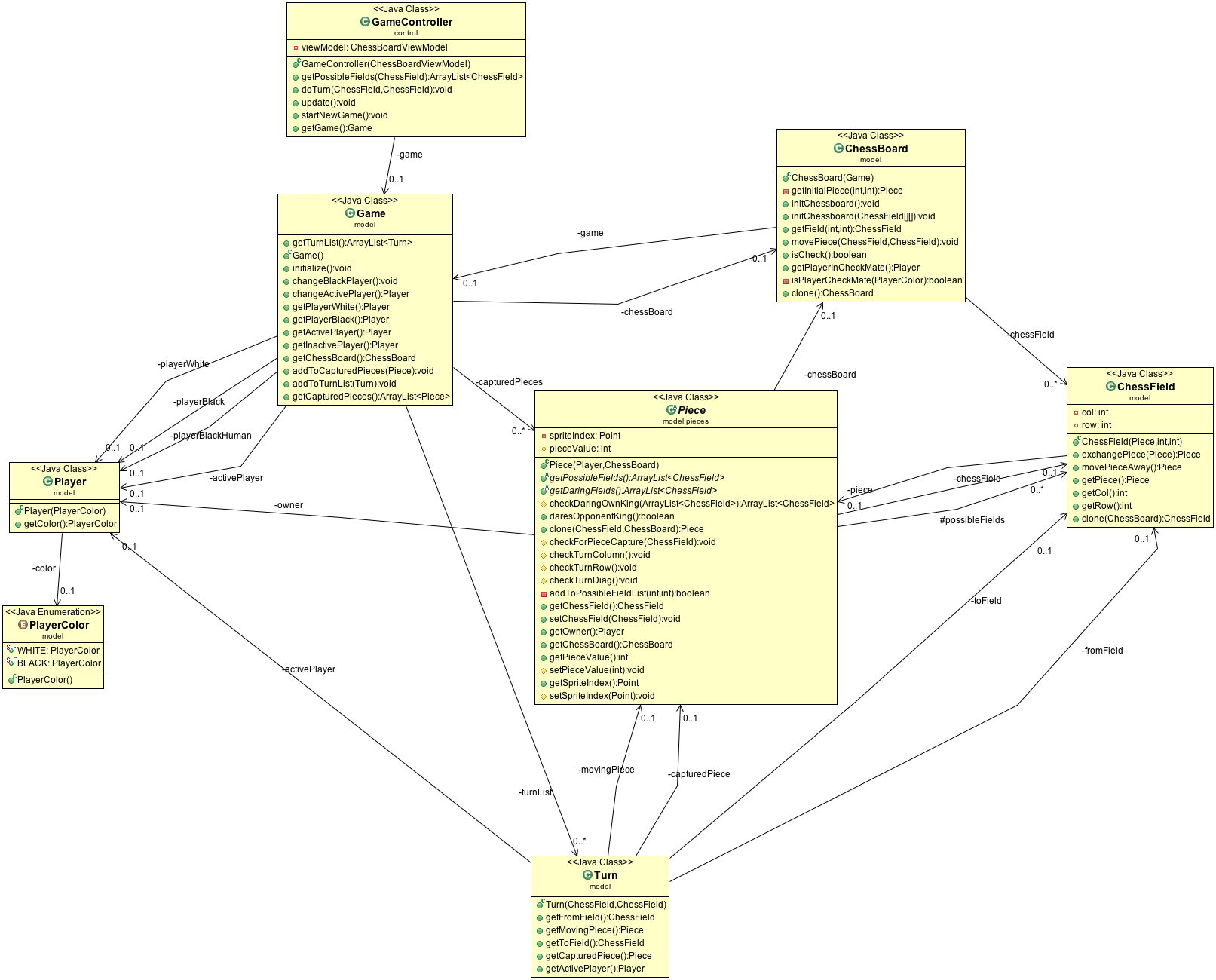
Schlussbericht

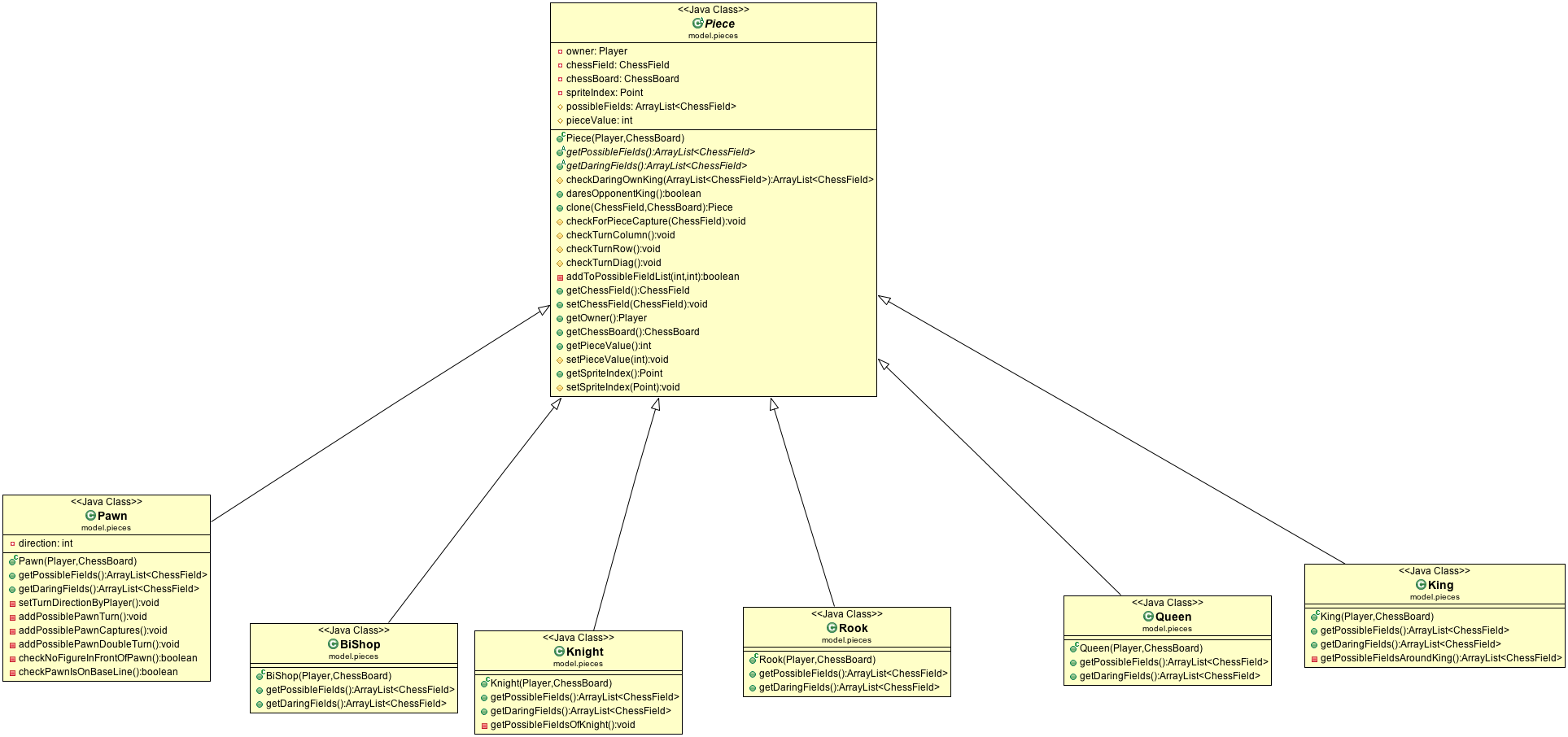
## Projektmanagement

|  |  |  |  |
| --- | --- | --- | --- |
|  | Benjamin Hohl | BH |  |
|  | Florian Bosshard | FB |  |
|  | Nadri Mamuti | NM |  |
|  | Sebastian Sprenger | SS |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| Bis und mit Meilenstein Projektskizze | |  |  |
| Mitarbeiter | Bezeichnung | Soll [h] | Ist [h] |
| Gruppe | Projektskizze | 8 | 8 |
| Gruppe | Projektskizze Präsentation | 3 | 4 |
| NM | UML für Eclipse einrichten | 1 | 2 |
| SS | Github für Eclipse einrichten | 2 | 2 |
| FB & BH | Eclipse Projekt aufsetzen | 1 | 1 |
| SS | Java Framework Slik anschauen | 2 | 1 |
|  |  |  |  |
| Total |  | 17 | 18 |
|  |  |  |  |
