Technische Universität Berlin

Software and Embedded Systems Engineering Group
Prof. Dr. Sabine Glesner

www.sese.tu-berlin.de Sekr. TEL 12-4 Ernst-Reuter-Platz 7 10587 Berlin

Softwaretechnik und Programmierparadigmen WiSe 2020/2021

Prof. Dr. Sabine Glesner
Joachim Fellmuth joachim.fellmuth@tu-berlin.de
Guilherme Grochau Azzi g.grochauazzi@tu-berlin.de
Simon Schwan s.schwan@tu-berlin.de

Hausaufgabenblatt 2 **Ausgabe: 29.01.2021** **Abgabe: 28.02.2021**

Im Rahmen dieser Hausaufgabe erweitert Ihr den generischen Spiele-Server aus der freiwilligen Hausaufgabe um das Spiel **Crazyhouse** mit leichten Modifikationen. Eure Aufgabe besteht dabei aus drei Teilen:

Server: Der Gameserver verwaltet den Spielablauf und die Verbindung mit dem Web-Frontend. Seine Hauptaufgabe besteht darin, *Züge auf ihre Korrektheit zu überprüfen* und *den neuen Spielzustand zu berechnen* und auszuliefern. Eure Aufgabe ist die konkrete Implementierung der Spielregeln von **Crazyhouse** innerhalb der Gameserver-Vorgabe. Die Implementierung erfolgt objekt-orientiert in **Java**.

Qualitätssicherung: Um einen korrekten Spielverlauf gewährleisten zu können, soll die von euch implementierte Funktionalität des Servers ausgiebig mit Hilfe von **JUnit** *getestet* werden. Außerdem soll die *Codequalität* mithilfe bestimmter Metriken gemessen und sichergestellt werden.

Bot: Da neben dem Spiel gegen einen menschlichen Herausforderer auch das Spiel gegen Computergegner möglich sein soll, entwickelt Ihr einen Bot in **Haskell**. Der Bot soll für gegebene, korrekte Spielzustände *alle möglichen Züge berechnen* und *einen aussuchen*. Es soll zum Schluss möglich sein, verschiedene Bots gegeneinander antreten zu lassen und so ein Turnier eurer Implementierungen zu veranstalten.

Crazyhouse

Crazyhouse[1] ist eine von vielen Varianten von Schach. Die Grundidee ist, dass keine Figuren geschlagen werden, sondern sie werden gefangen genommen und wechseln die Seite. Es handelt sich also um die normalen Schach-Regeln[2] mit ein paar Abweichungen:

Crazyhouse-Regeln:

- Eine geschlagene Figure (*captured*) verlässt das Spiel nicht, sondern wechselt die Farbe und geht in die Reserve (*pocket*) des schlagenden Spielers. Dort können beliebig viele Figuren aller Arten sein.
- Als gültiger Zug kann eine beliebige Figur aus der Reserve auf einem freien Feld eingesetzt werden (*drop*). Dabei gelten weiterhin die Schach-Regeln, d.h. ein Zug muss so durchgeführt werden, dass der König am Ende des Zuges nicht im Schach steht. Für Bauern (*pawn*) gibt es die Einschränkung, dass sie nicht in die erste oder letzte Reihe des Spielfelds eingesetzt werden dürfen.
- Anders als in Crazyhouse lassen wir alle umgewandelten Figuren (*promotion*; wenn ein Bauer die letzte Reihe erreicht) in ihrer umgewandelten Form in der Reserve. Sie werden nicht wieder in Bauern zurückverwandelt, sondern wechseln nur die Farbe.

Weitere Einschränkungen, mit denen wir von den Schach-Regeln abweichen:

- Es gibt keine Rochade (*castling*).
- Es kann nicht "En-passant" geschlagen werden. Bauern können entweder geradeaus laufen oder schräg schlagen.
- Wenn ein Bauer die gegenüberliegende Seite des Feldes erreicht (Promotion), wird er in eine zusätzliche Dame verwandelt. Der Benutzer kann nicht wählen.

