



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 2022/2023

Prof. Dr. Sabine Glesner
Milko Monecke
Simon Schwan

Hausaufgabenblatt UML/OCL

Ausgabe: 23.1.

Beispiellösung

Sie haben den Auftrag erhalten, ein Online-Spiel-Portal zu entwickeln. Dieses Spiel-Portal, der *GameServer*, soll flexibel für verschiedene Spiele verwendet werden können. Folgende Anforderungen wurden für das Portal definiert:

Das System soll es Benutzys¹ (*User*) ermöglichen, gegeneinander oder gegen Bots beliebige Spiele zu spielen. Dafür soll eine Server-Anwendung sämtliche Funktionalität bereitstellen, auf die Benutzys über ein Web-Interface zugreifen können. Jedes Benutzys muss sich am System registrieren bevor es sich anmelden kann, dazu muss es einen beliebigen eindeutigen Benutzynamen, einen Anzeigenamen und ein Passwort angeben. Nach dem Registrieren erfolgt eine erste Anmeldung automatisch.

Ein Benutzys kann ein neues Spiel beginnen. Dabei wählt es aus, wie viele seiner Gegenspiels durch Bots besetzt werden. Die minimale und maximale Anzahl Spiels für ein Spiel wird dabei durch das *konkrete Spiel* vorgegeben. Nach der Erstellung des Spiels werden die Bots als Spiels direkt zugewiesen, und auf menschliche Gegenspiels wird gewartet. Sobald genügend Spiels zugewiesen sind, wird das Spiel gestartet. Das Spiely, das das Spiel erstellt hat, ist zuerst am Zug.

Ein Benutzys kann außerdem einem Spiel beitreten. Wenn zu diesem Zeitpunkt bereits Spiele erstellt wurden, die noch auf Spiels warten, wird es dem Spiel, das am längsten wartet, als Spiels zugewiesen. Sollte kein Spiel auf Spiels warten, wird die Anfrage mit einer entsprechenden Meldung an das Benutzys beendet.

¹Entgendern nach Phettberg. Eine Erklärung und Begründung dazu gibt es auf der ISIS-Seite zum Kurs.

Ein Benutzzy kann in maximal acht Spielen, die noch nicht beendet sind, als Spiely zugewiesen sein. Es kann sich jederzeit die Spiele auflisten, bei denen es mitspielt, und sich zu einem Spiel anzeigen lassen, welches der aktuelle Spielstand (Figuren auf dem Brett) ist, wer an der Reihe ist, und welchen Status das Spiel hat. Der Status kann nach außen hin folgende Werte annehmen: „Warten auf Spiely(s)“, „Aktiv“, „Beendet“, „Beendet - Unentschieden“, „Beendet - Aufgegeben“. Intern kann dies auch detaillierter modelliert werden.

Sobald das Spiely an der Reihe ist, hat es folgende Optionen: Es kann einen Zug versuchen. Dieser Zug wird dem Spiel übermittelt, welches ihn prüft und ggf. durchführt. Sollte dieser Zug das Spiel beenden, wird das Spiel als beendet markiert und die Spielys entsprechend als Gewinny bzw. Verliery markiert. Das Spiely kann außerdem ein Unentschieden vorschlagen, welches für es für dieses Spiel vermerkt wird. Wenn alle Spielys dies getan haben, wird das Spiel als Unentschieden beendet. Ein Spiely kann außerdem aufgeben, in diesem Fall wird er zum Verliery und alle anderen Spielys haben gewonnen. Der Status ist dann „Beendet - Aufgegeben“.

Das Benutzzy kann sich jederzeit eine Statistik anzeigen lassen, die die folgenden Werte enthält:

- Anzahl gespielter Spiele (Gewonnen/Verloren/Unentschieden)
- Anteil gewonnener/verlorener/unentschiedener Spiele
- durchschnittliche Anzahl Spielzüge

Es sollen alle Spielzüge im System hinterlegt sein, um diese Statistiken jederzeit aktualisieren und den Verlauf grafisch darstellen zu können.