|  |  |  |  |
| Bis und mit Meilenstein Elaboration | |  |  |
| Mitarbeiter | Bezeichnung | Soll [h] | Ist [h] |
| NM | Projektmanagement erstellen | 1 | 1 |
| NM | Anwendungsfälle schreiben | 4 | 3 |
| SS | eine erste Architektur | 2 | 2 |
| BH | zusätzliche Spezifikationen | 2 | 2 |
| NM | Anwendugsfalldiagramm | 1 | 1 |
| FB | System-Sequenzdiagramm | 2 | 2 |
| FB | Systemoperationen | 0.5 | 1 |
| SS | Domänenmodell | 4 | 3 |
| FB | Glossar | 1 | 1 |
| SS | Software Codierung | 4 | 4 |
|  |  |  |  |
| Total |  | 21.5 | 20 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
| Bis und mit Meilenstein Design | |  |  |
| Mitarbeiter | Bezeichnung | Soll [h] | Ist [h] |
| NM | Projektmanagement erstellen | 1 | 1 |
| NM | definitive Architektur | 2 | 1.5 |
| FB | Design-Klassendiagramm | 1 | 1.5 |
| SS / FB | Klassenverantwortlichkeiten | 2 | 2 |
| FB | Zusammenarbeitsdiagramme | 2 | 2 |
| SS | GUI-Design | 3 | 4 |
| FB | Glossar | 1 | 0.5 |
| alle | Software Codierung | 10 | 10 |
|  |  |  |  |
| Total |  | 22 | 22.5 |
|  |  |  |  |
|  |  |  |  |
| Bis und mit Meilenstein Schlusspräsentation | | |  |
| Mitarbeiter | Bezeichnung | Soll [h] | Ist [h] |
| NM | Projektmanagement erstellen | 1 | 1 |
| FB | Klassendiagramm | 1 | 1 |
| SS FB | Code | 15 | 17 |
| BH | Testbericht | 3 | 3 |
| NM | Bedienungsanleitung | 5 | 3 |
| NM | Zusammenfassung | 5 | 3 |
| BH | Abschlusspräsentation | 3 | 3 |
|  |  |  |  |
| Total |  | 33 | 31 |
|  |  |  |  |
|  |  |  |  |
| Gesamt Total |  | 93.5 | 91.5 |