Für das Spielende gilt folgendes:

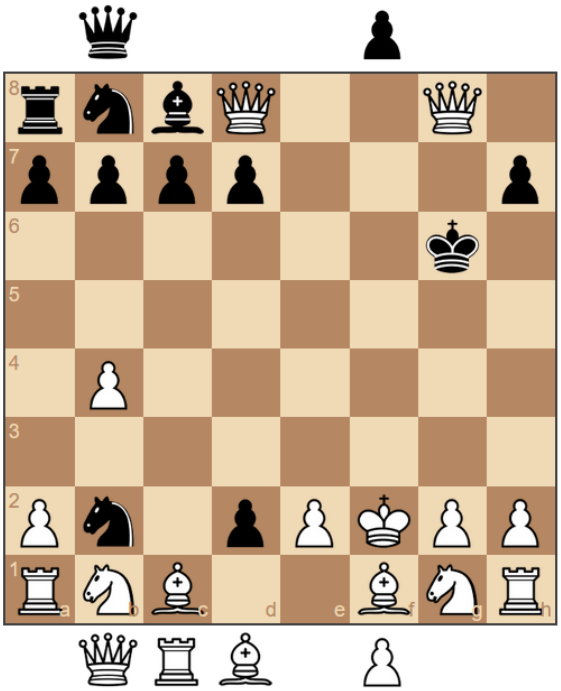
- Das Spiel endet automatisch wenn ein Spieler schachmatt ist, d.h. wenn ein Spieler durch seinen Zug den anderen Spieler in Schach setzt, und der Gegenspieler dann keinen gültigen Zug mehr machen kann, ist das Spiel vorbei und der ziehende Spieler hat gewonnen.
- Es muss kein automatisches Patt (*stalemate*) oder Remis umgesetzt (*draw*) werden. Wenn ein Spieler an der Reihe ist und keine Züge mehr machen kann, kann er über die GUI ein Unentschieden vorschlagen. Ein Bot kann in dem Fall keinen Zug zurückliefern und gibt damit automatisch auf.

[1] https://en.wikipedia.org/wiki/Crazyhouse
[2] https://de.wikipedia.org/wiki/Schach

Notation

Spielzustand (Board)

Unser Spielzustand hat drei wesentliche Anteile: Das 8x8-Felder Spielbrett selbst und die Reserven der beiden Spieler, in denen jeweils beliebig viele Figuren der jeweiligen Farbe enthalten sein können (Die Anzahl ist begrenzt durch die Züge ermöglichte Spielsituationen). In der GUI sieht das wie folgt aus. Über und unter dem Spielbrett befinden sich die Figuren der Reserven. Unsere GUI zeigt nur Figuren der Reserve an, wenn mindestens eine solche Figur enthalten ist.



Wir übertragen die Spielzustände in einem String vom Format ähnlich der ersten Komponente der Forsyth-Edwards-Notation (FEN-String)[3].

Eine Figur wird durch einen einzelnen Buchstaben beschrieben, und zwar wie folgt:

Figur	Repräsentation (schwarz)	Repräsentation (weiß)
König (king)	k	K
Dame (queen)	q	Q
Läufer (bishop)	b	B
Springer (knight)	n	N
Turm (rook)	r	R
Bauer (pawn)	p	P

[3]https://de.wikipedia.org/wiki/Forsyth-Edwards-Notation

Die Repräsentation des Gesamtspielbretts ist in neun Abschnitte, jeweils getrennt durch `'/'`, unterteilt. Die ersten acht Abschnitte repräsentieren die Reihen, beginnend von oben (bei der achten Reihe). Innerhalb einer Reihe sind die Inhalte in Spalten beginnend von 1 aufgeführt. Für eine Figur auf einem Feld steht der jeweilige Buchstabe. Leere Felder werden durch ihre Anzahl nebeneinanderliegend dargestellt. Eine Reihe mit zwei schwarzen Bauern (Spalten b und f) wird also so aussehen: `1p3p2`. Achtung: Abweichend von der allgemeinen Form der FEN-Strings gehen wir immer von der möglichst kompakten Form aus, d.h. `11n122` wird immer zu `2n5`. Im letzten Abschnitt sind die beiden Reserven zusammen enthalten: Die Figuren sind immer sortiert (nach Sortierung von char, also mit Großbuchstaben vor Kleinbuchstaben) aufgeführt, z.B. `NPPQbppp`. Das Bild oben wird beispielsweise wie folgt dargestellt:

`"rnbQ2Q1/pppp3p/6k1/8/1P6/8/Pn1pPKPP/RNB2BNR/BPQRppq"`

Der Startzustand des Spiels ist der Folgende:

`"rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR/"`

Spielzug (Move)

Ein Spielzug wird als ein String der Form `[Urspung]-[Ziel]` übertragen. Das Ziel ist immer eine *Position* der Form `[Spalte][Reihe]`, wobei das Feld unten links `a1` heißt und das Feld oben rechts `h8`. Der Ursprung ist entweder auch eine *Position*, wenn eine Figur auf dem Feld bewegt werden soll, oder eine *Figur* in der oben genannten Repräsentation wenn eine Figur aus der Reserve eingesetzt werden soll. Eine Bewegung auf dem Feld wäre zum Beispiel der Spielzug `"b3-c4"`, und mit `"r-c4"` würde ein schwarzer Turm auf das Feld `c4` gesetzt werden.

Hinweise zur Bewertung

Die Bewertung Eurer Abgabe erfolgt (soweit möglich) automatisch. Daher werden in erster Linie *funktionale Aspekte* Eurer Abgabe bewertet. Wir legen Wert auf eine faire Bewertung, das heißt, wir wollen sicherstellen, dass für alle Teilnehmer dieselben Bedingungen gelten und alle an dem selben Maßstab gemessen werden. Die faire Bewertung nicht-funktionaler Eigenschaften würde also eindeutig definierte Maßstäbe voraussetzen. Doch wie misst man die Schönheit von Code? Oder die Sinnhaftigkeit von unvollständigen Codefragmenten? Für einige Kriterien gibt es gute Metriken, deren Einhaltung wir unter dem Punkt Qualitätssicherung bewerten können. Für alles andere gilt, “sinnvoll” ist Code genau dann, wenn er der Erfüllung der Use-Cases zuträglich ist - was wir mit unseren Tests (und Ihr mit Euren) überprüfen.

1. Gameserver für Crazyhouse in Java (12 Punkte)

Aufgabenbeschreibung

Der Großteil des Gameservers wurde bereits implementiert. Das abstrakte Modell aus Hausaufgabe 1 wurde, ähnlich wie in der Beispiellösung spezifiziert, bereits umgesetzt. Außerdem ist die GUI und die Kommunikation dorthin bereits gegeben. Die Aufgabe ist es nun, in der bereits angefangenen Klasse CrazyhouseGame für das konkrete Spiel **Crazyhouse** ein Datenmodell zu entwickeln, mit dem der Spielzustand beschrieben werden kann. Unter Verwendung dieses Datenmodells soll die Funktion `tryMove(String moveString, Player player)` implementiert werden, die einen Zug prüft und ggf. durchführt. Konkret ist dabei folgendes zu tun:

- Ein Benutzer könnte das System angreifen oder schummeln, indem er die Anfragen an den Server manipuliert. Es ist zu prüfen, ob die Eingabe des Zuges im oben genannten Format erfolgt ist. Es ist außerdem zu prüfen, ob der übergebene Player an der Reihe ist.

- Es ist zu prüfen, ob der Zug nach den in diesem Aufgabenblatt gegebenen Regeln von Crazyhouse gültig ist.

- Falls der Zug gültig ist, muss er durchgeführt werden, d.h. die Änderung muss in der internen Repräsentation des Spielfelds hinterlegt werden, sodass der geänderte Spielzustand zurückgegeben werden kann. Anschließend muss der andere Spieler am Zug sein. Falls das Spiel durch den Sieg des Spielers endet muss dies entsprechend in den Attributen der Klasse `Game` vermerkt werden. Außerdem muss die Spielhistorie gepflegt werden (s. Oberklasse *Game*).

Im Normalfall startet ein neues *CrazyHouseGame* mit dem Startzustand (siehe Spiel-Regeln).

Zusätzlich sind noch zwei Funktionen zu implementieren, mit denen ein Spielzustand (siehe Notation) übergeben (`setBoard(String)`) bzw. abgerufen `String getBoard()` werden kann. `setBoard()` dient zusammen mit der schon existierenden Funktion `setNextPlayer()` in unseren und euren Testfällen dazu, das Spiel an jedem beliebigen Punkt zu starten und so einzelne Situationen kompakter testen zu können. Als Testfunktion muss sie nicht die Eingaben auf Validität prüfen. Stattdessen kann angenommen werden, dass alle Eingaben wohl-formatierte Spielsituationen sind, die tatsächlich in einem regelkonformen Spiel entstehen können.

