welcher durch das Konzept der Contracts entsteht, stehen die nicht-agilen Charakteristika von DbC gegenüber.

Ein neuer Ansatz, Testing by Contract, schafft genau hier Abhilfe. Zusammen mit Contracts for Java hat der Softwareentwickler Methoden und Tools zur Hand, die verbessertes Codedesign und erhöhte Codequalität versprechen.

Diese Studienarbeit legt das Thema TbC als Verbindung von TDD und DbC und dessen Nutzen für die praktische Anwendung dar. Dazu wurde eine Beispielapplikation anhand der Testing by Contract-Methode und mithilfe des C4J-Frameworks erstellt. Im Laufe dieser Arbeit stellte sich heraus, dass sich TbC mit C4J als durchdachter und flexibler Ansatz bewährt. Der Mehraufwand wird durch ein feinmaschigeres Sicherheitsnetz wettgemacht.

Management Summary

Ausgangslage

Hohe Codequalität und ein gutes Codedesign stehen gemeinhin als Kriterien für *gute* Software. Sie sorgen für Robustheit und Korrektheit des Programms und Wartbarkeit des Codes. Robustheit steht dabei für die Eigenschaft, auch unter ungünstigen Bedingungen noch zuverlässig zu funktionieren. Robustheit, auch als Fehlertoleranz bezeichnet, zählt zu den Qualitätskriterien für Software [4].

Dem Anspruch hoher Codequalität wird seit längerem mit *Test-Driven Development* (TDD) begegnet. Der grosse Verteil dabei ist, dass Code iterativ und sehr flexibel entwickelt werden kann. Diese Disziplin der agilen Softwareentwicklung hat sich über die Jahre bewährt und findet immer mehr Anklang in der Software Community.

Für ein stabiles Codedesign sorgt unter anderem *Design by Contract* (DbC), ein Konzept, das von Bertrand Meyer mit der Programmiersprache Eiffel eingeführt wurde. Richtig durchgesetzt hat sich diese Vorgehensweise jedoch bis heute nicht.

Mit ein Grund war die fehlende Unterstützung durch ein geeignetes Tool für die Entwickler. 2006 wurde das Framework *Contracts for Java* (C4J) in der ersten, und schliesslich 2012 in der zweiten Generation veröffentlicht. Dazu kam ein Plugin für die integrierte Entwicklungsumgebung eclipse. Vereint mit Anpassungen an der Methodik von DbC ergibt sich so ein neuer, agiler Ansatz für die Softwareentwicklung: *Testing by Contract* (TbC).

Vorgehen, Technologie

Contracts for Java ist, wie der Name schon sagt, ein Framework für Java. Ein Framework ist nach Ralph E. Johnson und Brian Foote eine semi-vollständige Applikation. Es stellt für Applikationen eine wiederverwendbare, gemeinsame Struktur zur Verfügung. Die Entwickler bauen das Framework in ihre eigene Applikation ein, und erweitern es derart, dass es ihren spezifischen Anforderungen entspricht[1]. In Falle von C4J wird eine Struktur zur Verfügung gestellt, mit dem Softwareverträge in den Code eingebunden werden können.

Softwareverträge und damit das Konzept von Design by Contract wurde schon 1985 von Bertrand Meyer eingeführt. Software besteht meist aus vielen Komponenten, welche oft in unterschiedlichsten Umständen wiederverwendet werden. Dass eine solche Komponente nicht wie vorhergesehen gebraucht wird, und entsprechend nicht wie erwartet reagiert, ist absehbar. DbC versucht dies abzuwenden und greift

dazu auf ein allgemein bekanntes Konzept zurück. Jede Interaktion mit einer Komponente soll durch einen Vertrag geregelt werden. Dieser schafft Klarheit darüber, was derjenige, welcher die Komponente benutzt, zu erfüllen hat, damit er das Resultat bekommt, das er erwartet.

Ein Kritikpunkt aus heutiger Sicht ist der fehlende agile Charakter von DbC. Was heisst das? Das Ziel agiler Softwareentwicklung ist es, den Softwareentwicklungsprozess flexibler und schlanker zu machen, als das bei den klassischen Vorgehensmodellen der Fall ist. Agile Softwareentwicklung versucht mit geringem bürokratischen Aufwand, wenigen Regeln und meist einem iterativen Vorgehen auszukommen [2].

Ein gutes Beispiel dafür ist Test-Driven Development. Dabei wird für jede Komponente zuerst ein Test geschrieben, der einen einzigen Testfall abdeckt. Dem Entwickler steht ein Tool zur Verfügung, welches ihm anzeigt, ob die Komponente diesen Test erfüllt. Die Komponente wird nun so angepasst, dass die diesen, und alle, in einer früheren Iteration, schon geschriebenen, Tests erfüllt. Dies geschieht iterativ, während eine Iteration meist nur wenige Minuten dauert. Muss später eine Änderung an der Komponente vorgenommen werden, kann mithilfe der Tests jederzeit gezeigt werden, dass jeder Testfall immer noch erfüllt wird.

Diese zwei Softwareentwurfsmethoden, TDD und DbC, knüpfen beide auf ihre Weise ein Sicherheitsnetz um den Code.

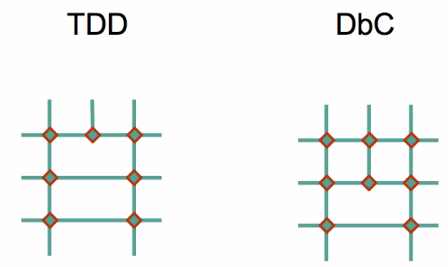


Abbildung 1: Sicherheitsnetz mit TDD und DbC [3]

Wünschenswert wäre natürlich eine Verbindung beider Methoden, welche die Vorteile vereint und zu einem noch feinmaschigeren Sicherheitsnetz führen würde. Mit Testing by Contract wird dies erreicht. Die Komponente wird nach TDD erstellt. Anschliessend werden weitere Tests geschrieben, jeder für einen Fall, bei dem eine Vertragsverletzung vorliegt. Damit dieser Test erfüllt wird, muss nun also auch der Vertrag geschrieben werden. Unterstützt wird der Entwickler dabei unter anderem durch ein Plugin von C4J, welches für die integrierte Entwicklungsumgebung eclipse verfügbar ist.

Ergebnisse

Das Resultat von Testing by Contract ist eine Komponente, welche bei zulässigem Gebrauch auch wie erwartet reagiert und bei nicht zulässigem Gebrauch entsprechend die Vertragsverletzung bemerkt. Man kann also durchaus von einem feinmaschigeren Sicherheitsnetz sprechen.

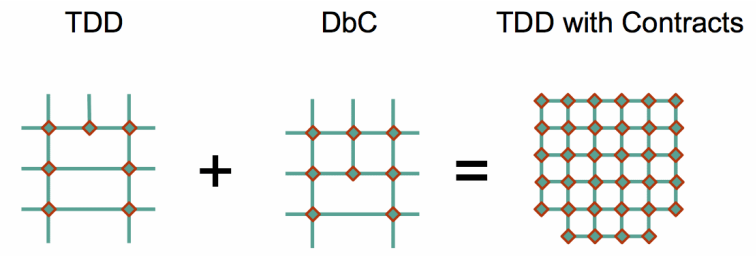


Abbildung 2: Sicherheitsnetz mit TbC

Testing by Contract, wie auch Test-Driven Development, erfordert natürlich einen erhöhten Initialaufwand. Viele Entwickler flüchten sich erfahrungsgemäss auch hinter diese Ausrede und lassen die Tests grosszügig weg. Wenn jedoch stabile, wartbare und robuste Software entwickelt werden soll, sind Tests nicht mehr wegzudenken.

In dieser Studienarbeit wird gezeigt, wie TbC konkret angewendet werden kann. Contracts for Java spielt dabei eine entscheidende Rolle. Ob, und wie gründlich, ein Entwickler ein Tool oder eine Methode gebraucht und anwendet, hängt stark davon ab, ob diese genug einfach und gleichzeitig genug mächtig sind. Diesen Balanceakt meistern sowohl TbC als auch C4J weitestgehend und der Nutzen für die praktische Anwendung ist durchaus gegeben.

Inhaltsverzeichnis

1. Einleitung

1.1 Ausgangslage

Agile Software Entwicklung ist derzeit das grosse Schlagwort in der Software Community. Design by Contract und Testing by Contract haben dabei jedoch noch kaum Beachtung gefunden. Diese Studienarbeit geht der Frage nach, ob man diese Disziplinen überhaupt mit agiler Entwicklung vereinbaren kann und wie weit sie auch im Verbund mit anderen Vorgehensweisen, insbesondere Test Driven Development, funktionieren.

1.1.1 Design by Contract

Design by Contract [5] , kurz DbC, ist ein Programmier- und Softwaredesignkonzept, bei dem es darum geht, einzelne Klassen über Verträge zu definieren. Diese Verträge beschreiben, wie sich die Klasse verhält, sollte von ihr Gebrauch gemacht werden. Diese Spezifikationen sollten formal, präzise und überprüfpar sein. Entwickler von Design by Contract ist Bertrand Meyer, Erfinder der Programmiersprache Eiffel, mit welcher er das Konzept einführte.

1.1.2 Test-driven development

TDD [6] (engl. test-driven development), testgetriebene Entwicklung, ist eine Methode zur agilen Entwicklung von Computerprogrammen. Es wird der Ansatz verfolgt, noch vor dem eigentlichen Schreiben des Produktivcodes Tests zu schreiben, auf welchen anschliessend der Produktivcode aufgebaut wird. Dieses Verfahren wird iterativ umgesetzt, wodurch eine hohe Testabdeckung der Software erzielt wird.

1.1.3 Testing by Contract

Um Design by Contract und die testgetriebene Entwicklung zu vereinen, wurde TbC [7] (engl. testing by contract) entwickelt. Es nimmt von beiden Konzepten einen Teil und integriert sie so, dass keine Grundprinzipien verletzt werden.

1.2 Vorgehen und Technologien

1.2.1 Contracts for Java

Contracts for Java, nachfolgend wird die offizielle Abkürzung C4J [8] gebraucht, ist ein Framework für Java 1.6 und höher. Erklärtes Ziel ist es, den Entwickler in einfacher Art in Design und Testing by Contract zu unterstützen. Mit der 2. Generation und dem Plugin für Eclipse[9] wurde das Framework auch für agile Entwicklungsmethoden ausgelegt.

1.2.2 Anwendungsbeispiel

In dieser Studienarbeit wird Testing by Contract anhand des Spieles Risiko [19] untersucht und angewendet. Die Regeln des Spieles werden hier nur zum Teil erklärt, da sie keinen grossen Einfluss auf die Ergebnisse haben. Das Spiel Risiko wurde deshalb gewählt, da sowohl das Spiel als auch die Spieler verschiedene Zustände haben können. In der Literatur hat sich gezeigt, dass Applikationen, welche das State Pattern[20] verwenden, sehr dankbare Anwendungsfälle für Contracts sind.

1.3 Ergebnisse

Wie in der Aufgabenstellung zu dieser Studienarbeit erwartet, liegen folgende Resultate vor:

1. Implementierung eines Service Layers für das Brettspiel Risiko als Beispielapplikation und Demonstration der Anwendung von TbC.

2. Diskussion und Auswertung der gewonnenen Erkenntnisse. Dabei wird das Thema TbC und dessen Nutzen für die praktische Anwendung dargelegt und die Zusammenhänge mit DbC und TDD beleuchtet.

2. Methoden und Tools

Dieses Kapitel soll dem tieferen Verständnis der zentralen Methoden und der verwendeten Tools dieser Arbeit dienen.

2.1 Design by Contract

Design by Contract [5] ist, wie bereits in der Einleitung (Kapitel 1, Seite 13) erwähnt, ein Ansatz, um Software zu entwerfen. Dieser Ansatz wird mit der Erstellung von Verträgen für Aufrufer und Aufgerufener erreicht. Die nachfolgenden Abschnitte beschreiben, woraus ein Vertrag besteht und was alles durch einen Vertrag abgedeckt wird. Es wird hier immer von Klassen ausgegangen, da auch Subsysteme aus Interfaces bestehen, welche wiederum durch Klassen abgebildet werden.

2.1.1 Der Vertrag

Ein Vertrag in DbC besteht aus drei Teilen, der Vorbedingung (engl. *precondition*), Nachbedingung (engl. *postcondition*) und Invariante (engl. *invariant*).

- **Vorbedingungen** legen fest, welche Bedingungen der Aufrufer erfüllen muss, damit die Methode ein korrektes Resultat liefert.
- **Nachbedingungen** beschreiben die Zusicherungen, welche die Methode gegenüber dem Aufrufenden erfüllt.
- **Invarianten** definieren die Zustände der Klasse, welche stets gelten.

Ein Vertrag umfasst die gesamte Klasse, inklusive Parameter, Variablen, Objektzustände gehaltener oder betroffener Objekte. Werden die Vorbedingungen und Klasseninvarianten vom Aufrufer eingehalten, garantiert der Vertrag, dass die Methode kein unerwartetes Resultat liefert, also die Nachbedingungen erfüllt.