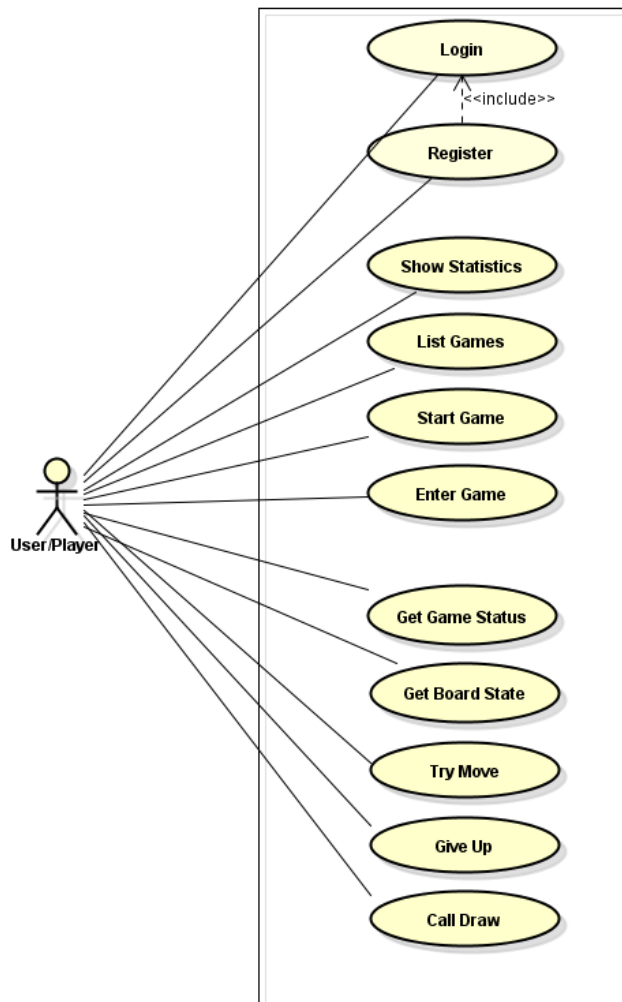
Das Web-Interface ermöglicht die Kommunikation des Benutzzy mit dem System. Es implementiert dabei keine eigene Funktionalität, sondern reicht alle Anfragen des Benutzzy direkt an das System weiter. Es kann davon ausgegangen werden, dass alle Daten, z.B. Spielstand, Züge und ggf. Textmeldungen als Antwort auf Aktionen als Strings ausgetauscht werden können.

Zu beachten ist, dass diese Modellierung nur eine abstrakte Sicht auf Spiel und Spielys enthält - das tatsächliche Prüfen und Durchführen einer Aktion im Spiel und das automatische Beenden nach einem Zug wird vom konkreten Spiel implementiert, das hier nicht beschrieben ist. Teile des Modells sind also abstrakt zu halten, damit sie später durch konkrete Instanzen implementiert werden können.

1. Use-Case-Diagramm

Modelliert die Anwendungsfälle des Systems mit einem Use-Case-Diagramm.

– Lösung Anfang –



Anmerkungen zur Beispiellösung:

- Im Text werden Benutzys und Spielys als externe Aktoren genannt. Daher wäre es auch ok die Use-Cases auf zwei Aktoren aufzuteilen. Da dies aber die gleiche Person ist haben wir uns für nur einen Aktor entschieden.
- Anmelden und Registrieren sind Voraussetzung für alle anderen Use-Cases. Man könnte auch davon ausgehen, dass die Umgebung diese Funktionalität bereitstellt und dies nicht explizit modellieren. Da es aber im Text explizit erwähnt wird haben wir uns dafür entschieden.