Sämtlicher von euch eingefügter Code soll mit JUnit-Tests automatisiert getestet werden. Dabei soll 100% Zweigabdeckung (auf Bytecode-Ebene) erreicht werden. Testgegenstand ist dabei allerdings die Spezifikation - wenn also Funktionalität fehlt, gelten die Testfälle entsprechend auch nicht als vollständig. Nur diejenigen Testfälle werden bei der Messung der Testabdeckung gewertet, die auch erfolgreich durchlaufen.

Nach der Implementierung der Funktionalität und der Testfälle müssen eventuell noch geeignete Refactorings angewendet werden. Der von euch zu entwickelnde Code muss die folgenden Metriken erfüllen:

Metrik	Maximalwert
Zeilen pro Funktion (Method Lines of Code)	25
McCabe zyklomatische Komplexität	10
Verschachtelungstiefe (Nested Block Depth)	4
Anzahl Parameter pro Funktion	5

Vorgabe

Als Vorgabe haben wir euch ein Eclipse Servlet-Projekt zusammengestellt, in dem sämtliche Client-Funktionalität (HTML-Dateien mit Javascript) und das Servlet bereits enthalten sind. Im Projekt existiert auch eine README-Datei, die einige Informationen zur Orientierung enthält. In den schon vorhandenen Klassen markieren TODO-Kommentare die Stellen, an denen Ihr weitermachen sollt (Funktionalität fehlt in der Klasse CrazyHouseGame). Selbstverständlich fehlen auch noch Klassen. Neue Klassen müssen im Paket `de.tuberlin. sese.swtpp.gameserver.model.crazyhouse` erstellt werden. Bestehender Code darf nicht verändert werden (außer wenn es explizit da steht).

Die gewählte Test-Ebene ist ein Integrationstest mit den öffentlichen Funktionen der Klasse Game als Schnittstelle [4] zwischen Spiel und restlichem Gameserver, bzw. der GUI. Die von euch implementierten Funktionen sind der Anteil des Schnittstelle, den eure Tests abdecken sollen. Für die Testfälle haben wir ein spezielles Format definiert. Ein Beispieltestfall findet sich in der Klasse `TryMoveIntegrationTest`[5] (hier sollen auch eure Testfälle implementiert werden). Es stehen euch für die Testfälle drei Funktionen zur Verfügung:

- Die Funktion `startGame()` sorgt dafür, dass das Spiel gestartet wird. Zum einfacheren Testen kann optional ein beliebiger Spielstand (Figurenbelegung auf dem Board im FEN-String) und die Auswahl, wer an der Reihe sein soll, übergeben werden.
- Die Funktion `assertMove(...)` übergibt dem Spiel einen Spielzug und den ausführenden Spieler, und prüft, ob das Ergebnis eurem erwarteten Ergebnis entspricht (tryMove gibt true zurück, wenn der Spielzug gültig war und durchgeführt wurde).
- Die Funktion `assertGameState(...)` prüft, ob sich der Gesamt-Spielzustand (aktuelles Board(FEN), nächster Spieler, Spiel beendet oder nicht, unentschieden, wer hat gewonnen) mit dem deckt, was ihr als Erwartung übergebt.

[4] Normalerweise würden wir die Testabdeckung mit *Unit-Tests* erreichen, die durch kleinteiligere TestCase-Klassen (z.B. für jede Klasse) implementiert sind. Nun wissen wir aber nicht welche Klassen und Methoden ihr implementiert, und wie sinnvolle Tests dafür aussehen bzw. automatisch geprüft werden könnten. Daher verwenden wir eine Schnittstelle, die uns und euch vorab bekannt ist. Dies bedeutet auch, dass euer Code auf manche Dinge verzichten muss (unbenutzte Funktionen, zusätzliche asserts, default branches in switches...), die ihr über die Schnittstelle nicht abdecken könnt.