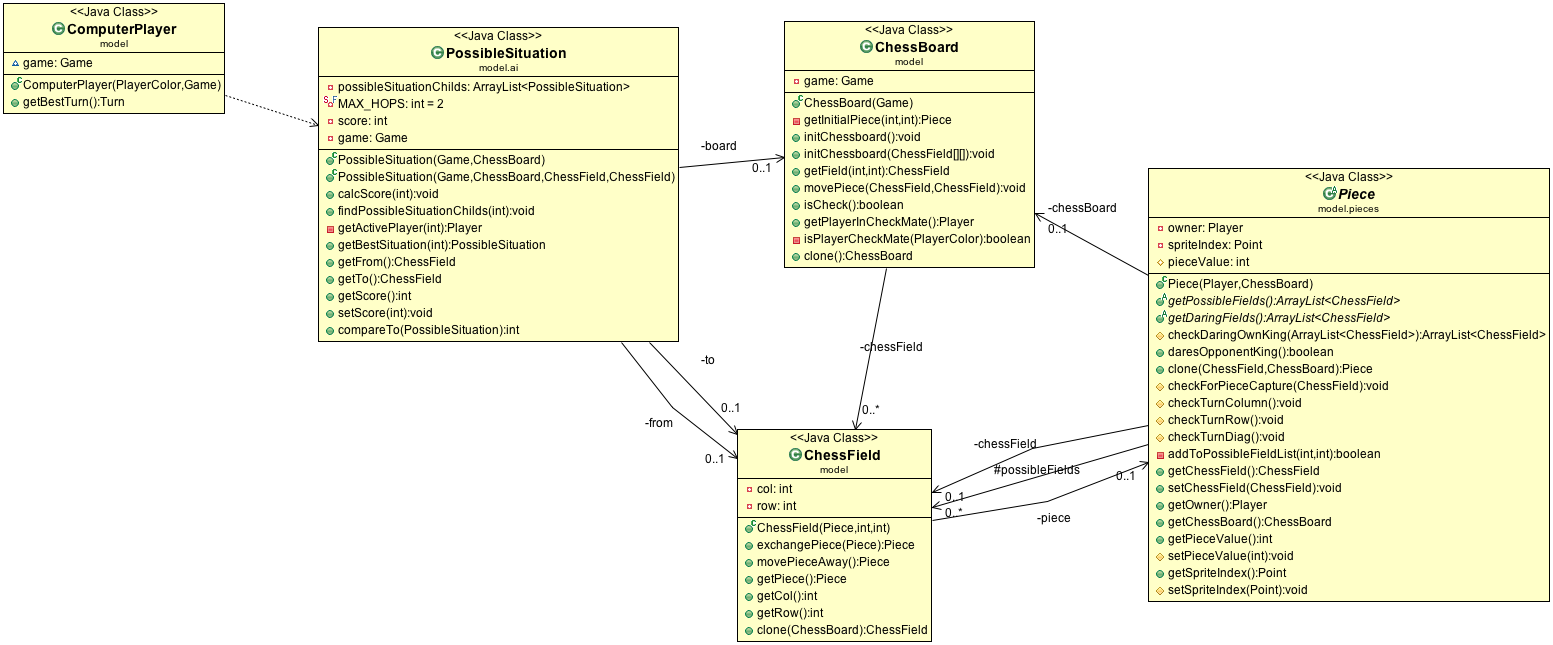
## Klassendiagramm



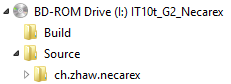
### Figuren (Pieces)



### Künstliche Intelligenz



## Code

Der Code und der letzte Build befinden sich auf der CD:

* Das Eclipse Projekt ch.zhaw.necarex befindet sich im Ordner Source und enthält den ganzen Code sowie alle benötigten Abhängigkeiten.
* Die Ausführbare Datei necarex.jar befindet sich im Ordner Build.
* Das JavaDoc kann hier gefunden werden: Source\ch.zhaw.necarex\doc

## Macintosh HD:Users:florian:Desktop:Bildschirmfoto 2012-12-09 um 18.09.32.pngTestbericht

### Test der Figuren mittels JUnit

Die Gangart Figuren wurden zu einem Zeitpunkt erstellt, an dem noch kein GUI vorhanden war. So wurden zuerst Unit-Tests erstellt und die Funktionalität danach entwickelt. Die Unit-Tests sind in der Klasse *PieceTester* abgelegt.

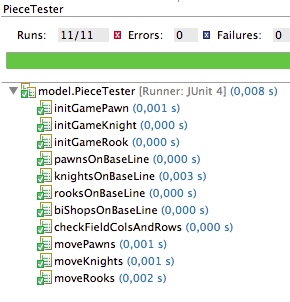
Zu Beginn des Testlaufs wird ein neues Game erzeugt. Danach laufen folgende Methoden nacheinander ab:

|  |  |
| --- | --- |
| Methode | Beschreibung |
| initGamePawn | Prüft, ob bei der Grundaufstellung die Bauern am richtigen Ort stehen. |
| initGameKnight | Prüft die Grundaufstellung der Springer |
| initGameRock | Prüft die Grundaufstellung der Türme |
| pawnsOnBaseLine | Prüft die Zugmöglichkeiten aus der Startposition aller Bauern |
| knightsOnBaseLine | Prüft die Zugmöglichkeiten aus der Startposition des Springers |
| rooksOnBaseLine | Prüfung, dass Türme keine Zugsmöglichkeiten haben in der Grundposition |
| checkFieldColsAndRows | Im ChessBoard hat es ein zweidiemensionales Array aller Felder. Zusätzlich weiss jedes Feld, auf welchem Feld es steht. Dies muss übereinstimmen. |
| movePawns | Verschiedene Bauernzüge und Kontrollen |
| moveKnights | Verschiedene Züge mit dem Springer und Kontrollen |
| moveRooks | Verschiedene Züge mit dem Turm und Kontrolle |

In der folgenden Auflistung die Züge und Kontrollen der movePawns. Die entsprechenden Methoden der anderen Figuren sind ähnlich aufgebaut.