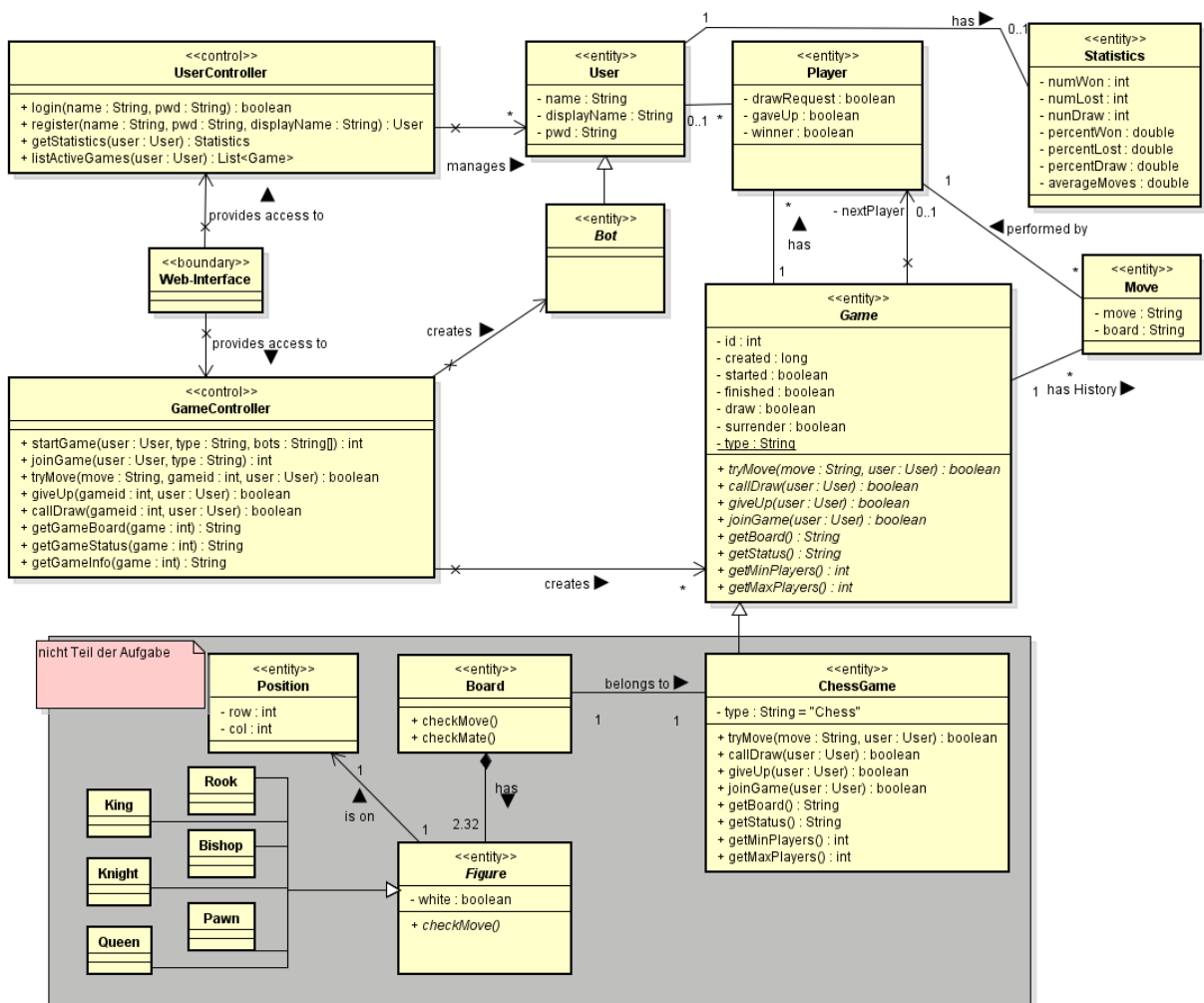
– Lösung Ende –

2. Klassendiagramm

Erstellt für das System ein Klassendiagramm, in dem alle Anforderungen bzgl. der Struktur, die aus dem Text hervorgehen, enthalten sind. Dabei ist folgendes zu beachten:

- Die in Aufgabe 1 identifizierten Use-Cases müssen als Operationen mit notwendigen Parametern auftauchen. Von weiteren Operationen kann abstrahiert werden, sofern sie nicht relevant sind.
- Assoziationen sollten über einen Namen und eine Leserichtung verfügen. Multiplizitäten und Navigationsrichtungen sind erforderlich sofern es aus den Anforderungen hervorgeht.
- Alle Klassen müssen über eindeutige Stereotype entsprechend dem Entity-Control-Boundary Pattern aus der Vorlesung verfügen.