2.1.2 Vor- und Nachbedingungen

Das Prinzip eines Vertrages besteht aus den Rechten und Pflichten, sowohl seitens des Kunden als auch des Anbieters. Die Rechte eines Kunden sind die Pflichten eines Anbieters, wobei umgekehrt die Pflichten eines Kunden die Rechte eines Anbieters sind[10]. Somit genügt es, nur eine Seite, in diesem Fall die des Anbieters, zu definieren.

Bei einer Klasse entsprechen die Methoden den Angeboten. Damit dieses Angebot erfüllt werden kann, sind vom Kunden einige Bedingungen (Pflichten), auf welche der Anbieter ein Recht hat, zu erfüllen. Diese Bedingungen werden nun durch die *Vorbedingungen* geprüft. Erfüllt der Kunde seine Pflichten, garantiert die *Nachbedingung*, dass die Rechte des Kunden auch eingehalten werden. Die *Nachbedingung* prüft also, ob der Anbieter seine Pflichten korrekt erfüllt. Somit legen die Vor- und

Nachbedingungen den Vertrag zwischen Kunde und Anbieter fest. Sobald die Vorbedingung erfüllt ist, muss die Nachbedingung ebenfalls erfüllt werden, ansonsten liegt eine Vertragsverletzung vor.

2.1.3 Invarianten

Invarianten sichern den Zustand aller Instanzen der Klasse. Sie gelten so lange, wie die Instanz existiert. Die Invarianten überprüfen den Zustand der Instanz jeweils vor und nach dem Ausführen einer Methode. Es ist legitim, Invarianten während der Abarbeitung einer Methode zu verletzen, sofern beim Abschluss alle Invarianten wieder eingehalten werden. Invarianten sind somit implizite Vor- und Nachbedingungen.

2.1.4 Vererbung

Das liskovsche Substitutionsprinzip[11] besagt, dass ein Programm, welches mit einer Basisklasse arbeitet, auch mit der davon abgeleiteten Klasse korrekt funktionieren muss. Übertragen auf DbC bedeutet dies, dass bei der Erfüllung von Vorbedingungen der Oberklasse die aufgerufene Methode der Unterklasse auch die Nachbedingung der Oberklasse erfüllen muss. Somit sind Unterklassen nicht frei in der Gestaltung ihrer Vor- und Nachbedingungen. Sie müssen mindestens die Vor- und Nachbedingungen der Oberklasse erfüllen, dürfen die Vorbedingungen nicht verstärken und die Nachbedingungen nicht lockern. Es steht den Unterklassen jedoch frei, ihre eigenen Vorbedingungen abzuschwächen oder die Nachbedingungen stärker einzugrenzen.

2.1.5 Die sechs Prinzipien

In ihrem Buch Design by Contract by Example[5] erklären Richard Mitchell und Jim McKim sechs Prinzipien um gute Verträge zu schreiben. Folgend die Übersetzung der sechs Prinzipien in Anlehnung an die Folien von Hagen Buchwald[12].

1. Trenne **Abfragen** (engl. *queries*) von **Kommandos** (engl. *commands*). Abfragen geben ein Ergebnis zurück, ändern jedoch nicht die sichtbaren Eigenschaften des Objektes. Kommandos können das Objekt verändern, geben jedoch kein Resultat zurück.

2. Trenne **elementare** (engl. *basic*) Abfragen von **abgeleiteten** (engl. *derived*) Abfragen. Abgeleitete Abfragen können durch elementare Abfragen ausgedrückt werden.

3. Erstelle für jede abgeleitete Abfrage eine Nachbedingung, die beschreibt, welches Ergebnis zurückgegeben wird, ausgedrückt durch eine oder mehrere elementare Abfragen. Somit wissen wir, mit Hilfe der elementaren Abfragen, den Wert der abgeleiteten Abfragen.

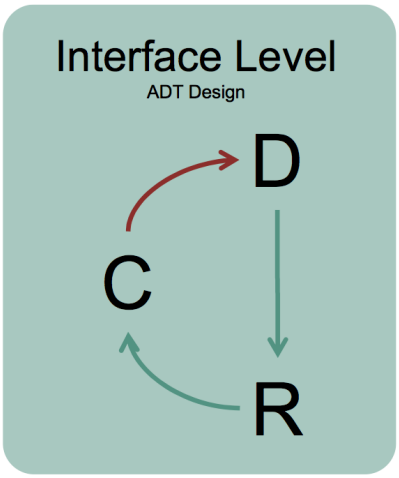
4. Erstelle für jedes Kommando eine Nachbedingung, die den Wert jeder elementaren Abfrage auflistet. Der vollständige sichtbare Effekt eines jeden Kommandos ist nun bekannt.

5. Prüfe für jede Abfrage und Kommando, ob eine Vorbediungung erforderlich ist. Vorbedingungen schränken den Aufrufer ein, wenn er Abfragen oder Kommandos ausführen will.

6. Formuliere Invarianten um die unveränderlichen Eigenschaften des Objekts zu beschreiben. Wähle diejenigen Attribute, die den Anwender unterstützen, das konzeptionelle Modell der Klasse zu verstehen.

2.1.6 DbC-Zyklus

Der DbC-Zyklus ist wie folgt aufgebaut:



- *Declare* (Prinzip 1 & Prinzip 2)
  - Ergänze das Interface um die Deklaration einer neuen Methode
    - Abfrage (@Pure)
      - elementar
      - abgeleitet
    - Kommandos
- *Refactor* (Prinzip 4 & Prinzip 6)
  - Bei Hinzufügen einer elementaren Abfrage
    - Ergänze alle bestehenden Nachbedingungen um einen Ausdruck basierend auf dieser neuen Abfrage
    - Ergänze ggf. die Klasseninvariante
- *Assert* (Prinzip 3 & Prinzip 4 & Prinzip 5)
  - Für die neue Methode:
    - Formuliere die Vorbedingung
    - Formuliere die Nachbedingung

Abbildung 2.1: DbC-Zyklus[13]

Bei Design by Contract wird das Design einer Komponente durch Hinzufügen von neuen Methodendeklarationen entwickelt. Diese können entweder Abfragen oder Kommandos sein.
Eine neu hinzugefügte elementare Abfrage gibt unter anderem Auskunft über den Zustand der Instanz nach dem Aufruf eines Kommandos. Entsprechend müssen die Nachbedingungen der Kommandos ergänzt werden, um die Auswirkung auf die neue Abfrage sichtbar und nachvollziehbar zu machen.

2.1.7 Beispiel am Stack

Der DbC-Zyklus soll nun anhand eines einfachen Beispiels erklärt werden. Hierzu wird ein Stack herangezogen. Der Stack besitzt eine Maximalgrösse, *maxSize* und darf nicht mit *null* befüllt werden. Das Beispiel bedient sich der Syntax von C4J, auf welche in Kapitel 2.4.5 auf Seite 26 genauer eingegangen wird. Es wird davon ausgegangen, dass die Methode *size* (gibt die Anzahl Elemente im Stack zurück) als elementare und die Methode *peek* (gibt das obereste Element des Stacks zurück) als abgeleitete Abfrage bereits wie folgt in C4J implementiert sind:

```
1    //...
2    @Override
3    public int size() {
4        return ignored();
5    }
6
7    @Override
8    public Object peek() {
```

```
9           if (preCondition()) {
10              assert target.size() > 0 : "size must be > 0";
11          }
12          return ignored();
13      }
14      //...
```

Listing 2.1: StackContracts der Methoden size() und peek()

Bei genauer Betrachtung fällt auf, dass die Methode *size* keine Abhängigkeiten besitzt, was erklärt, wieso die Methode eine elementare Abfrage ist. Auf Zeile 10 jedoch wird klar, wieso *peek* eine abgeleitete Abfrage von *size* ist. Die Vorbedingung von *peek* überprüft anhand von der elementaren Abfrage *size*, ob die Grösse grösser als 0 ist. Nun soll der Stack um die Methode *push* mit Hilfe des DbC-Zyklus ergänzt werden.

1. *Declare*: Prinzip 1 besagt, dass Abfragen von Kommandos getrennt werden sollen. Bei der Methode *push* wird ein Element auf den Stack gelegt. Da diese Methode den Stack ändert, handelt es sich hierbei um ein Kommando. Da es sich um ein Kommando handelt, entfällt hier die Durchführung von Prinzip 2.

2. *Refactor*: Da keine neue elementare Abfrage hinzugefügt wurde, müssen keine bestehenden Methoden ergänzt werden. Prinzip 4 entfällt somit für diesen Schritt. Prinzip 6 wird auf Grund der neuen Methode *push* nicht angewendet, die Klasseninvariante bleibt also unverändert.

3. *Assert*: Für die neue Methode werden nun gemäss Prinzip 3, 4 und 5 die Vor- und Nachbedingungen erstellt. Prinzip 3 entfällt, da es sich um ein Kommando handelt. Prinzip 4 und 5 können jedoch wie folgt angewendet werden:

```
1     //...
2     @Override
3     public void push(final Object o) {
4        if (preCondition()) {
5           assert o != null : "object must not be null";
6           assert target.size() < target.maxSize() : "stack must not be
                full";
7        }
8        if (postCondition()) {
9           assert target.size() == old(target.size()) + 1 : "size must
                increment";
10       }
11    }
12    //...
```

Listing 2.2: StackContract der Methode push()

Da *size* die einzige elementare Abfrage ist, muss keine weitere Nachbedingung geprüft werden. Die Vorbedingung verhindert, dass *null* auf den Stack gelegt werden kann und dass beim Aufruf von *push* die Grösse kleiner sein muss, als die Maximalgrösse.

Für die Implementation einer neuen Methode würde nun zu Punkt 1, *Declare*, zurückgesprungen werden.

2.2 TDD

2.2.1 Was ist TDD?

TDD (engl. *test-driven development*), testgetriebene Entwicklung, ist eine Methode zur agilen Entwicklung von Computerprogrammen. Es wird der Ansatz verfolgt, noch vor dem eigentlichen Schreiben des Produktivcodes Tests zu schreiben, auf welchen anschliessend der Produktivcode aufgebaut wird. Das heisst, dass zuerst der Test, danach der Code dieses Teils der Komponente geschrieben wird. Dieses Vorgehen wird iterativ angewendet, was in den TDD-Zyklus im nachfolgenden Abschnitt resultiert.

2.2.2 TDD-Zyklus

Der Ablauf der Entwicklung mit Unit-tests sieht wie folgt aus:

1. Erstelle einen Test für das gewünsche Verhalten oder für einen gefundenen Fehler des Programmes. Dieser Test wird zu Beginn fehlschlagen, da die zugehörige Implementierung nicht vorhanden oder fehlerhaft ist.

2. Implementiere das Verhalten so, dass der Test erfolgreich durchlaufen wird. Wende dazu nur so viel Zeit wie nötig auf. Dabei ist nicht auf sauberen Code zu achten.

3. Schreibe den Code nun so um, dass er lesbar und verständlich wird. Dazu können verschiedene Methoden, wie zum Beispiel Refactoring, zum Zuge kommen. Hierbei ist es wichtig, dass die Tests weiterhin fehlerfrei durchlaufen.

Eine solche Iteration ist sehr kurz, meist nur wenige Minuten. Die oben aufgeführten Schritte werden nun so lange wiederholt, bis die vollständig gewünschte Funktionalität der Komponente (Unit) komplett implementiert ist und keine weiteren Unit-tests mehr sinnvoll erscheinen.

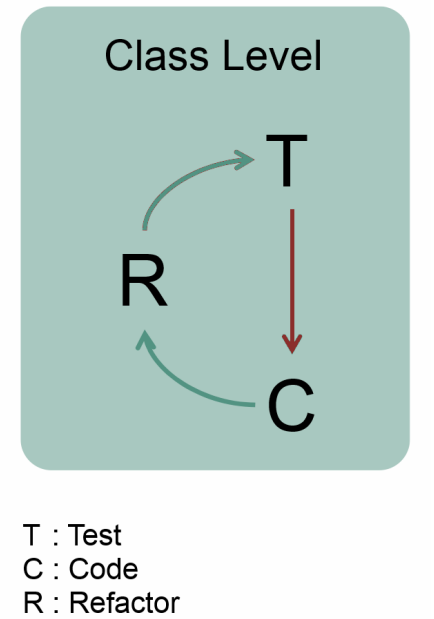


Abbildung 2.2: TDD-Zyklus

2.2.3 Der Stack als Beispiel

Auch hier wird zur Erklärung des TDD-Zyklus wieder der Stack als Beispiel herangezogen, wie dies bereits in Kapitel 2.1.7 auf Seite 17 der Fall war. Auch hier wird wieder davon ausgegangen, dass der Stack eine Maximale Grösse *maxSize* besitzt und *null* nicht auf den Stack gelegt werden kann. Zudem soll ebenfall die Methode *push* implementiert werden. Es hat sich jedoch herausgestellt, dass der Stack eine fehlerhafte Implementierung aufweist und *null* als Wert akzeptiert. Diesen Fehler gilt es nun zu korrigieren. Die Methode *push* ist zur Zeit wie folgt implementiert:

```
1   //...
2   @Override
3   public void push(final Object o) {
4      data[size] = o;
5      size++;
6   }
7   //...
```

Listing 2.3: Implementation der Methode push() vor dem Test

1. Da überprüft werden muss, ob der Stack *null* erlaubt oder nicht, muss ein entsprechender Test dazu geschrieben werden. Dieser sieht folgendermassen aus:

```
1    //...
2    @Test
3    public void nullTest() {
4       IStack stack = new Stack(10);
5       final int oldSize = stack.size();
6       stack.push(null);
7
8       assertEquals(stack.size(), oldSize);
9    }
10   //...
```

Listing 2.4: nullTest für Stack

   Zuerst wird der Stack mit der Maximalgrösse von 10 initialisiert. Anschliessend die alte Grösse des Stacks in *oldSize* gespeichert, um diese nach dem Hinzufügen von *null* Schluss mittels *assertEquals(stack.size(), oldSize);* zu überprüfen. Führt man den Testcode nun aus, wird der Test fehlschlagen, da sich die Grösse des Stacks um 1 ändern wird.