[5] de.tuberlin.sese.swtpp.gameserver.test.crazyhouse

In euren Testfällen dürfen nur diese drei Funktionen mit explizit als String-Konstanten und Booleans übergebenen Argumenten (keine Variablen, siehe Beispieltestfall) verwendet werden. Jeder Testfall kann beliebig viele Aufrufe all dieser Funktionen enthalten - für eine hohe Abdeckung empfehlen sich aber sehr kurze Testfälle (Nur erfolgreiche Testfälle werden gewertet). Jeder Testfall **muss** mit einem Aufruf von `assertGameState` enden. Testfälle, die zusätzliche Funktionen aufrufen oder nicht dem Format entsprechen, werden NICHT berücksichtigt.

Hinweise

- In der Vorgabe ist eine simple Persistierung eingebaut, die erstellte Daten speichert und beim Neustart wieder lädt. Das soll euch das manuelle Testen über die GUI erleichtern. Sollte es zu inkonsistenten Zuständen des Webservers kommen, so könnt ihr die Daten löschen, indem ihr die Datei `test.db` löscht. Diese wird mithilfe eines relativen Pfades in der Klasse *GameServerServlet* definiert. Neue Klassen sollten das Interface `java.io.Serializable` implementieren. Der Speicherort des Datenbank-Files darf angepasst werden.

- Wenn ihr den Bot (siehe Aufgabe 2) mit eurer Java-Implementierung ausprobieren wollt, könnt ihr ihn mit Hilfe der Klasse HaskellBot anbinden. Dazu müsst ihr den Bot zu einer ausführbaren Datei kompilieren (siehe Main.hs) und den Pfad, in dem diese zu finden ist, in der GameFactory-Klasse hinterlegen.

- Die Abdeckung wird von uns mit dem Plugin *Emma* in Eclipse geprüft. Wenn all euer Code in Emma grün angezeigt wird (100% **Branch** Coverage), ist ausreichende Testabdeckung erreicht. Die Abdeckung wird abzüglich der Anteile gewertet, die durch die Testfälle in der Vorgabe schon abgedeckt sind. Außerdem werden wir mit eigenen Testfällen vom gleichen Format prüfen, ob die Funktionalität vollständig implementiert ist. Bei fehlender Funktionalität fehlen also zusätzlich auch Testfälle.

- Die Vorgabe enthält einen Example-Test. Dieser ist vom selben Format wie unsere Testfälle. Wenn er also nicht durchläuft, stimmt also evtl. grundsätzlich etwas mit eurer Implementierung nicht.

- Die Metriken messen wir mit dem Eclipse-Plugin *metrics 1.3.8* . Dabei gilt auch hier: Nur bei ausreichend implementierter Funktionalität kann auch die volle Erfüllung der Metriken erreicht werden.

- Abzugeben ist das Projekt als exportiertes eclipse-Projekt im .zip-Format. Benennt auch das Projekt um, so dass es den Gruppennamen enthält. Weitere Infos zur Entwicklungs- und Ausführungsumgebung:
  - Java: Mindestens JDK 11 (JRE reicht für Tomcat nicht aus). Im Projekt ist JDK 11-Kompatibilität eingestellt. Falls ihr mit einer neueren Java-Version kompiliert solltet ihr trotzdem für Java 11 bauen. Das Emma-Plugin unterstützt Java 15 noch nicht.

  - Eclipse in der J2EE-Version (Vorgabe getestet mit 2020-09, lief aber auch schon mit früheren Versionen).
  - Tomcat wurde von uns bisher mit Version 9 `https://tomcat.apache.org/download-90.cgi` verwendet.
  - Eine Anleitung und Tipps zur Einrichtung liegt dem Hausaufgabenblatt in der Datei `J2EE_Manual.pdf` auf ISIS bei.
  - Es kann sein, dass das Projekt nicht in jedem Browser läuft. Firefox sollte gehen, Javascript sollte dabei aktiviert sein. Der interne Browser von Eclipse geht nicht. Bitte beachtet, dass es für Tests mit mehreren Spielern nicht reicht verschiedene Tabs aufzumachen. Benutzt mehrere Browser, einen Spieler im Private Browsing oder eben verschiedene Computer (wofür das ja eigentlich gemacht ist).
  - Die direkte Verwendung von Code aus dem Internet ist nicht erlaubt und wird als Plagiat bewertet. Das Gleiche gilt für gruppenübergreifende Plagiate. Sollte wir so etwas bei der Korrektur sehen, wird die Modulprüfung bei allen beteiligten Gruppenmitgliedern als "Nicht bestanden" gewertet. Das gilt natürlich für alle Teile der Abgabe.