– Lösung Anfang –



Anmerkungen zur Beispiellösung:

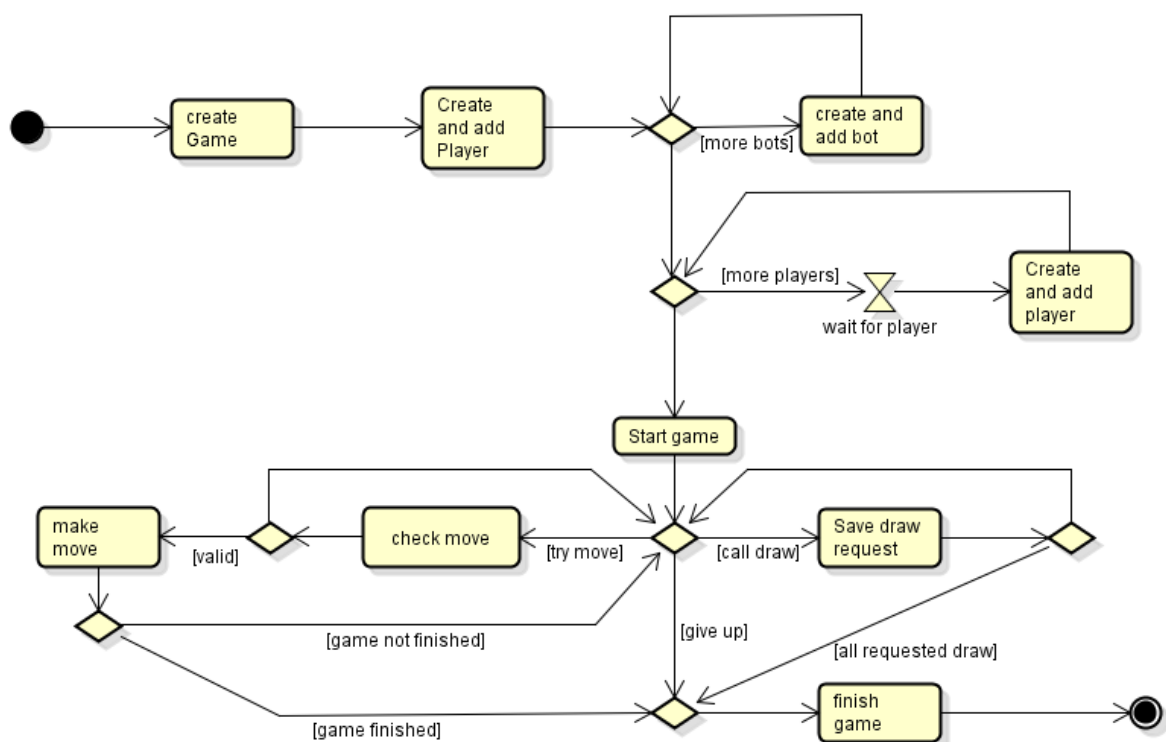
- Es gibt laut Text nur einen Actor (Benutzy/Spiely) und eine Boundary (Web-Schnittstelle), daher wäre auch ein Controller ok. Wir haben uns der Übersichtlichkeit halber dafür entschieden, die Use-Cases nach “Thema” in zwei Controller-Klassen zu trennen: Die zur generellen Verwaltung und solche, die Zugriffe auf ein bestimmtes Spiel beinhalten.
- Um bei einem Spiel mitzuspielen, könnte **Bot** auch von **Player** erben. Die hier gewählte Modellierung des Bots als *Benutzy* ermöglicht es aber, den Bot in allen Situationen wie einen gleichwertiges menschliches Spiely zu behandeln. Z.B. erhält der Bot so einen Namen, der im Web-Interface als Gegenspiely mit aufgelistet werden kann. Wie und wann der Bot dann seine Züge ausführt ist nicht Teil der Beschreibung bzw. der Beispiel-Modellierung.
- Funktionen, Attribute und Parameter sind nicht alle direkt im Text wiederzufinden, sondern dokumentieren eine spätere Implementierung.
- Die Statistik könnte theoretisch auch bei jeder Anfrage neu berechnet werden und muss nicht statisch “existieren” wie hier im Klassenmodell. An dieser Stelle dient es dazu, die im Text beschriebenen Anforderungen im Modell zu repräsentieren.

– Lösung Ende –

3. Aktivitätsdiagramm

Modelliert den Ablauf (Workflow) eines Spiels in einem Aktivitätsdiagramm von seiner Erstellung über den Start bis zum Ende. Außerdem sollen die möglichen Züge der Spiels modelliert werden. Von Details eines konkreten Spiels soll abstrahiert werden, ihr dürft dafür die vorgegebenen Aktionen „Zug prüfen“ und „Zug ausführen“ verwenden. Es kann davon ausgegangen werden, dass Benutzers angemeldet sind.