2. Damit der Test erfolgreich durchläuft, muss nun die Methode *push* aus Listing 2.3 ergänzt werden. Dies könnte dann so aussehen:

```
1    //...
2    @Override
3    public void push(final Object o) {
4       if(o == null) {
5          return;
6       }
7       data[size] = o;
8       size++;
9    }
10   //...
```

Listing 2.5: Implementation der Methode push() nach dem Test

Führt man den Test erneut aus, läuft er grün, das gewollte Verhalten wurde demnach erfolgreich implementiert.

3. Der letzte Schritt entfällt, da der Code bereits verständlich und lesbar ist und es somit keinem Refactoring bedarf.

Falls nun eine weitere gewünschte Funktionalität implementiert werden sollte, würde man zu Schritt 1 wechseln und den Zyklus erneut durchlaufen.

2.3 TbC

2.3.1 Was ist TbC?

Testing by Contract ist ein neuer Ansatz der agilen Softwareentwicklung. Er versucht, Design by Contract (DbC) und Test-driven development (TDD) zu vereinen. Beide Ansätze haben einen eigenen Ablauf. Während TDD schrittweise in kleinen Iterationen arbeitet (Kapitel 2.2.2, Seite 19), erstellt man bei DbC (Kapitel2.1.6, Seite 17) direkt den gesamten Vertrag einer Klasse, ohne ihn mit Tests zu überprüfen.

2.3.2 TbC-Zyklus

Um die zwei Ansätze zu vereinen muss lediglich die Vorgehensweise angepasst werden. Folgender Ansatz wird von Hagen Buchwald [14] verfolgt (Anmerkungen in Blockschrift von den Autoren hinzugefügt):

TDD-Zyklus (Arbeiten auf der Klassen-Ebene)

1. *Erstelle einen positiven Test für die Zielklasse.*
   Bei positiven Tests wird davon ausgegangen, dass alle Aufrufe an die Klasse erlaubt sind und somit zu keinem Fehlverhalten führen.
2. *Der Test scheitert.*
3. *Überarbeite die Zielklasse so, dass der neue Test grün wird.*
4. *Führe ein Refactoring (Clean Code) auf der Zielklasse und der Testklasse aus.*
5. *Wenn es weitere, redundanzfreie positive Tests gibt: Gehe zurück zu Schritt 1.*

Vorbereiten des DbC-Zyklus

6. *Extrahiere das Interface aus der Zielklasse und passe die Zielklasse so an, dass sie dieses Interface implementiert.*
7. *Erstelle eine Vertragsklasse für das Ziel-Interface.*
   In C4J ist es auch möglich, den Vertrag von der Zielklasse erben zu lassen, statt das gemeinsame Interface zu implementieren.

DbC-Zyklus (Arbeiten auf der Typen-Ebene)

8. *Erstelle einen negativen Test für die Zielklasse.*
   Negative Tests sind diejenigen, welche ohne Contracts ein fehlerhaftes oder unerwartetes Verhalten bei der Klasse hervorrufen. Diese Tests müssen somit eine Assertion Exception (in Java) erwarten.
9. *Der Test scheitert.*

10. *Überarbeite die Pre-Conditions der Vertragsklasse des Ziel-Interfaces so, dass der neue Test grün wird.*

11. *Führe ein Refactoring (Clean Code) auf der Vertragsklasse und der Testklasse aus.*

12. *Wenn es weitere, redundanzfreie negative Tests gibt: Gehe zurück zu Schritt 8.*

Abschliessen des DbC-Zyklus

13. *Führe ein abschliessendes Refactoring (Clean Contract) auf der Vertragsklasse aus, insbesondere:*

    - *Clean Contract Prinzip 4: Erstelle als Verallgemeinerung der Aussagen der positiven Tests die Beschreibung der Veränderung des sichtbaren Zustands jeder Methode des Ziel-Interfaces in Form von Post-Conditions, in der über jede elementare Abfrage des Ziel- Interfaces via assert-Statement eine Aussage getroffen wird.*
    - *Clean Contract Prinzip 6: Vermeide Redundanz durch Klassen-Invarianten.*

Wie man aus dem Vorgehen herauslesen kann, werden zuerst nur positive Unittests gemacht, für alle negativen Tests sind die Vertragsklassen zuständig. Durch diese Regelung wird das DRY- (don't repeat yourself) Prinzip nicht verletzt und man schreibt auch keine Tests doppelt.

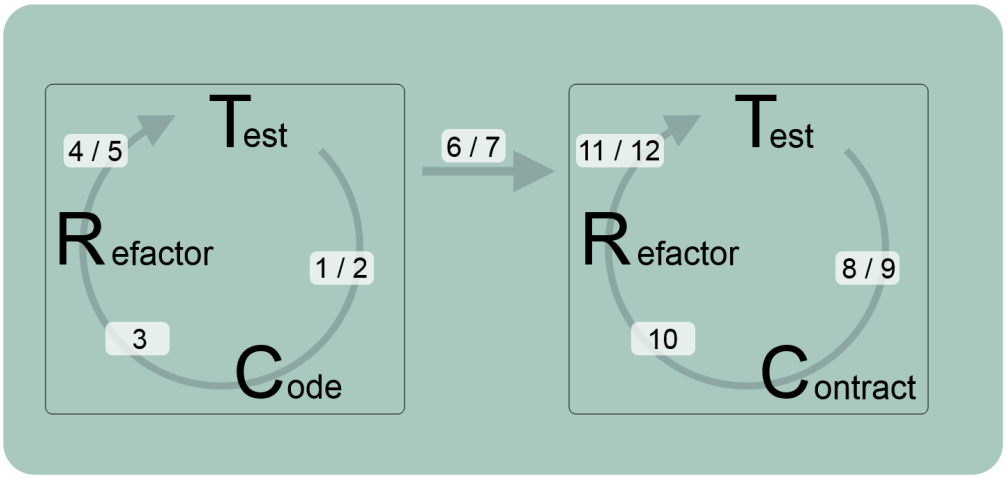


Abbildung 2.3: TbC-Zyklus

2.3.3 Der TbC-Zyklus am Stack

Wie in den vorhergehenden Kapiteln 2.1.7 und 2.2.3 wird auch hier der Zyklus wieder anhand eines einfachen Stacks vorgestellt. Hier wird nun jedoch davon ausgegangen, dass nur noch der Test *putObjectOnStack* getestet und die Methode *push* implementiert werden muss. Alle übrigen, positiven Tests wurden bereits geschrieben, ebenso die dazugehörige Implementation. Der Stack besitzt erneut eine maximale Grösse *maxSize* und nimmt keine *null*-Argumente entgegen. Zuerst beginnt der TDD-Zyklus:

1. Schritt 1 schreibt vor, einen positiven Test für die Zielklasse zu erstellen:

```
1    //...
2    @Test
3    public void testPush() {
4       Stack stack = new Stack(10);
5       final int oldSize = stack.size();
6
7       final Object o = new Object();
8       stack.push(o);
9
10      assertEquals(stack.size(), oldSize + 1);
11   }
12   //...
```

Listing 2.6: Methode testPush() für den Stack

   Da wir ein gültiges Objekt, *Object o*, auf den Stack legen wollen, ist dies ein positiver Test.

2. Da die Implementierung von *push* auf dem Stack noch leer ist, wird dieser Test schief laufen.

3. Die überarbeitete Zielklasse sieht mit der neuen Methode wie folgt aus:

```
1    public void push(final Object o) {
2       data[size] = o;
3       size++;
4    }
```

Listing 2.7: Implementierung der Methode push im Stack

   Der Test wird nun erfolgreich durchlaufen.

4. Dieser Schritt entfällt, da der Code bereits gut lesbar geschrieben wurde.

5. Da es, wie in der Einleitung zu diesem Kapitel erwähnt, keine weiteren, positiven Tests mehr gibt, wird hier mit Schritt 6 weitergefahren.

6. Schritt 6 und 7 werden hier zu einem zusammengefasst. Anstatt ein Interface zu kreieren, wird von der Stack-Klasse geerbt:

7.

```
1  import static de.vksi.c4j.Condition.ignored;
2  import static de.vksi.c4j.Condition.old;
3  import static de.vksi.c4j.Condition.postCondition;
4  import static de.vksi.c4j.Condition.preCondition;
5  import de.vksi.c4j.ClassInvariant;
6  import de.vksi.c4j.Target;
7
8  public class StackContract extends Stack {
9
10    @Target
11    private Stack target;
12    //...
```

Listing 2.8: StackContract.java

8. Die nächste Aufgabe besteht darin, einen negativen Test für die Zielklasse zu erstellen. Unser Stack darf keine *null*-Elemente auf den Stack legen. Dies bietet sich für einen negativen Test an:

```
1   @Test(expected = AssertionError.class)
2   public void nullTest() {
3      Stack stack = new Stack(10);
4      final int oldSize = stack.size();
5      stack.push(null);
6   }
```

Listing 2.9: nullTest für StackContract

Hierbei gilt zu beachten, dass beim Test eine Expection der Klasse *AssertionError.class* erwartet wird, da C4J mittels *assert* die Vor- und Nachbedingungen prüft. Die Überprüfung findet somit nicht mehr mittels *assertEquals* wie in Listing 2.6 statt. Wird der Test nun durchgeführt, schlägt er fehl.

9. Damit der Test erfolgreich ausgeführt werden kann, müssen die Vorbedingungen der Vertragsklasse für die Methode *push* angepasst werden:

```
1    //...
2   @Override
3   public void push(final Object o) {
4      if (preCondition()) {
5         assert o != null : "object must not be null";
6      }
7   }
8   //...
```

Listing 2.10: Vorbedingung für die Methode push

Wird der Test nun erneut durchgeführt, wird der von der Testklasse geforderte *AssertionError* geworfen und war erfolgreich.

10. Auch hier kann Schritt 11 ausgelassen werden, da der Code bereits leserlich geschrieben wurde.

11. In der Annahme, dass alle weiteren, negativen Tests bereits alle durchgeführt wurden, wird zu Schritt 13 übergegangen. Ansonsten müssten die Schritte 8-11 erneut durchgeführt werden.

12. Als letzten Schritt müssen nun die Nachbedingungen für die Methode *push* definiert werden. Diese werden mit Hilfe der elementaren Abfragen erstellt. Da die elementare Abfrage *size* bereits existiert, wird diese in die Nachbedingung eingebaut.

```
1    //...
2   @Override
3   public void push(final Object o) {
4      if (preCondition()) {
5         assert o != null : "object must not be null";
6         assert target.size() < target.maxSize() : "stack must not be
              full";
7      }
8      if (postCondition()) {
9         assert target.size() == old(target.size()) + 1 : "size must
              increment";
```

```
10        }
11    }
12    //...
```

Listing 2.11: Nachbedingung für die Methode push

Die Vorbedingung auf Zeile 6 wurde hier der Vollständigkeit halber eingefügt. Diese wurde natürlich ebenfalls im Laufe des TbC-Zyklus ermittelt. Die Nachbedingung stellt nun anhand der elementaren Abfrage *size* sicher, dass sich die Grösse des Stacks nach dem Aufruf der Methode *push* um eins vergrössert hat. Klasseninvarianten und weitere Nachbedingungen müssen nicht implementiert werden. Unser TbC-Zyklus ist somit zu Ende.

In Listing 2.11 ist gut zu sehen, dass die *null*- und Kapazitätsüberprüfung in der Vertragsklasse erfolgen. Auf diese kann somit im produktiven Code verzichtet werden, was den Code überschaubarer und performanter macht.

2.4 C4J

Damit der Entwickler die zusätzliche Zeit und Energie in Codedesign und -Qualität investiert, müssen die Werkzeuge gleichzeitig mächtig und einfach sein. Mit diesem Hintergedanken wurde das Javaframework C4J vom schwedischen Entwickler Jonas Bergström entwickelt.

2.4.1 Geschichte

Erste Generation

Die erste Generation von C4J wurde von Bergström veröffentlicht. Er hatte es für seinen früheren Arbeitgeber entwickelt und nutzte es in seinem Team über mehrere Jahre hinweg. 2006 merkte Bergström dann, dass mit der in Java 1.5 neu eingeführten *javaagent* Option die Funktionsweise von C4J beträchtlich verbessert werden konnte. Da sonst kein vergleichbares Produkt verfügbar war, entschied er sich, das Framework auf sourceforge in der Version 1.0 freizugeben. Bis zur Version 2.7.5 lag die Entwicklung bei Bergström.