* Zug des weissen Bauern B2 nach B4
* Prüfung, ob auf B4 ein Bauer steht
* Prüfung, ob auf B2 kein Bauer steht
* Prüfung, ob sich in der Zugliste ein Zug befindet
* Prüfungen, ob alle Attribute des Turn richtig erfasst sind
* Zug von C7 nach C5
* Prüfung, ob auf C5 ein Bauer steht
* Prüfung, ob auf C7 kein Bauer steht
* Zug B4 x C5 (Schlagen des schwarzen Bauerns)
* Prüfung, ob auf C5 ein weisser Bauer steht
* Prüfung, im Turn das Attribut capturedPieces korrekt gesetzt ist.
* In den capturedPieces der Klasse game ist der schwarze Bauer drin

Hier die Bestätigung von Eclipse, dass alle Testfälle erfolgreich durchgeführt wurden.



Weitere Unit-Tests neben den grundsätzlichen Zugmöglichkeiten der Figuren wurden aus Zeitgründen nicht erstellt.

Wichtig wären Tests betreffend spezifischer Situation (Schachsituationen, Schach-Matt, Patt). Diese sind relativ aufwändig zu schreiben, weil zuerst eine entsprechende Schachsituation aufgebaut werden muss. Die Klasse ChessBoard ist dafür vorbereitet, von einer Testklasse eine vordefinierte Situation zu erhalten.

Bei der künstlichen Intelligenz ist es ebenfalls schwierig, aussagekräftige Testfälle zu definieren, da es unterschiedliche Möglichkeiten für einen Spielverlauf gibt.

### Integrationstests

Im Folgenden werden einige Integrationstests beschrieben, die das System als Ganzes betrachten.

|  |  |
| --- | --- |
| Testfall 1a Mögliche Felder darstellen | |
| Zu testende Funktionalität | Beim Klick auf eine Figur werden die möglichen Felder berechnet und farblich dargestellt |
| Vorgehen | Klick auf Bauer auf dem Feld E2. |
| Resultat SOLL | Die Felder E3 und E4 sind markiert. |
| Resultat IST | Die Felder E3 und E4 sind markiert. |
| Erfolgreich? | JA |

|  |  |
| --- | --- |
| Testfall 1b Mögliche Felder darstellen | |
| Zu testende Funktionalität | Beim Klick auf eine Figur werden die möglichen Felder berechnet und farblich dargestellt |
| Vorgehen | Klick auf Turm auf dem Feld A1. |
| Resultat SOLL | Keine Felder sind markiert. |
| Resultat IST | Keine Felder sind markiert |
| Erfolgreich? | JA |

|  |  |
| --- | --- |
| Testfall 1c Mögliche Felder darstellen | |
| Zu testende Funktionalität | Beim Klick auf eine Figur werden die möglichen Felder berechnet und farblich dargestellt |
| Vorgehen | Folgende Züge wird gespielt:   1. E2 – E4 E7-E5 2. Dd1 – h5   Nun versucht Schwarz den Zug F7-F6 durchzuführen. Dies ist nicht erlaubt, da nach diesem Zug der schwarze König im Schach steht. |
| Resultat SOLL | Keine Felder sind markiert. |
| Resultat IST | Keine Felder sind markiert  Macintosh HD:Users:florian:Desktop:Bildschirmfoto 2012-12-09 um 18.49.29.png |
| Erfolgreich? | JA |

|  |  |
| --- | --- |
| Testfall 2 Schachpartie Mensch gegen Mensch durchspielen | |
| Zu testende Funktionalität | Zwei Menschen sollen ein Spiel gegeneinander durchspielen können. |
| Vorgehen | Gegenspieler Mensch auswählen und folgende Kurzpartie durchspielen   1. E2 – E4 E7-E5 2. Lf1-c4 A7-A6 3. Dd1-h5 H7-H6 4. Dd1-f7 Schach Matt |
| Resultat SOLL | Es wird angezeigt, dass das Spiel beendet ist. Schwarz kann nicht mehr spielen. |
| Resultat IST | Macintosh HD:Users:florian:Desktop:Bildschirmfoto 2012-12-10 um 09.17.14.png  Rechts unten wird der Sieger dargestellt. Schwarz hat keine Zugmöglichkeit mehr |
| Erfolgreich? | JA |

|  |  |
| --- | --- |
| Testfall 3 Partie Mensch gegen Computer durchspielen | |
| Zu testende Funktionalität | Ein menschlicher Spieler kann ein korrektes Spiel gegen den Computer durchspielen. |
| Vorgehen | Nach dem Start der Applikation wird als Gegenspieler der Computerspieler ausgewählt |
| Resultat SOLL | Nach einem weissen Zug spielt schwarz einen gültigen Zug. |
| Resultat IST | Nach einem weissen Zug spielt schwarz einen gültigen Zug. Er versucht dabei die Situation für sich zu optimieren (Ziele: keine eigenen Figurenverluste, Gegner verliert Figuren, gegnerischen König ins Schach stellen). Dies wird nach jedem Zug durchgeführt. |
| Erfolgreich? | JA |