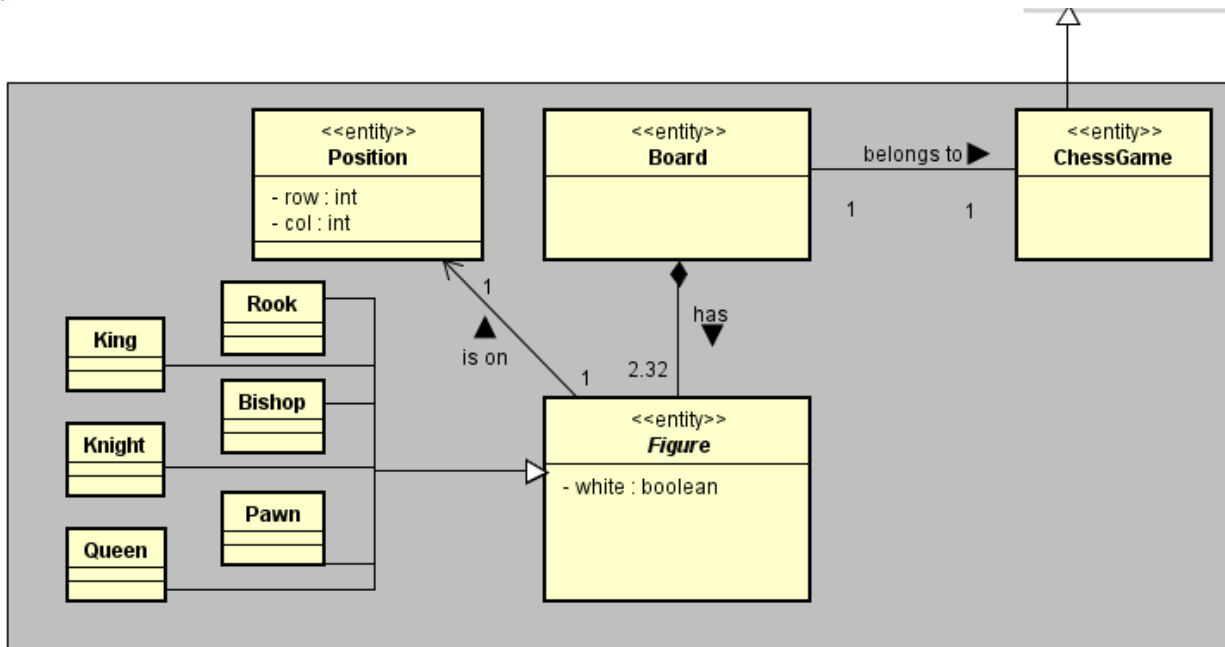
– Lösung Anfang –



– Lösung Ende –

4. Invarianten

Erweitert nun euer Modell um das konkrete Spiel „Schach“. Beispiel für das Klassendiagramm:



Präzisiert eure Modellierung formal mithilfe von folgenden Invarianten in OCL:

- Die Benutzys haben eindeutige Benutzykennungen.
- Zu jedem Spiel gehört mindestens ein Spiely. Zu jedem gestarteten Schachspiel gehören zwei Spielys.
- Kein Benutzys darf mehrere Spiely in einem Spiel kontrollieren.
- Benutzys dürfen nicht gleichzeitig in mehr als acht Spielen, die nicht beendet sind, teilnehmen.
- Die Anzahl der gewonnenen Spiele in der Statistik eines Benutzys entspricht den beendeten Spielen, für die dieses Benutzys tatsächlich als Gewinnny vermerkt ist.
- Bevor ein Schachspiel startet, stehen auf dem Spielfeld 16 Figuren jeder Farbe.
- Auf einer Position dürfen niemals zwei Figuren stehen.
- Wenn alle Spiely ein Unentschieden gefordert haben ist das Spiel als Unentschieden beendet.
- Zu langes Warten auf Zuweisung eines Gegenspielys soll verhindert werden: Es kann kein Spiel mehr als ein menschliches Spiely haben solange ein früher erstelltes Spiel noch auf Spielys wartet.

Hinweise:

- Wir empfehlen, alle OCL-Ausdrücke mit dem USE-OCL-Tool zu überprüfen (<http://sourceforge.net/projects/useocl/>).