Zweite Generation

Im Herbst 2011 begann ein Team rund um Ben Romberg und Hagen Buchwald mit der Entwicklung einer zweiten Generation des Contracts Frameworks. Dieses sollte die nicht-agilen Charakteristika der vorherigen Versionen ausmerzen. Im Oktober 2012 wurde die finale Version schliesslich vom Verein der Karlsruher Software-Ingenieure veröffentlicht.[15]

2.4.2 Umsetzung von DbC

Die Verträge werden von der Zielklasse, bzw. dem Zielinterface, separiert erstellt. Dies ergibt gleich mehrere Vorteile:

- Der Produktivcode bleibt übersichtlich und beschränkt sich auf die tatsächliche Implementierung.
- Contracts können nachträglich hinzugefügt werden. Dies ist besonders wichtig bei Legacysystemen.
- Die Überprüfung der Verträge zur Laufzeit kann bequem zu- oder abgeschaltet werden.

2.4.3 Hinter den Kulissen

C4J beruht auf Java Bytecode Instrumentation (BCI) und benutzt dazu die Javassist library von JBoss. Mit dem in Java 1.5 eingeführten Hook der javaagent VM-Option kann zur Laufzeit jede geladene Klasse beliebig durch *bytecode injection* angepasst werden. Der C4J instrumentator geht dabei für jede geladene Klasse wie folgt vor:[16]

- Hat die Klasse oder eine ihrer Elternklassen eine gültige *ContractReference* Annotation? Oder ist eine externe Contract Klasse dieser Klasse oder einer der Elternklassen angebunden? Falls keiner der Fälle eintritt, wird diese Klasse nicht weiter beachtet.
- Falls die Klasse einen *direkten* Contract (im Gegensatz zu einem geerbten) besitzt, füge ihr eine private Membervariable der Vertragsklasse hinzu.
- Prüfe dann für jede Methode:

  existiert eine Vorbedingung im Contract (oder dem einer Elternklasse): füge einen Aufruf darauf am Beginn der Methode an.

  existiert eine Nachbedingung im Contract (oder dem einer Elternklasse): füge einen Aufruf darauf am Ende der Methode an.

  existiert eine Klasseninvariante im Contract (oder dem einer Elternklasse): füge einen Aufruf darauf am Ende der Methode an.
- Falls sowohl ein direkter als auch ein geerbter Contract vorhanden ist: füge Aufrufe darauf gemäss den Vererbungsregeln für Contracts an.

2.4.4 Umsetzung der Vererbungsregeln

C4J setzt das liskovsche Substitutionsprinzip anders um, als dies Mitchell und McKim in Design by Contract by Example[5] beschreiben:

1. Im Falle von Mehrfachvererbung (in Java mithilfe von Interfaces) prüft C4J die vorhandenen Preconditions durch eine UND-Verknüpfung im Gegensatz zu Eiffel, welches die ODER-Verknüpfung benutzt.
2. Existiert für eine geerbte Methode eine Precondition im Contract einer Elternklasse, darf dafür keine eigene Vorbedingung definiert werden.

Eine ausführliche Diskussion zu diesen Themen findet sich auf dem sourceforge Forum von C4J[17].

2.4.5 C4J Syntax

Contracts for Java kommt mit einigen syntaktischen Besonderheiten daher, die es bei der Entwicklung zu beachten gilt. Wie bereits erwähnt, sind die Contracts (die *Vertragsklassen*) vom Produktivcode (die *Zielklassen*) getrennt. Dabei besteht die Möglichkeit, ein gemeinsames Interface zu implementieren, oder die Vertragsklasse von der Zielklasse erben zu lassen.
Das Framework arbeitet mit einigen Annotationen und statischen Methoden, die in den folgenden zwei Abschnitten kurz beschrieben werden. Als Beispiel soll eine Klasse *Zielklasse.java* durch Contracts in der Klasse *Vertragsklasse.java* geschützt werden.

Annotationen

Aus der Zielklasse wurde folgendes Interface extrahiert:

```
1  package contract;
2
3  import de.vksi.c4j.ContractReference;
4  import de.vksi.c4j.Pure;
5
6  @ContractReference(Vertragsklasse.class)
7  public interface ExtractedInterface {
8
9     public void foo(int i);
10
11    @Pure
12    public int bar();
13 }
```

Listing 2.12: ExtractedInterface.java

2 Annotationen sind hier sichtbar:

- *@ContractReference*: Damit wird die Vertragsklasse referenziert. Anstelle davon könnte die Vertragsklasse(Listing 2.14) direkt annotiert werden. In diesem Fall würden die Klassendeklarationen so aussehen:

```
1  @Contract (forTarget = Zielklasse.class)
2  public class Vertragsklasse {
3     // ...
4  }
5
6  public class Zielklasse {
7     // ...
8  }
```

Listing 2.13: alternative Klassenannotation

- *@Pure*: Damit wird eine Methode deklariert, welche den Zustand der Instanz nicht ändern darf. Allgemein gesprochen sind das Queries (s. dazu Kapitel 2.1.5).

Die Vertragsklasse offenbart noch zwei weitere Annotationen:

```
1  package contract;
2
3  import static de.vksi.c4j.Condition.ignored;
4  import static de.vksi.c4j.Condition.postCondition;
5  import static de.vksi.c4j.Condition.preCondition;
6  import static de.vksi.c4j.Condition.old;
7  import static de.vksi.c4j.Condition.result;
8  import de.vksi.c4j.ClassInvariant;
9  import de.vksi.c4j.Target;
10
11 public class Vertragsklasse implements ExtractedInterface {
12
13    @Target
14    private ExtractedInterface target;
15
16    @ClassInvariant
```

```
17    public void classInvariant() {
18       assert target.bar() > 0;
19    }
20
21    @Override
22    public void foo(int i) {
23       if (preCondition()) {
24          assert i > 0 : "Keine Negativwerte erlaubt";
25          assert i != old(target.bar()) : i + " war schon gesetzt";
26       }
27       if (postCondition()) {
28          assert target.bar() == i;
29       }
30    }
31
32    @Override
33    public int bar() {
34       if (postCondition()) {
35          int result = result();
36          assert result > 0;
37       }
38       return ignored();
39    }
40
41 }
```

Listing 2.14: Vertragsklasse.java

- *@Target*: Dies markiert eine Instanzvariable des zu schützenden Objekts.
- *@ClassInvariant*: Markierung einer Methode, welche für die Überprüfung der Klasseninvariante zuständig ist.

Die Zielklasse sieht folgendermassen aus:

```
1  package contract;
2
3  public class Zielklasse implements ExtractedInterface {
4
5     private int myInt;
6
7     @Override
8     public void foo(int i) {
9        myInt = i;
10    }
11
12    @Override
13    public int bar() {
14       return myInt;
15    }
16
17 }
```

Listing 2.15: Zielklasse.java

Bemerkenswert ist hierbei, dass von C4J keine Spur vorhanden ist.

Statische Methoden

In der Vertragsklasse (Listing 2.14) werden 5 Methoden von C4J gebraucht[18]:

- *preCondition()*: Vorbedingung einer Methode
- *postCondition()*: Nachbedingung einer Methode
- *result()*: Rückgabewert der Methode (nur in Post-Conditions verfügbar)
- *ignored()*: Liefert einen Dummy-Rückgabewert und wird benötigt, wenn die zu schützende Methode in der zu schützenden Klasse (target class) einen Rückgabewert besitzt (Query)
- *old()*: Abfragen des Zustands eines Objekts vor der Methodendurchführung (nur in Post-Conditions verfgbar)

3. Ergebnisse

In diesem Kapitel wird erläutert, wie versucht wurde, *Testing by Contract* (Kapitel 2.3, Seite 21) mit Hilfe des Tools *C4J* (Kapitel 2.4, Seite 25) und dem Anwendungsbeispiel Risiko[19] umzusetzen und welche Fallstricke sich dabei in den Weg legten. Weiter ist zu beachten, dass das Spiel ohne Benutzeroberfläche umgesetzt wurde. Lediglich die Logik fand Einzug in die Arbeit.

Risiko ist ein Brettspiel. Auf dem Brett ist die Weltkarte vorzufinden, welche in Kontinente und deren Länder unterteilt ist. Zu Beginn des Spieles kriegt jeder Spieler eine Mission zugeteilt, die es zu erfüllen gilt. Dies kann zum Beispiel die Eroberung von 24 Ländern sein. Jeder Spieler kriegt zudem Truppen zur Verfügung gestellt, mit denen er seine eigenen Länder verteidigen oder den Gegner angreifen kann. Die Regeln des Spieles werden hier nur zum Teil erklärt, da sie keinen grossen Einfluss auf die Ergebnisse haben. Das Spiel Risiko wurde deshalb gewählt, da sowohl das Spiel als auch die Spieler verschiedene Zustände haben können. In der Literatur hat sich gezeigt, dass Applikationen, welche das State Pattern[20] verwenden, sehr dankbare Anwendungsfälle für Contracts sind.

Zu Beginn der Arbeit wurden Architekturdiagramme erstellt, welche stetig an den Stand der Arbeiten angepasst wurden. Um sich einen besseren Überblick der geschriebenen Library machen zu können, werden zuerst einige dieser Architekturdiagramme vorgestellt und erklärt. Somit können Designentscheide nachvollzogen und die erklärten Beispiele besser verstanden werden.

3.1 Klassendiagramm

Das Klassendiagramm zeigt vor allem die Schnittstelle unseres Service-Layers und wie die dahinterliegende Implementation aussieht. Es wurden lediglich die wichtigsten Klassen dargestellt. Das vollständige Diagramm würde die Darstellung nur unübersichtlich gestalten und für Verwirrung sorgen. Das in der Einleitung erwähnte State-Pattern [20] wird zudem in den Zustandsdiagrammen im nachfolgenden Kapitel behandelt.