- Um Spielfelder zu visualisieren könnt ihr die GUI Testseite verwenden. Ihr findet sie bei gestartetem Server unter `http://localhost:8080/GameServer/Crazyhouse_Board.html`. Das anzuzeigende Spielbrett könnt ihr als JavaScript Parameter übergeben, zum Beispiel so:
  `http://localhost:8080/GameServer/CrazyHouse_Board.html?rnbQ2Q1/pppp3p/6k1/8/1P6/8/Pn1pPKPP/RNB2BNR/BPQRppq`

- Effizienter, kompakter Code benötigt meist weniger Testfälle. Falls zu viele Testfälle erstellt werden müssen, könnten geeignete Refactorings helfen.

2. CrazyHouse-Bot in Haskell (8 Punkte)

Ziel dieser Aufgabe ist es, einen Bot für Crazyhouse in Haskell zu entwickeln.
Auf ISIS findet Ihr eine Vorgabe zur Entwicklung eurer Lösung. Es gilt:

- Der Name des Moduls (`CrazyhouseBot`) **darf nicht verändert werden**.
- Alle Bibliotheken, die Haskell Platform mitliefert, können verwendet werden (Hinweis: Die Aufgabe lässt sich ohne weitere Imports lösen). Netzwerkzugriffe und die Einbettung von anderen Programmiersprachen sind nicht erlaubt. Deklarationen und Definitionen der `Util`-Funktionen **dürfen nicht verändert werden**.
- Die Signaturen und Namen der Funktionen `getMove` und `listMoves` **dürfen nicht verändert werden** (sie dienen der späteren Bewertung eurer Abgabe).

Eure Aufgabe besteht darin, die `getMove` und `listMoves` Funktionen sinnvoll zu vervollständigen. Beide erhalten als Argumente eine als String codierte Spielsituation, die sich aus dem Spielbrett (siehe Notation), einem Leerzeichen und dem nächsten Spieler ("b" für schwarz und "w" für weiß) in dieser Reihenfolge zusammensetzt. Ihr könnt davon ausgehen, dass ihr nur valide Strings erhaltet, die tatsächlich mögliche Spielsituationen codieren. Der Beispielzustand aus der Aufgabe 1 wäre wie folgt dargestellt, wenn der weiße Spieler den nächsten Zug bestimmen soll:
`"rnbQ2Q1/pppp3p/6k1/8/1P6/8/Pn1pPKPP/RNB2BNR/BPQRppq w"`

Die `listMoves`-Funktion soll eine Liste aller in der Situation möglichen Spielzüge berechnen und als String gemäß der Notation ausgeben. Die vollen Punkte erhaltet Ihr nur, wenn in allen getesteten Spielsituationen **alle regelkonformen Züge** korrekt berechnet werden. Dabei dürfen Züge **nicht mehrfach** vorkommen.

Die `getMove`-Funktion wählt aus den möglichen Spielzügen einen aus und gibt nur diesen zurück. Das Auswählen geschieht nach eurer eigenen Strategie. Diese Funktion dient der Implementierung eines ausführbaren Bots für manuelle Tests mit der Java-Implementierung. Die Implementierung dieser Funktion wird **nicht bewertet**. Wir werden außerdem ein Turnier durchführen mit den Bots derjenigen Gruppen, die hier mehr Zeit investiert haben und gerne teilnehmen wollen. Das Turnier wird nach dem zweiten Test durchgeführt, und die Bots für das Turnier dürfen auch noch nach der Abgabe weiterentwickelt werden.

Eure Lösungen werden mit HUnit automatisch getestet. In der Vorgabe findet Ihr außerdem eine Test-Datei, die eure Ausgaben auf korrekte Formatierung überprüft. Bitte reicht nur Abgaben mit korrekter Formatierung ein, da wir sonst keine Punkte vergeben können! Ihr startet die Tests, indem ihr das `Format`-Modul öffnet und `main` ausführt. Voraussetzung ist, dass Ihr HUnit[6] korrekt installiert habt.
Für die Abgabe ist nur die `CrazyhouseBot.hs` Datei interessant. Euer Bot wird unabhängig von eurem Server getestet.

[6] `https://github.com/hspec/HUnit`