– Lösung Anfang –

Anmerkungen zur Beispiellösung:

- Alle in diesem Dokument enthaltenen OCL-Lösungen sind auch in den mitgelieferten USE-Modellen zu finden. Zur Erinnerung: `.use` und `.soil`-Dateien lassen sich mit einem normalen Editor öffnen.

```
context UserController inv A5aUsersUnique:
    user.name->size() = user.name->asSet()->size()

context Game inv A5bGameHasPlayer:
    player->size() >= 1 and (self.oclIsTypeOf(ChessGame) and started)
    implies player->size() = 2

context Game inv A5cDifferentPlayers:
    player.user->size() = player.user->asSet()->size()

context User inv A5dMaxGames:
    player.game->select(not finished)->size() < 8

context Statistics inv A5eStatsCorrect:
    numWon = user.player->select(game.finished and winner)->size()

context ChessGame inv A5fStartBoardFigures:
    not self.started implies self.board.figure->size() = 32
    and self.board.figure->select(f|f.white)->size() = 16

context Board inv A5gOnlyOneFigureOnPos:
    let positions = figure.position->collect(row * 10 + col) in
    positions->asSet()->size() = positions->size()

context Game inv A5hDraw:
    player->forall(drawRequest) implies (finished and draw)

context Game inv A5iNoWaiting:
    player.user->select(oclIsTypeOf(User))->size() >= 2 implies
```



```
not Game.allInstances()-> exists(g| g.id <> self.id  
    and g.created < self.created and not g.started)
```

– Lösung Ende –

5. Contracts in OCL

Spezifiziert folgende Operationen mithilfe von Contracts in OCL. Berücksichtigt mögliche Fehlerfälle. Sofern dies für den Contract erforderlich ist könnt ihr davon ausgehen, dass das Benutzzy durch das Web-Interface am System angemeldet wurde und die Benutzzydaten als Objekt zur Verfügung stehen.

- a) Benutzzy registrieren
- b) Aufgeben
- c) Unentschieden vorschlagen
- d) Spiel starten
- e) Eine Funktion `calculateAverageMoves`, in der die durchschnittliche Anzahl der Züge aller beendeten Spiele eines Benutzzy ermittelt wird.