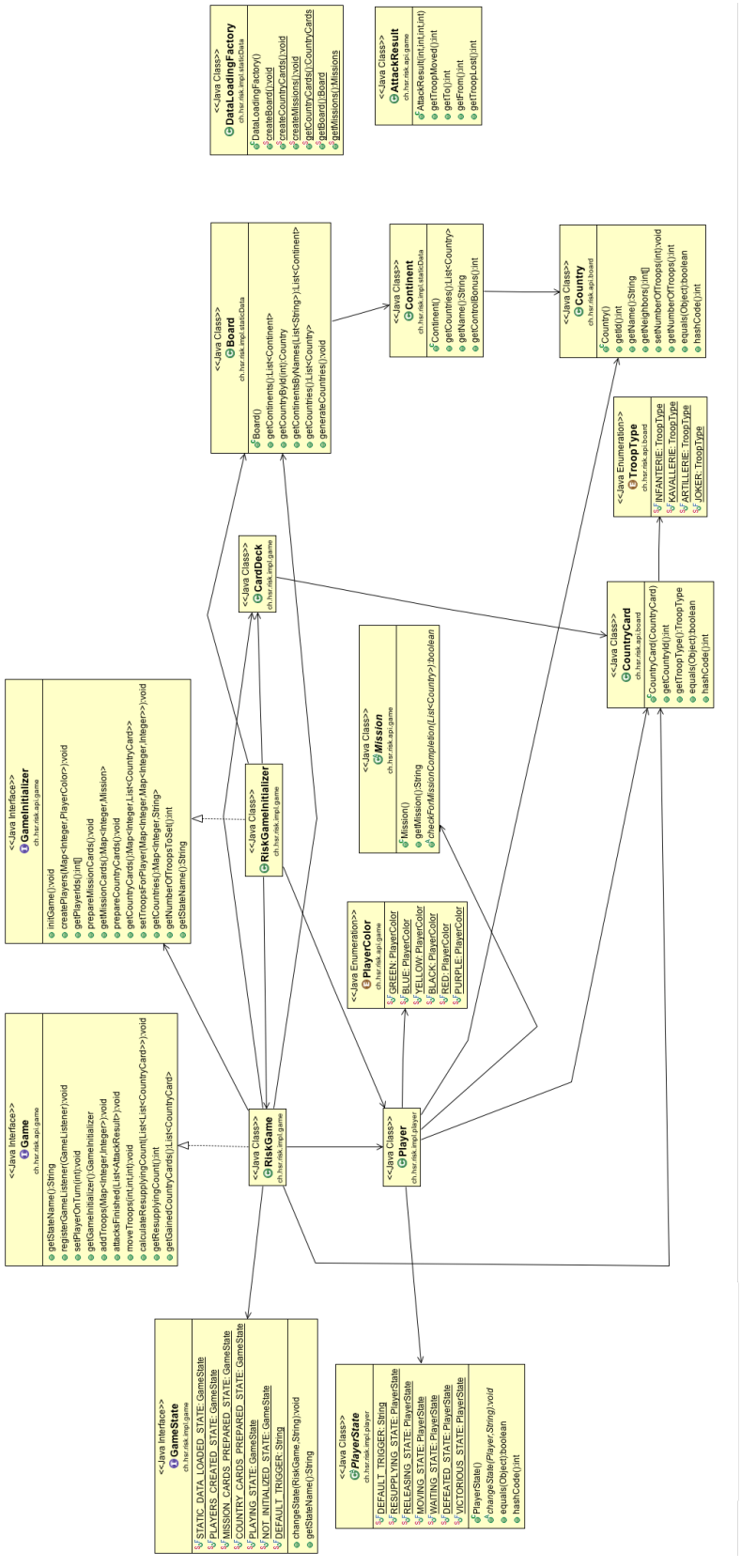


Abbildung 3.1: Klassendiagramm Risk-Library

Der Hintergedanke bei der Entwicklung der Library war, so wenig Informationen wie möglich dem aufrufenden Layer preiszugeben. Deshalb werden auch nur zwei Schnittstellen zur Verfügung gestellt, *GameInitializer*, welcher in Kapitel 3.5 genauer vorgestellt wird, und *Game*, welches die Spiellogik beinhaltet. Der Grund für die Trennung war die Abbildung der Spielregeln, welche sich im Verlaufe der Entwicklung als sehr komplex herausstellten. Zudem wurde so eine bessere *separation of concerns*[21] erreicht. Die Implementation von *Game* ist in *RiskGame* realisiert, welches seinerseits mehrere Stati, *GameState*, enthält. Diese sind gemäss dem State Pattern [20] implementiert, wobei sie vor allem für die Initialisierung des Spieles von Bedeutung sind. Während des Spielens sind die *PlayerStates* des Players interessant. Damit das Spiel überhaupt gespielt werden kann, müssen *Player* erstellt werden können, die jeweils eine andere *PlayerColor* besitzen. Jeder Spieler verfügt zudem über eine Mission, welche er im Verlaufe des Spieles erfüllen muss, um das Spiel zu gewinnen. Um die Missions- und Länderkarten an Spieler verteilen zu können, greift das *RiskGame* auf die Klasse *CardDeck* zu, welche die Länderkarten, *CountryCard*, verwaltet. Jede Länderkarte besitzt einen speziellen *TroopType*, die im Spielverlauf eine Rolle spielen. Das *Board* dient dazu, die Kontinente (*Continent*) zu verwalten, welche ihrerseits Länder (*Country*) enthalten. Ein Spieler kann mehrere dieser Länder besitzen, was unter anderem für die Erfüllung seiner Mission von Bedeutung sein kann. Damit diese ganze Infrastruktur aufgebaut werden kann, lädt die *DataLoadingFactory* die entsprechenden Konfigurationen aus XML-Dateien in das Spiel. Über diese Konfigurationsdateien können somit weitere Farben, Länder, Kontinente und Missionen definiert werden. Da es in dem Spiel auch um Ländereroberungen geht, muss es möglich sein, einen Spieler anzugreifen. Das Resultat dieses Angriffes teilt der Clientlayer dem *RiskGame* über ein *AttackResult* mit.

3.2 Zustandsdiagramme

Da sowohl das Spiel Risiko als auch die Spieler, wie bereits in der Einleitung (Seite 13) erwähnt, verschiedene Zustände haben können, wurden dafür Zustandsdiagramme entwickelt.
Wie in Abbildung 3.2 ersichtlich ist, besitzt das Spiel vor allem in der initialen Phase verschiedene Zustände, wobei es am Schluss in den Zustand *playing* übergeht und dort bleibt, bis das Spiel beendet wird.

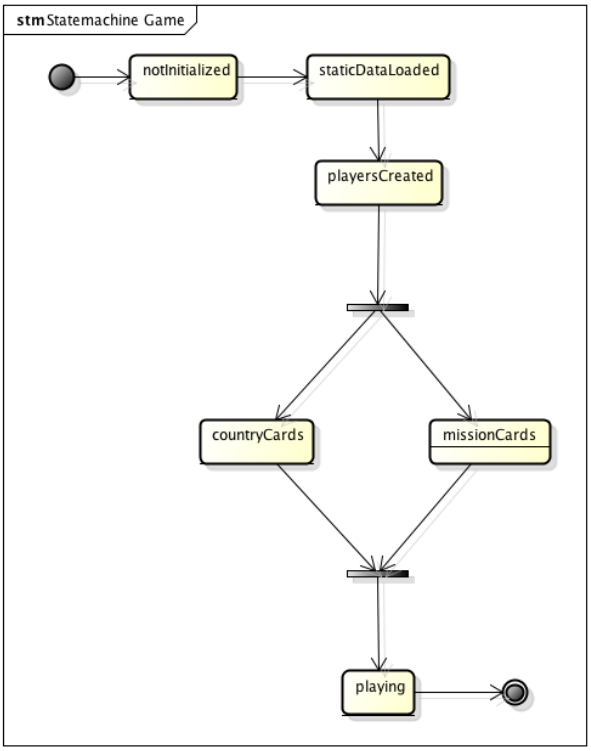


Abbildung 3.2: Zustandsdiagramm Game

Etwas interessanter ist das Zustandsdiagramm des Spielers in Abbildung 3.3. Der Spieler befindet sich in einem Loop, nämlich über die Zustände emphwaiting, emphresupplying troops, emphreleasing country und emphmoving troops. Wie die Namen bereits vermuten lassen, sind hier die verschiedenen Möglichkeiten aufgelistet, die der Spieler hat, wenn er am Zug ist.

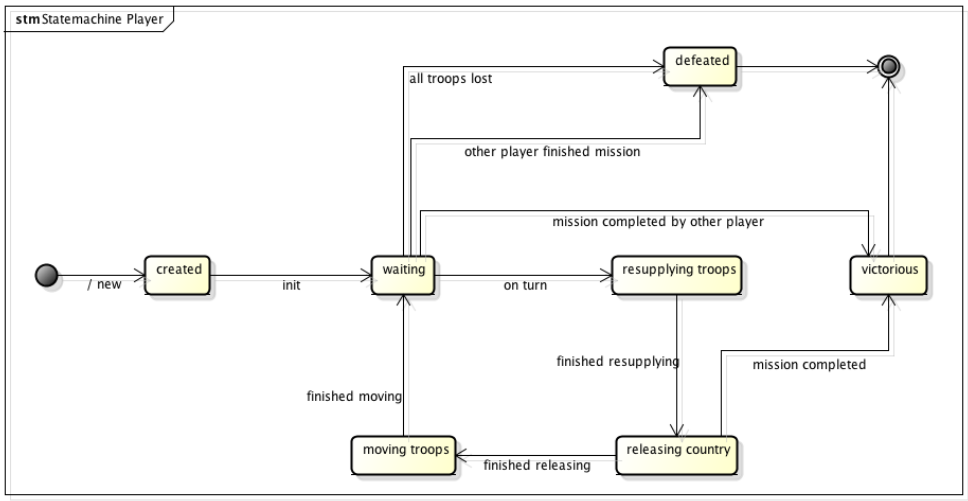


Abbildung 3.3: Zustandsdiagramm Spieler

3.3 Sequenz Diagramme

Um die Zustandsübergänge und Aufrufe aus der Benutzeroberfläche abzuarbeiten, wurde die Logik in zwei Teile gegliedert. Der erste Teil, game init genannt, ist für die Initialisierung des Spieles verantwortlich. Darunter fallen unter anderem die Erstellung der Weltkarte, das Verteilen von Länderkarten und das Setzen der Truppen des Spielers. In Abbildung 3.4 ist die Initalisierung aufgeführt.