|  |  |
| --- | --- |
| Testfall 3 Während der Partie den Gegenspieler wechseln | |
| Zu testende Funktionalität | Während des Spiels kann der Benutzer vom Mensch zum Computer wechseln und umgekehrt. |
| Vorgehen | Weiss zieht e2-e4. Danach wird gewechselt auf den Computerspieler. |
| Resultat SOLL | Schwarz zieht |
| Resultat IST | Schwarz zieht nicht. Der Auslöser für den Computerspieler ist direkt nach dem weissen Zug angehängt. |
| Erfolgreich? | NEIN |
| Konsequenz | Soll in der Benutzerdokumentation dokumentiert werden.  Soll in einem zukünftigen Release als Funktion eingebaut werden. |

## Zusammenfassung

### Funktionalität

Das Schachspiel „necaREx“ soll in Schachvereinen und Schachkursen als effiziente Lernsoftware eingesetzt werden. Dies ist durchaus denkbar, da „necaREx“, mit Ausnahme von ein paar speziellen Funktionen, mit den herkömmlichen Schachregeln zu spielen ist.  
  
Durch das schlichte und intuitive Design ist „necaREx“ einfach und schnell zu erlernen. Dies war einer der wichtigsten Aspekte, welchen wir erreichen wollten, denn damit das Spiel Spass macht und man etwas dabei lernen kann, darf es nicht zu komplex aufgebaut sein. Wir denken diese Ziel erreicht zu haben.  
  
Dank unserer übersichtlichen Benutzeroberfläche kann der Spieler zu jeder Zeit wichtige Informationen dem Spiel entnehmen. Dies ist vor allem für die Übersichtlichkeit und für den Lernerfolg wichtig.

### Fehler

Fehler als solche haben wir nicht gefunden, jedoch haben wir eine Weile gebraucht, bis wir uns entscheiden konnten, wie wir das Problem mit der Umstellung zwischen „Mensch“ und „Computer“ lösen wollten. Es gab verschiedene Ideen, so wollten wir zuerst bei einem Zustandswechsel des Gegenspielers gleich noch das Spiel neu starten, um Probleme zu vermeiden. Zum Schluss konnten wir eine Lösung finden, bei der man den Gegenspieler „on the fly“ wechseln kann, wobei man darauf achten muss, zu welchem Zeitpunkt man den Gegenspieler wechselt.

### Funktionen

Es konnten glücklicherweise alle vorgesehenen Funktionen implementiert werden, jedoch konnte nicht alle möglichen Erweiterungen realisiert werden.

Erweiterungen / Ideen die nicht realisiert werden konnten

Folgende Erweiterungen waren bereits in den Anforderungen als nicht zwingend definiert und konnten mangels Zeit nicht implementiert werden. Sie bieten Ideen für eine Weiterentwicklung der Applikation.

* Erreicht ein Bauer das obere Spielfeldende, so wählt der Spieler die einzusetzende Figur aus.
* Rochade
* Schlagen „en passant“
* Verschiedene Computer-Schwierigkeitsstufen
* Spiel zweier Menschen über das Netzwerk

### Rückblick auf SEPS 2012 Gruppe 2

Für jedes Mitglied der Gruppe war es eine sehr lehrreiche und interessante Erfahrung, welche wir in dieser Gruppenarbeit und Konstellation erleben durften. Schlussendlich konnte jedes Teammitglied in allen Teilbereichen, vom Projektablauf bis zur Programmierung viel zur Gruppenarbeit beitragen.

Obwohl die einzelnen Gruppenmitglieder vor dieser Teamarbeit nicht viel miteinander zu tun hatten, hat die Arbeit im Team äusserst gut funktioniert. Man hat sich gegenseitig abgesprochen und respektiert und darauf geachtet, dass kein Mitglied zu kurz kommt oder zu viel machen musste.

Spezielle positive Erfahrungen gab es in der Gruppe durch den Informationsaustausch in der Programmierung und im Projektmanagement.

An negative Erfahrungen können wir uns nicht erinnern.