– Lösung Anfang –

```
context UserController ::
  register(uid:String, pwd:String, displayName:String): User
    pre: uid <> '' and pwd <> '' and displayName <> ''
    pre: not User.allInstances()->exists(uid=name)
    post: user->exists(u: User | u.name=uid and u.pwd=pwd and
      u.displayName=displayName and u.ocIsNew() and
      u.ocIsTypeOf(User) and result=u)

context GameController ::
  startGame(user: User, bots: Bag(String)): Integer
    pre: true
    post: game->exists(g | g.ocIsNew() and
      g.player.user->includes(user) and
      g.player.user->
        select(ocIsKindOf(Bot)).displayName = bots and
        result = g.id)

context GameController ::
  giveUp(gameid: Integer, u: User): Boolean
    pre: game->exists(id=gameid and not finished
      and player.user->includes(u))
    post: game->exists(id=gameid and finished and
      player->forAll(user=u implies (gaveUp and not winner)) and
```

```

        player→forall(user◊u implies (not gaveUp and winner)))
    and result=true

context GameController::
    callDraw(gameid:Integer, u:User): Boolean
        pre: game→exists(id=gameid and not finished and
            player.user→includes(u))
        post: let g:Game = game→any(id=gameid) in
            (g.player→forall(user=u implies drawRequest)) and
            ((g.player→forall(drawRequest)) implies
                g.finished and g.draw)
            and result=true

context User::
    calculateAverageMoves() : Integer
        pre: true
        post: let games = player.game→select(finished) in
            if games→size() = 0 then result=0
            else result= (games→iterate(g:Game; sum:Integer =0 |
                sum + g.move→size()) / games→size())
            endif

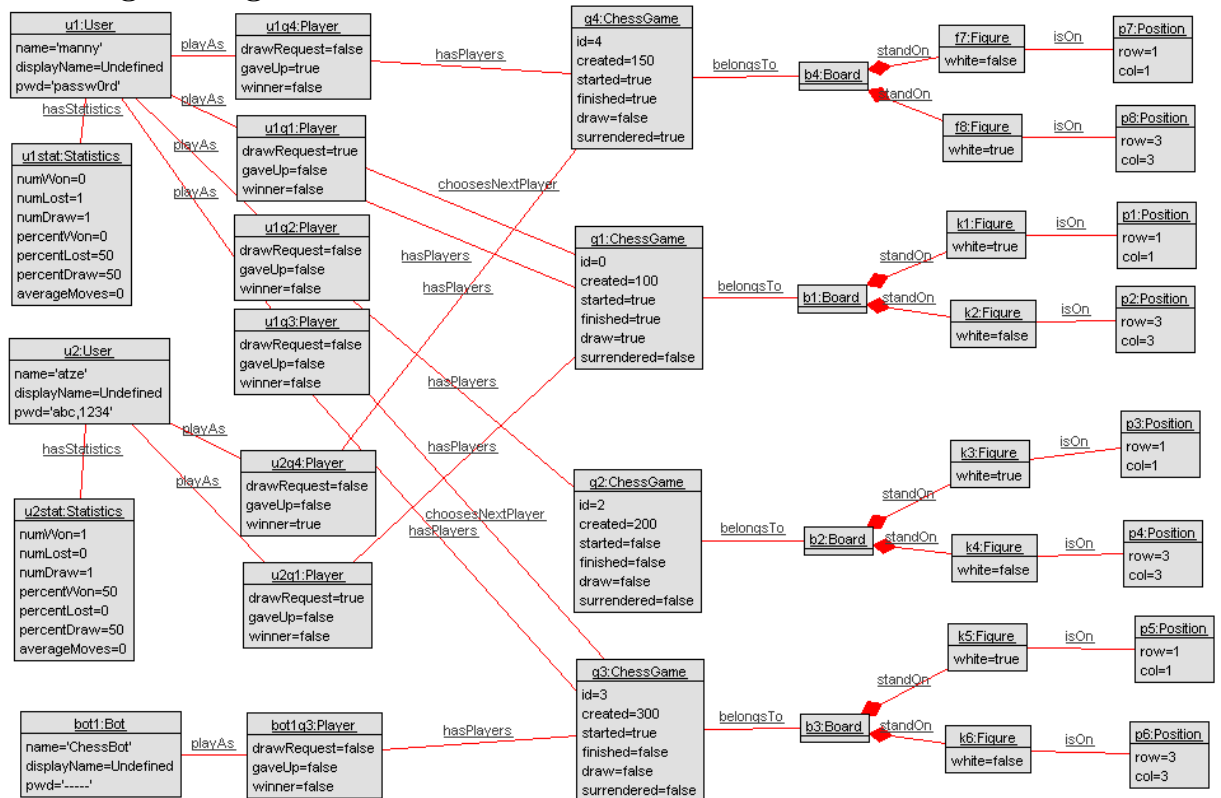
```

– Lösung Ende –

6. Objektdiagramm

Stellt einen Zustand des Systems in einem Objektdiagramm dar, der entsprechend eurem Klassendiagramm und den OCL-Invarianten gültig ist. Dabei soll mindestens ein Schachspiel auf Spielys warten und eines beendet sein. Um das Diagramm übersichtlich zu halten muss Invariante 5f nicht unbedingt erfüllt sein.

– Lösung Anfang –



– Lösung Ende –

7. Hoare Kalkül

Gegeben ist eine Implementierung der Operation `calculateAverageMoves` aus Aufgabe 6.e). Beweist mithilfe der Ableitungsregeln des Hoare Kalküls, dass das Programm nach Ausführung die Nachbedingung erfüllt (*partielle* Korrektheit). Führt außerdem einen Terminierungsbeweis für das Programm durch.

```

calculateAverageMoves(int[] moves):
P{moves.length >= 0}
    sum := 0;
    cnt := 0;
    erg := 0;

    while cnt < moves.length do
        sum := sum + moves[cnt];
        cnt := cnt + 1
    od;

    if moves.length > 0 then
        erg := sum / cnt
    else
        skip
    fi

```

$$Q\{(moves.length = 0 \wedge erg = 0) \vee (moves.length > 0 \wedge erg = \frac{\sum_{i=0}^{moves.length-1} moves[i]}{moves.length})\}$$