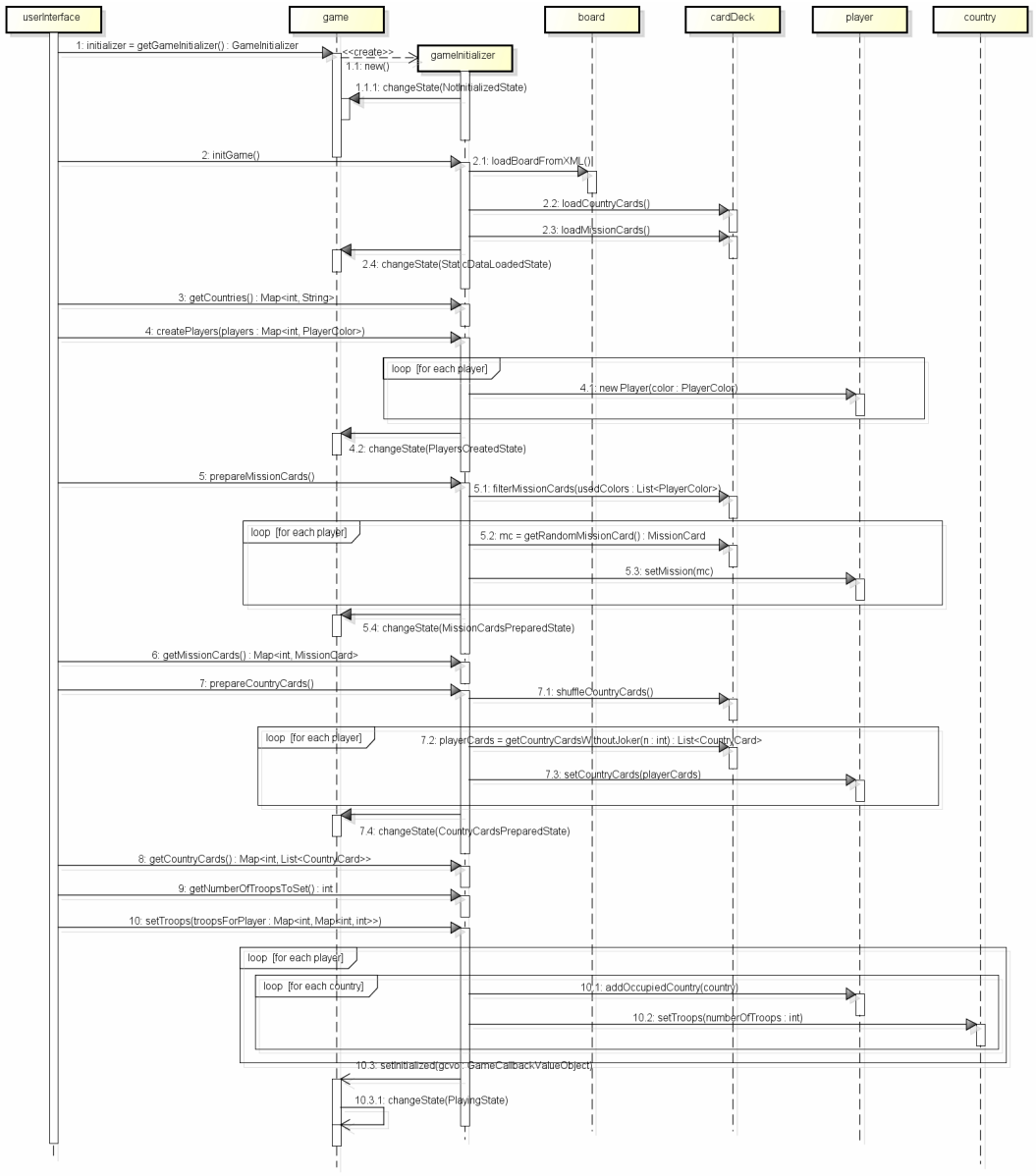


Abbildung 3.4: Sequenzdiagramm game init

GamePlay, der zweite Teil der Logik, besitzt folgendes Sequenzdiagram

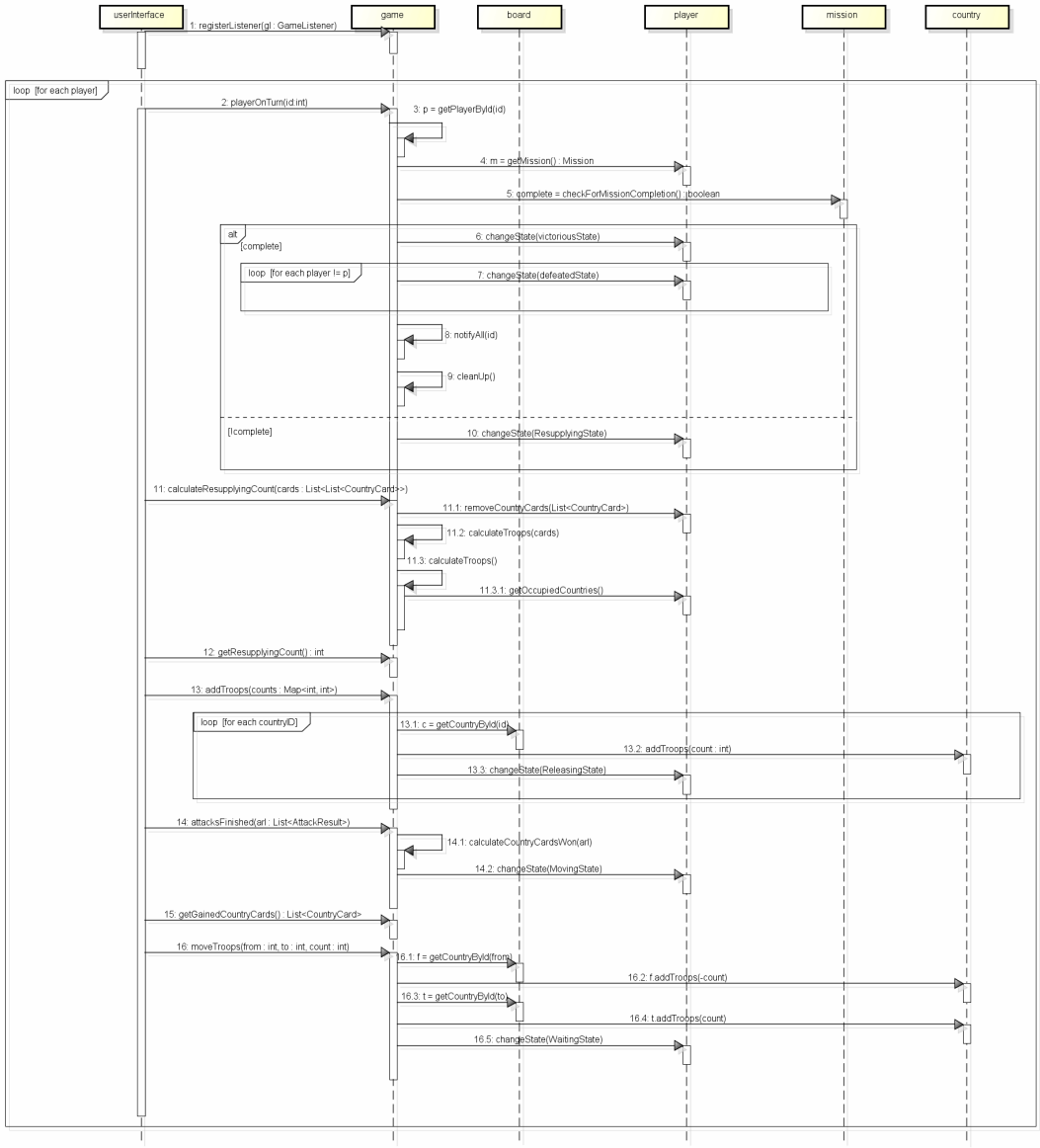


Abbildung 3.5: Sequenzdiagramm game play

Hier ist die ganze Spiellogik während des Spiele inbegriffen, das heisst die Einhaltung der Regeln des Spieles werden beim Aufruf aus einem darüberliegenden Layer, z.B. der Benutzeroberfläche, geprüft.

3.4 TbC am Beispiel Player

Anschliessend zur Architektur wurde der Code entwickelt. Hier wurde so vorgegangen, dass zuerst einfache Datenklassen wie Spieler oder die Mission erstellt wurden. Dies hat den Hintergedanken, dass auf diese Klassen während des Spielverlaufes oft zurückgegriffen wird und sie somit die Grundlage für das Spiel darstellen.

Der TbC-Zyklus (Kapitel 2.3.2) schreibt vor, zuerst den TDD Zyklus abzuarbeiten, negative Tests jedoch nicht miteinzubeziehen:

```
1  public class PlayerTest {
2     Player player;
3
4     @Before
5     public void setUp() throws Exception {
6        player = new Player(PlayerColor.GREEN, 1);
7     }
8
9     @Test
10    public void addMissionCardTest() {
11       player.setMission(new DefeatPlayerMission());
12       assertNotNull(player.getMission());
13    }
14
15    //...
16 }
```

Listing 3.1: Methode setUp und addMissionCardTest aus PlayerTest.java

Wie unschwer zu erkennen ist, werden hier nur gültige Eingaben getestet. Anschliessend wird die dazugehörige Implementation erstellt:

```
1
2  public class Player {
3     private Mission mission;
4
5     //...
6
7     public void setMission(Mission mission) {
8        this.mission = mission;
9     }
10
11    public Mission getMission() {
12       return mission;
13    }
14 }
```

Listing 3.2: Auszug Player.java

Was passiert, wenn eine ungültige Mission oder *null* übergeben wird, wird bisher noch nicht getestet. Hier kommen nun die Contracts zum Zug. Der dazugehörige Test sieht wie folgt aus:

```
1
2  //...
3  @Test(expected = AssertionError.class)
4  public void missionNullTest() {
5     player.setMission(null);
6  }
7  //...
```

Listing 3.3: missionNullTest aus PlayerTest.java

Dieser Test schlägt zunächst fehl. Um ihn auf grün zu bringen, wird die Contract Klasse folgendermassen ergänzt:

```
1
2    @Target
3    private Player target;
4
5    @Override
6    public void setMission(Mission mission) {
7       if (preCondition()) {
8          assert mission != null : "mission can not be null";
9          assert target.getMission() == null : "mission can not be set twice";
10      }
11      if (postCondition()) {
12         assert target.getMission().equals(mission);
13         assert old(target.getPlayerColor()).equals(target.getPlayerColor())
               : "player color not affected";
14         assert old(target.getPlayerId()).equals(target.getPlayerId()) :
               "player id not affected";
15         assert old(target.getOccupiedCountries()).equals(
16               target.getOccupiedCountries()) : "players occupied countries
                     not affected";
17         assert old(target.getCountryCards()).equals(
18               target.getCountryCards()) : "players country cards not
                     affected";
19         assert old(target.getState()).equals(target.getState()) : "player
               state not affected";
20      }
21   }
```

Listing 3.4: setMission aus PlayerContract.java

In der Precondition auf Zeile 8 ist nun schön zu sehen, dass mit dem Ausdruck *assert mission != null : "mission can not be null";* überprüft wird, ob die übergebene Missionskarte null ist. Wäre dies der Fall, würde ein *AssertionError* geworfen werden. Wie in Listing 3.4 ersichtlich ist, wurden noch einige weitere Bedingungen geprüft. Die Regeln des Spieles besagen, dass dem Spieler nur am Anfang des Spieles eine Mission zugewiesen werden darf. Mit der Zeile 5 wurde dies sichergestellt. Bei den Postcondition wurde das Clean Contract Prinzip 4 angewendet, welches in Kapitel 2.3.2 vorgestellt wurde. Zeile 9 beispielsweise zeigt auf, inwiefern sich die Farbe ändert, falls die Methode *setMission* aufgerufen wird. Die Beschreibung "player color not affected" sagt, dass die Farbe bei dem Aufruf nicht verändert wird, was durch das C4J-Framework überprüft wird. Findet hier eine Verletzung statt, wird entsprechend ein *AssertionError* geworfen.

3.5 Der komplexere Vertrag

Beim vorangegangenen Beispiel des Players wird schnell ersichtlich, dass die Logik der Verträge nicht sehr komplex ist, da der Player nicht von anderen Klassen mit Verträgen abhängig ist und er fast ausschliesslich mit get- und set-Methoden arbeitet. Er ist quasi eine Datenklasse und besitzt kaum Logik. Die Playerstati von Kapitel 3.3 sind alleine ohne Bezug auf den Spielkontext ebenfalls nicht interessant. Aus diesen Gründen wird hier nun ein etwas komplexeres Beispiel herangezogen, der *GameInitializer*. In Abbildung 3.4 ist ersichtlich, dass ein gewisser Ablauf eingehalten werden muss, um das Spiel korrekt zu initialisieren. Damit keine ungültigen Eingaben getätigt werden können, überprüfen die Verträge jeweils den Status des *GameInitializer*. Stimmt der Status nicht mit dem erwartenden Status überein, schlägt der Aufruf fehl und der Vertrag wirft einen Error. Wie solch eine Logik implementiert wird, soll hier nun an einem Beispiel gezeigt werden.

Dazu wird die Methode *setTroopsForPlayer* verwendet. Zu Beginn wird von einem positiven Testfall ausgegangen, wie ihn der TbC-Zyklus[14] vorschreibt:

```
1   //...
2   @Test
3   public void setTroopsTest() {
4      prepareUntilPlayersCreated();
5
6      initializer.prepareMissionCards();
7      initializer.prepareCountryCards();
8
9      setTroops(initializer.getNumberOfTroopsToSet());
10
11     assertTrue(riskGame.getStateName().equals(
12           new PlayingState().getStateName()));
13  }
14  //...
```

Listing 3.5: setTroopsTest aus GameInitializerTest.java

Die private Methode *prepareUntilPlayersCreated* bringt das Spiel bereits in den Zustand *PlayersCreatedState*, welcher erforderlich ist, um die beiden Aufrufe *prepareMissionCards* und *prepareCountryCards* auszuführen. Erst dann dürfen gemäss dem Sequenzdiagramm die Truppen der Spieler gesetzt werden. Dies übernimmt hier die private Methode *setTroops*. Am Schluss wird geprüft, ob sich das Spiel auch im korrekten Zustand befindet, nämlich im *PlayingState*. Ob die Truppen korrekt gesetzt wurden ist nicht Bestandteil dieses Tests. Die dazugehörige Implementation wurde so realisiert:

```
1   @Override
2   public void setTroopsForPlayer(
3         Map<Integer, Map<Integer, Integer>> troopsSetByPlayers) {
4      for (Entry<Integer, Map<Integer, Integer>> player : troopsSetByPlayers
5            .entrySet()) {
6         for (Entry<Integer, Integer> countryWithTroops : player.getValue()
7               .entrySet()) {
8            Country countryById = board.getCountryById(countryWithTroops
9                  .getKey());
10           playerList.get(player.getKey()).addOccupiedCountry(countryById);
11           countryById.setNumberOfTroops(countryWithTroops.getValue());
12        }
13     }
14
15     for (Player p : playerList) {
16        p.getState().changeState(p, PlayerState.DEFAULT_TRIGGER);
17     }
18     for (List<CountryCard> li : countryCardsForAllPlayer.values()) {
19        li.clear();
20     }
21     countryCardsForAllPlayer.clear();
22
23     cardDeck.loadCards();
24     getGameState().changeState(game, GameState.DEFAULT_TRIGGER);
25
26     game.setInitialized(createCallbackValueObject());
27  }
```

Listing 3.6: setTroopsForPlayer aus RiskGameInitializer.java

Zu beachten gilt hierbei, dass sowohl die Spielerstati, als auch der Status vom Spiel geändert wurde. Würde der Spielstatus unverändert bleiben, würde der Test in Listing 3.5 fehlschlagen. Dass die Spielerstati verändert werden ist insofern interessant, da diese ihrerseits auch wieder über Verträge verfügen, die erfüllt werden müssen. Würde der Aufruf einen Vertrag verletzen, käme auch hier ein *AssertionError* zum Vorschein, obwohl hier nicht der direkt Aufrufende, nämlich der Test, sondern die Implementation der Methode *setTroopsForPlayer* eine Verletzung begeht. Die etwas komplizierteren *for*-Schlaufen zu Beginn der Methode weisen die Truppen dem Spieler zu. Auch hier werden wieder Vor- und Nachbedingungen auf der Klasse *Player* und *Country* ausgeführt, welche es zu erfüllen gilt.
Weiter sei erwähnt, dass jegliche Überprüfung, ob das übergebene Argument *null* ist, oder ob die Truppen auch korrekt gesetzt wurden, nicht hier stattfindet. Das ist für einen positiven Test auch noch gar nicht nötig. Doch was passiert, wenn nun ein negativer Test zum Zuge kommt?

```
1  @Test(expected = AssertionError.class)
2  public void setTroopsOnTooFewCountriesTest() {
3     prepareUntilPlayersCreated();
4     initializer.prepareCountryCards();
5     initializer.prepareMissionCards();
6     Map<Integer, List<CountryCard>> countryCardsForPlayer = initializer
7           .getCountryCards();
8     Map<Integer, Map<Integer, Integer>> troopsSetByPlayers = RiskGameTest
9           .setTroopsOnCountries(initializer.getNumberOfTroopsToSet(),
10                countryCardsForPlayer);
11    troopsSetByPlayers.get(0).remove(
12          countryCardsForPlayer.get(0).get(0).getCountryId());
13    initializer.setTroopsForPlayer(troopsSetByPlayers);
14 }
```

Listing 3.7: setTroopsOnTooFewCountryTest aus GameInitializerTest.java

Dieser Test überprüft, ob es möglich ist, dass ein Spieler nicht auf alle seiner Länder Truppen setzt. Da dies die Spielregeln verbieten, muss dieser Fehlerfall getestet werden. Um nun die geforderte Exception des Typs *AssertionError.class* zu erhalten, ohne dabei den produktiven Code umzuschreiben, muss der Vertrag für diese Klasse implementiert werden:

```
1  @Override
2  public void setTroopsForPlayer(
3        Map<Integer, Map<Integer, Integer>> troopsSetByPlayers) {
4     if (preCondition()) {
5        assert troopsSetByPlayers != null : "troops can not be null";
6        assert troopsSetByPlayers.keySet().size() ==
             target.getPlayerIds().length : "number of players has to be the
             same as created";
7        checkTroopsSet(troopsSetByPlayers);
8        assert (methodCallFlag & 16) == 0 : "method can only be called
             once";
9        methodCallFlag = methodCallFlag | 16;
10       assert cardsPreparedFlag == 3 : "mission and country cards must
             have been prepared";
11    }
12    if (postCondition()) {
13       assert target.getStateName().equals(
14             GameState.PLAYING_STATE.getStateName()) : "game is in playing
                  state. actual: "
```

```
15              + target.getStateName();
16          assert target.getCountries().equals(old(target.getCountries())) :
                "countries have not changed";
17          assert Arrays.equals(target.getPlayerIds(),
18                old(target.getPlayerIds())) : "players have not changed";
19          assert target.getNumberOfTroopsToSet() == old(target
20                .getNumberOfTroopsToSet()) : "troops to be set unaffected";
21          assert target.getMissionCards().equals(
22                old(target.getMissionCards())) : "countrycards are
                    unaffected";
23          assert target.getCountryCards().size() == 0 : "country cards reset";
24       }
25    }
26
27    private void checkTroopsSet(
28          Map<Integer, Map<Integer, Integer>> troopsSetByPlayers) {
29       for (Entry<Integer, Map<Integer, Integer>> troopsPerCountryPerPlayer :
              troopsSetByPlayers
30              .entrySet()) {
31          final List<CountryCard> countriesForPlayer = target
32                .getCountryCards().get(troopsPerCountryPerPlayer.getKey());
33          int countOfCountriesForPlayer = countriesForPlayer.size();
34          int countOfCountriesInArg = troopsPerCountryPerPlayer.getValue()
35                .keySet().size();
36          assert countOfCountriesInArg == countOfCountriesForPlayer : "count
                of countries in argument must be equal to count of countries
                for player";
37
38          for (Integer countryId : troopsPerCountryPerPlayer.getValue()
39                .keySet()) {
40             boolean playerHasCountry = false;
41             for (CountryCard cc : countriesForPlayer) {
42                if (cc.getCountryId() == countryId) {
43                   playerHasCountry = true;
44                   break;
45                }
46             }
47             assert playerHasCountry : "can not set troops on country that
                    doesn't belong to player";
48          }
49
50          int total = 0;
51          for (Integer troopsPerCountry : troopsPerCountryPerPlayer
52                .getValue().values()) {
53             assert troopsPerCountry > 0 : "must set at least one troop on
                    each country";
54             total += troopsPerCountry;
55          }
56          assert total == target.getNumberOfTroopsToSet() : "wrong number of
                troops set for player";
57       }
58    }
```

Listing 3.8: setTroopsForPlayer aus GameInitializerContract.java

Hier wurde absichtlich der ganze Umfang des Codes abgebildet. Denn nur so kann der Umfang komplexer Verträge nachvollzogen werden. In der Methode *setTroops-ForPlayer* ist vor allem Zeile 7 interessant. Denn hier wird die private Methode

checkTroopsSet aufgerufen, welche die fehlerhafte Eingabe des Tests erkennt und den *AssertionError* wirft. In diesem Fall würde das *assert*-Statement auf Zeile 36 den Fehler entdecken.

4. Schlussfolgerungen

Bereits bei einfachen Klassen wie der in Kapitel 3.4, dem Spieler, macht es Sinn, Verträge einzuführen und gemäss dem TbC-Zyklus [14] zu arbeiten. Einerseits weil Überprüfungen von Argumenten in die Verträge ausgelagert werden können und dies den Code schlanker und übersichtlicher gestaltet, andererseits aus dem Grund, dass sich die Fehlerlokalisierung, durch die Vor- und Nachbedingungen und deren Beschreibung, sehr einfach gestaltet. Es ist direkt ersichtlich, bei welchem Aufruf eine Verletzung des Vertrages stattgefunden hat. Wurde zudem eine verstndliche Beschreibung hinterlegt, weiss der Aufrufer direkt, welche Bedingung er verletzt hat. Als Erklärung soll hier ein kleines Beispiel dienen:

```
1
2   @Target
3   private Player target;
4
5   @Override
6   public void setMission(Mission mission) {
7      if (preCondition()) {
8         assert mission != null : "mission can not be null";
9         assert target.getMission() == null : "mission can not be set twice";
10     }
11     if (postCondition()) {
12        assert target.getMission().equals(mission);
13        assert old(target.getPlayerColor()).equals(target.getPlayerColor())
              : "player color not affected";
14        assert old(target.getPlayerId()).equals(target.getPlayerId()) :
              "player id not affected";
15        assert old(target.getOccupiedCountries()).equals(
16              target.getOccupiedCountries()) : "players occupied countries
                    not affected";
17        assert old(target.getCountryCards()).equals(
18              target.getCountryCards()) : "players country cards not
                    affected";
19        assert old(target.getState()).equals(target.getState()) : "player
              state not affected";
20     }
21  }
22  //...
```

Listing 4.1: setMission aus PlayerContract.java

Bei der Ausführung des Codes

```
1   //...
2   @Test
3   public void duplicateMissionTest() {
4      Player p = new Player(PlayerColor.BLUE, 0);
5      p.setMission(new DefeatPlayerMission());
6      p.setMission(new OccupyCountriesMission());
7   }
8   //...
```

Listing 4.2: duplicateMissionTest aus PlayerTest.java

wird dem Entwickler die Fehlermeldung *java.lang.AssertionError: mission can not be set twice (pre-condition)* gezeigt. Dies ist in Listing 4.1 auf Zeile 9 definiert. Der Aufrufende weiss somit, dass es nicht erlaubt ist, dem Spieler zweimal eine Mission zuzuweisen. Hilfreich ist dies jedoch nicht nur für den Benutzer, sondern auch für den Programmierer der Library. Während der Entwicklungsphase wurde des öfteren ein Vertrag verletzt, da eine Bedingung in Vergessenheit geriet. Durch die konsequente Umsetzung des TbC-Zyklus wurde dies direkt zu Beginn erkannt und der Fehler konnte behoben werden. Dadurch wurde ein sehr robustes Grundgerüst erzeugt, welche die Wahrscheinlichkeit für Fehler stark senkte. Hagen Buchwald hat dazu ein sprechendes Bild veröffentlicht:

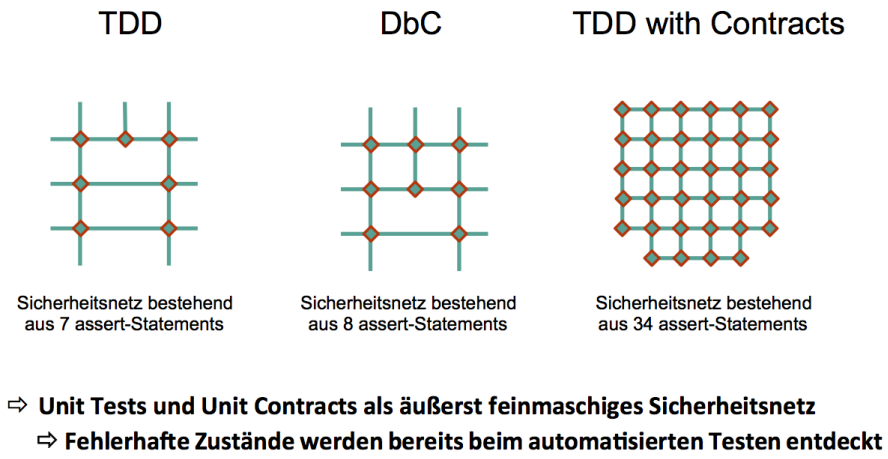


Abbildung 4.1: Feinmaschiges Sicherheitsnetz mit TbC[3]

Durch die Einführung von Verträgen ist der Entwickler gezwungen, sich an diese Vorgaben zu halten. Dies hat auf die Robustheit des Codes und somit des Programmes und seiner Logik einen starken Einfluss. Robustheit[4] steht dabei für die Eigenschaft, auch unter ungünstigen Bedingungen noch zuverlässig zu funktionieren. Werden die Verträge bereits in den tiefen Layern sauber definiert, bilden sie so eine solide Grundlage für die Schichten, die darauf aufbauen. Die Verträge der darüberliegenden Schichten werden dann deutlich komplexer, wie in Kapitel 3.5 gezeigt wird. Denn der Entwickler wird gezwungen, seine eigenen Regeln auf alle Fälle einzuhalten.

Auf den ersten Blick sieht dies sehr positiv aus, es kann jedoch auch seine Tücken haben. Bei der Implementierung der Tests für das *GamePlay* stellten sie einige Hürden in den Weg. Bei den positiven Tests verlief vorerst alles wie gewohnt. Für

das Gewinnen des Spiels musste jedoch ein etwas komplexerer Algorithmus entwickelt werden. Zuerst musste die Missionskarte des Spielers gelesen, anschliessend seine Mission zur Erfüllung gebracht werden. Da Anfangs noch keine Verträge implementiert wurden, war dies auch kein Problem, denn es war durchaus legitim, dem Spieler neue Länder vom *Board* zuzuweisen um so die Mission zu erfüllen. Mit der Einführung der Verträge war dies jedoch nicht mehr möglich. Der Vertrag verbietet es, dass der Spieler gewisse Spielzüge machen darf. Es darf dem Spieler kein Land hinzugefügt werden, das einem anderen Spieler gehört, ohne es gemäss den Spielregeln zu erobern. Um dies nun zu testen, hätten für das Spiel so genannte *Bots*[22] implementiert werden müssen, also Programme, welche das Spiel vollautomatisch testen. Dies hätte jedoch den Umfang dieser Studienarbeit gesprengt, weshalb darauf verzichtet wurde. Als Kompromiss wurden die Tests für den Spielverlauf entweder mit oder ohne aktiviertem Framework C4J durchgeführt.
Gerade bei komplexen Klassen wie dem *GameInitializer* haben sich die Verträge jedoch bezahlt gemacht. Denn so konnte ungefähr die Hälfte des Codes ausgelagert werden. Sowohl Vertrags- als auch Implementationsklasse besitzen 230 Zeilen Code. Hätte dies in einer Klasse untergebracht werden müssen, wären direkt 500 Zeilen Code angefallen. Dies hätte der Übersicht stark geschadet.

Der TbC-Zyklus hat sich im Verlaufe der Arbeit als sehr effektiv und effizient herausgestellt. Dadurch, dass nur Teile von TDD [6] und DbC [5] übernommen werden, wird Redundanz und somit duplizierter Code vermieden. Das feinmaschige Sicherheitsnetz, das sich während der Entwicklung einstellt, hilft dem Programmierer, robusten Code zu schreiben. Die Möglichkeit, die Verträge im produktiven System auszuschalten hat den Vorteil, dass sowohl die Performance gesteigert wird, als auch der Code übersichtlicher und besser strukturiert wird. Die Möglichkeit, den Vor- und Nachbedingungen sinnvolle Beschreibungen zuzuweisen erleichtern zudem die Dokumentation des Codes und die Verwendung der Klasse.

4.1 Zusammenfassung

1. Der TbC-Zyklus wurde während der Semesterarbeit als gut durchdacht und effizient empfunden. Er ist iterativ und für TDD-erfahrene Entwickler geht die Umstellung schnell von statten. Vor allem die Fehlerlokalisierung bei Verletzungen der Vor- oder Nachbedingungen werden als sehr hilfreich eingestuft.

2. Das durch den TbC-Zyklus resultierende Sicherheitsnetz wurde schnell zu schätzen gelernt. Es hilft dem Entwickler, sich an vergessene Überlegungen zu erinnern und bildet eine solide Grundlage für weiteren Code. Es senkt die Wahrscheinlichkeit für Fehler und lässt ihnen wenig Spielraum

3. Testing by Contract birgt jedoch einige Tücken. Die Tests müssen unter Umständen, zum Beispiel bei komplexer Logik, umfangreicher geschrieben und durchdacht werden, was zeitraubend und aufwändig sein kann. Die Mühe wird aber durch robusten Code belohnt, was Sinn und Zweck von TbC ist.

4. C4J ist ein sehr hilfreiches Tool, um Testing by Contract umzusetzen. Die Möglichkeit, Vertragsklassen zu einem späteren Zeitpunkt ohne Eingriff in den aktuellen Code hinzuzufügen, ist überaus praktisch und hilfreich. Die Dokumentierung der Verträge ist durch C4J ebenfalls sichergestellt. Die klare Syntax von C4J gibt auch DbC-unerfahrenen Entwicklern einen einfachen Einstieg.

5. Erfahrungsberichte

5.1 Erfahrungsbericht Matthias Hauser

Das Erarbeiten dieser Studienarbeit war für mich eine interessante und lehrreiche Herausforderung. Der Beginn war zwar etwas holprig, da relativ lange eine konkrete Aufgabenstellung fehlte. Gleichzeitig konnten wir so den Verlauf der Arbeit auch noch gemäss unseren Wünschen und Vorstellungen etwas steuern.
Diese erste Phase nutzten wir, um uns ins Thema Design by Contract einzuarbeiten. Das Konzept war mir in den Grundzügen schon bekannt. Durch das Buch Design by Contract by Example wurde mir dann jedoch vieles noch klarer. Um uns mit DbC und dem Framework C4J vertraut zu machen, begannen wir mit einigen kleinen Beispielen.
Hier folgte für mich die erste Ernüchterung. Wir stolperten nämlich genau über den fehlenden agilen Charakter von DbC. Im Gegensatz zum mir gut bekannten TDD verlangt DbC von Beginn weg eine ziemlich klare Vorstellung des Designs. Es ist zwar möglich, bis zu einem bestimmten Grad inkrementell vorzugehen, doch war für mich der initiale Aufwand, über das Design nachzudenken, schon etwas zu weit vorgegriffen. Gleichzeitig fing mir das Konzept der Contracts an zu gefallen. Doch es musste ein anderes Vorgehen her.
Auf die Sprünge half uns schliesslich Hagen Buchwald, der uns einen konkreten Vorschlag präsentieren konnte, wie Testing by Contract aussehen könnte. Unterdessen war auch klar, was unsere Beispielapplikation sein sollte. So konnten wir nun beginnen, TbC anschaulich anzuwenden.
Anfangs lagen noch einige Stolpersteine auf dem Weg, sowohl in Bezug auf die Tools, als auch auf die neu zu erlernende Methodik. Auch hier war uns Herr Buchwald wieder eine grosse Hilfe, etwa wenn es darum ging eine Lösung zu finden, wie JUnit Tests im Verbund mit C4J in eclipse zum Laufen gebracht werden.
Anhand von noch nicht so komplexen Klassen tasteten wir uns an TbC heran. Gleichzeitig wuchs der Respekt vor dem Zeitpunkt, wo wir uns um die Contracts von weitaus komplexeren Klassen kümmern sollten. Dies war teilweise auch berechtigt und es wurde mir einmal mehr klar, weshalb man im Code eine lose Kopplung zu erreichen versucht.
Für mich war die Lernkurve in dieser Arbeit sehr erfreulich. Phasen mit eintöniger Fleissarbeit gab es für mich fast nie. Ständig tauchten neue Probleme und Herausforderungen und damit auch Erkenntnisse auf.
Abschliessend kann ich sagen, dass mir die Arbeit gut gefallen hat und ich mit den Ergebnissen zufrieden bin. Gerade C4J scheint mir ein gelungenes Tool zu sein um mit Contracts zu arbeiten. Ich kann mir gut vorstellen, in Zukunft die Entwicklung von TbC weiter zu verfolgen und auch einzusetzen.

5.2 Erfahrungsbericht Philip Kaufmann

Zu Beginn der Arbeit war mir noch nicht klar, was genau von uns erwartet wird. Das Thema Testing by Contract war mir total fremd und über Contracts hatten wir nur sehr wenig im Modul Software Engineering 2 (SE2) gelernt. Doch die Studienarbeitbeschreibung war für mich sehr ansprechend und neue Programmierpraktiken, um seinen eigenen Code robuster zu machen, fand ich schon immer interessant.
Anfangs war die Einarbeitung in die Theorie von Nöten. Das Buch Design by Contract by Example von Richad Mitchell und Jim McKim [5] war sehr aufschlussreich, auch wenn es vor allem mit der Sprache Eiffel erklärt wurde. Es gab einem jedoch einen guten Einblick in die Funktionsweise und den Hintergedanken von Contracts. Weitere Unterlagen wurden uns von unserem Betreuer, Prof. Hans Rudin, zugesendet. Dies beinhaltete einen Vortrag vom Schöpfer und Erfinder von C4J, Jonas Bergström. Er stellte seine Pokersoftware vor und erklärte, dass er mit Unit-tests nie den gesamten Umfang der Applikation abdecken könnte. Es waren auch Zustandsdiagramme der Spiele und Spieler vorhanden. Ich konnte mir zwar in etwa eine Vorstellung machen, wieso er nun Contracts brauchte und wieso dass er zum Testen Bots gebraucht hat, aber ganz verstanden habe ich es erst zu Ende unseres eigenen Projektes.
Parallel zur Einarbeitung in die Theorie versuchten wir uns am Framework C4J. Anfangs programmierten wir einen Stack, da sowohl im Buch als auch in SE2 der Stack als Beispiel herangezogen wurde. Bereits dort sind wir auf einige Fallstricke gestossen. Zum Beispiel wurde im XML-File vergessen, die Validierung der @Pure-Methoden (Abfragen, queries) zu aktivieren. Wir waren dann sehr überrascht, dass wir in den Basic Queries Änderungen an der Klasse machen konnten, ohne dass ein AssertionError geworfen wurde. Nach einigen Nachforschungen konnte jedoch auch dieser Fehler behoben werden.
Es musste nun jedoch eine komplexere Applikation her, um TbC auf Herz und Nieren zu prüfen und einen aussagekräftigen Abschlussbericht zu liefern. Ein Spiel erschien uns sehr passend, eines, das auch Zustände hat und von der Logik einiges hergibt. Nach einigen Überlegungen entschieden wir uns für das Spiel Risiko. Um es ganz zu verstehen und die Abläufe aufzuzeichnen, haben wir das Brettspiel ausgelehnt und durchgespielt. Schnell war klar, dass dies ein ideales Spiel für unsere Arbeit sein wird. Sowohl Spiel als auch Spieler besitzen Zustände und die Regeln können mit den Contracts gut abgebildet werden. Während der Evaluierung des Spieles nahmen wir Kontakt mit Hagen Buchwald, Besitzer der zweiten Generation von C4J, auf. Wir fragten an, ob er Unterrichtsunterlagen zu C4J, best practices oder weitere Literatur zu C4J besitzt, die er uns zustellen möchte. Unsere Emails wurden prompt beantwortet und im Anhang fanden wir sehr hilfreiche Folien, auf welchen unsere Studienarbeit aufbaut. Er beschrieb den DbC-, TDD- und TbC-Zyklus sehr ausführlich und erklärte, dass der TbC-Zyklus mit ungefähr 1800 Studenten erarbeitet wurde. Im Verlaufe des Projektes haben wir uns diesen angeeignet und auf seiner Basis unsere Applikation geschrieben.
Da anfangs der normale TDD-Zyklus im Vordergrund stand, gab es für mich nicht eine allzu grosse Umstellung. Lediglich nur die positiven Tests schreiben zu dürfen war etwas ungewohnt. Ich habe mir immer überlegt, was denn nun mit den negativen Tests passiert. Doch als ich dann den ersten Contract implementieren durfte, wurde mir schnell klar, wie und wo sie getestet werden. Diesen Ansatz fand ich sogar sehr sinnvoll, da die Überprüfung so nicht mehr im produktiven Code, sondern in den Contracts ausgelagert war.
Da unsere Applikation nur so viele Informationen wie nötig an die Benutzeroberfläche preisgeben möchte, stellte sich mir anfangs die Frage, wieso wir denn schon auf Player- oder Carddeckbasis Contracts schreiben müssen, da wir die Klassen selbst brauchen und uns doch an die Abmachung halten. Doch kaum wurde die Spiellogik

implementiert, wurde schnell klar, dass die eine oder andere Abmachung vergessen geht und umso mehr war ich um die für einen selbst geschriebenen Verträge froh. Oft nämlich haben wir Aufrufe in den Tests gemacht, die nicht legitim waren. Das Framework hat uns dann freundlicherweise mit einem Fehler in den Tests darauf hingewiesen. Was hier als Vorteil klingt, sollte sich später jedoch auch als Nachteil herausstellen.
Die ganze Spiellogik, sprich die Regeln, waren zu Beginn in einer Klasse zentralisiert. Obwohl wir das Spiel in Initialisierungs- und Spielphase aufgeteilt hatten, wurden sie in einer Klasse gehalten. Doch als ich die Tests, anfangs nur die positiven, für die Initialisierungsphase geschrieben habe und diese bereits über 300 Zeilen Code (ohne Contracts) enthielt, merkten wir, dass dies getrennt werden musste. Zudem war mit den positiven Tests ja nur der "Happy-path" abgebildet, die ganze Logik, wann was aufgerufen werden darf, war noch gar nicht enthalten. Hier wurde uns dann so langsam bewusst, dass die Contracts sehr komplex und aufwändig werden würden. Doch nur schon beim Schreiben der positiven Tests bin ich in einige Probleme gelaufen. Um gewisse Aufrufe zu machen, um beispielsweise Truppen auf den Ländern des Spielers zu verteilen, konnte ich nicht einfach dem Spieler einige Länder zuweisen und eine beliebige Anzahl von Truppen verteilen. Die Contracts verhinderten dies, da es keine legitimen Aufrufe auf den Klassen waren. Ich durfte also Algorithmen schreiben, welche sich dynamisch an den Zustand des Spieles anpassen und entsprechend reagieren, um die Tests zum erfolgreichen Durchlaufen zu bringen. Dies zog sich dann durch die ganzen Tests durch, wodurch die Zeit vor allem in Testcode investiert wurde.
Da die komplette Testklasse der Initalisierungsphase bereits 500 Zeilen Code beinhaltete, entschieden wir uns, Initialisierung und Spiel zu trennen. Beim Spiel stiessen wir jedoch auf die selben Probleme wie bei der Initalisierung. Um ein Spiel zu gewinnen, mussten komplexe Algorithmen geschrieben werden. Es ging sogar so weit, dass wir die Tests mit und ohne Contracts trennen und die Testklassen einzeln ausführen mussten. Das Problem hierbei war, dass die Contracts verhinderten, dass wir uns den Sieg unrechtmässig erschlichen. Und da das Spiel zufällig die Missions- und Länderkarten verteilt, und wir so im vornherein nicht wissen können, welche Karten wir erhalten, konnten wir keine Tests mit aktivierten Contracts schreiben, um das Spiel zu gewinnen. Hier wurde mir bewusst, wozu für die Pokerapplikation von Jonas Bergström Bots geschrieben wurden. Wir sind selbst an einem Punkt angekommen, wo Bots geschrieben werden müssten, um unsere Risiko-Library zu testen. Ansonsten können niemals alle Zustände und Variationen abgedeckt werden. Doch dies hätte den Aufwand für die Studienarbeit gesprengt.
Abschliessend ist zu sagen, dass ich TbC mit C4J auch weiter einsetzen werde. Ich bin überzeugt, dass mit TDD und DbC vereint robusterer Code geschrieben werden kann. Immer wieder stiess ich auf eine Vertragsverletzung, an die ich nicht mehr gedacht habe. Bei der Implementation einer neuen Funktion konnte man so auch direkt testen, ob nicht doch irgendwo ein falscher Aufruf gemacht wird. Die Möglichkeit, die Überprüfung der Contracts in einer Applikation im produktiven Betrieb auszuschalten und somit eine Performancesteigerung zu erreichen spricht ebenfalls für den Einsatz von C4J. Durch die Beschreibung der Pre- und Postconditions hat man zudem direkt eine saubere Dokumentierung seines Codes und der benutzende Programmierer weiss direkt, wie er mit der Komponente umzugehen hat.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[1] Ralph E. Johnson, Brian Foote: “Designing Reusable Classes“ im “Journal of Object-Oriented Programming“ (1988)

[2] http://de.wikipedia.org/wiki/Agile_Softwareentwicklung

[3] Dr. Lars Alvincz, Hagen Buchwald - *TDD with Contracts-v2*, Folie 28, XP Days Germany 2013, 14. November 2013

[4] http://de.wikipedia.org/wiki/Robustheit, abgerufen am 12.12.2013

[5] Richard Mitchell, Jim McKim, *Design by Contract, by Example*

[6] Mike Hill, Joshua Kerievsky - *Test-Driven Development*
Evolving Simple Designs Guided By Tests - Java Edition

[7] Dr. Lars Alvincz, Hagen Buchwald - *TDD with Contracts-v2*, Folie 22-28, XP Days Germany 2013, 14. November 2013

[8] http://c4j-team.github.io/C4J/, abgerufen am 12.12.2013

[9] http://www.eclipse.org/, abgerufen am 12.12.2013

[10] Dr. Lars Alvincz, Hagen Buchwald - *TDD with Contracts-v2*, Folie 11-14, XP Days Germany 2013, 14. November 2013

[11] https://en.wikipedia.org/wiki/Liskov_substitution_principle, abgerufen am 12.12.2013

[12] Hagen Buchwald - *Contracts for Java - VM4*, Folie 9, DHBW Karlsruhe, 28. Juni 2013

[13] Hagen Buchwald - *Contracts for Java - VM4*, Folie 10, DHBW Karlsruhe, 28. Juni 2013

[14] Dr. Lars Alvincz, Hagen Buchwald - *TDD with Contracts-v2*, Folie 26, XP Days Germany 2013, 14. November 2013

[15] http://c4j-team.github.io/C4J/about.html, abgerufen am 12.12.2013

[16] http://c4j-team.github.io/C4J/under-the-hood.html, abgerufen am 12.12.2013

[17] http://sourceforge.net/p/c4j/discussion/547819/thread/07ea2493, abgerufen am 12.12.2013

[18] Aus einer Kurzreferenz, die an der DHBW an die Studenten abgegeben wird, C4J6.0 - Syntax, DHBW Karlsruhe, 04. Juni 2013

[19] https://en.wikipedia.org/wiki/Risk_(game), abgerufen am 12.12.2013

[20] https://en.wikipedia.org/wiki/State_pattern, abgerufen am 12.12.2013

[21] https://en.wikipedia.org/wiki/Separation_of_concerns, abgerufen am 12.12.2013

[22] https://en.wikipedia.org/wiki/Computer_game_bot, abgerufen am 12.12.2013