– Lösung Anfang –

```

calculateAverageMoves(int[] moves):
{moves.length >= 0}
    {moves.length >= 0 ∧ 0 = 0 ∧ 0 = 0 ∧ 0 = 0}
    sum := 0;
    {moves.length >= 0 ∧ sum = 0 ∧ 0 = 0 ∧ 0 = 0}
    cnt := 0;
    {moves.length >= 0 ∧ sum = 0 ∧ cnt = 0 ∧ 0 = 0}
    erg := 0;
    {moves.length >= 0 ∧ sum = 0 ∧ cnt = 0 ∧ erg = 0}
    {moves.length >= 0 ∧ sum =  $\sum_{i=0}^{0-1} moves[i]$  ∧ cnt = 0 ∧ erg = 0}
    {moves.length >= 0 ∧ sum =  $\sum_{i=0}^{cnt-1} moves[i]$  ∧ cnt = 0 ∧ erg = 0}
    {moves.length >= 0 ∧ sum =  $\sum_{i=0}^{cnt-1} moves[i]$  ∧ cnt <= moves.length ∧ erg = 0}
    while cnt < moves.length do
        {cnt < moves.length ∧ moves.length >= 0 ∧ sum =  $\sum_{i=0}^{cnt-1} moves[i]$  ∧ cnt <= moves.length ∧ erg = 0}
        {cnt < moves.length ∧ moves.length >= 0 ∧ sum =  $\sum_{i=0}^{cnt-1} moves[i]$  ∧ erg = 0}

```

```

{cnt < moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt moves[i] - moves[cnt] ∧ erg = 0}
{cnt < moves.length ∧ moves.length ≥ 0 ∧ sum + moves[cnt] = ∑i=0cnt moves[i] ∧ erg = 0}
sum := sum + moves[cnt];
{cnt < moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt moves[i] ∧ erg = 0}
{cnt + 1 ≤ moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt+1 moves[i] - moves[cnt + 1] ∧ erg = 0}
cnt := cnt + 1
{cnt ≤ moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt moves[i] - moves[cnt] ∧ erg = 0}
{cnt ≤ moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt-1 moves[i] ∧ erg = 0}
od;
{cnt ≥ moves.length ∧ cnt ≤ moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt-1 moves[i] ∧ erg = 0}
{cnt = moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0cnt-1 moves[i] ∧ erg = 0}
{cnt = moves.length ∧ moves.length ≥ 0 ∧ sum = ∑i=0moves.length-1 moves[i] ∧ erg = 0}
if moves.length > 0 then
  {moves.length > 0 ∧ moves.length ≥ 0 ∧ sum = ∑i=0moves.length-1 moves[i] ∧ cnt = moves.length ∧ erg = 0}
  {moves.length > 0 ∧ sum = ∑i=0moves.length-1 moves[i] ∧ cnt = moves.length ∧ erg = 0}
  {moves.length > 0 ∧ sum / cnt = ∑i=0moves.length-1 moves[i] / moves.length ∧ erg = 0}
  erg := sum / cnt
  {moves.length > 0 ∧ erg = ∑i=0moves.length-1 moves[i] / moves.length}
  {⊥ ∨ (moves.length > 0 ∧ erg = ∑i=0moves.length-1 moves[i] / moves.length)}
  {(moves.length = 0 ∧ erg = 0) ∨ (moves.length > 0 ∧ erg = ∑i=0moves.length-1 moves[i] / moves.length)}
else
  {moves.length ≤ 0 ∧ moves.length ≥ 0 ∧ sum = ∑i=0moves.length-1 moves[i] ∧ cnt = moves.length ∧ erg = 0}
  {moves.length = 0 ∧ sum = ∑i=0moves.length-1 moves[i] ∧ cnt = moves.length ∧ erg = 0}
  {moves.length = 0 ∧ erg = 0}
  skip
  {moves.length = 0 ∧ erg = 0}
  {(moves.length = 0 ∧ erg = 0) ∨ ⊥}
  {(moves.length = 0 ∧ erg = 0) ∨ (moves.length > 0 ∧ erg = ∑i=0moves.length-1 moves[i] / moves.length)}
fi
{(moves.length = 0 ∧ erg = 0) ∨ (moves.length > 0 ∧ erg = ∑i=0moves.length-1 moves[i] / moves.length)}

```

Terminierungsfunktion: moves.length - cnt

a) {cnt ≤ moves.length} ⇒ moves.length - cnt ≥ 0

b) while cnt < moves.length do
 {moves.length - cnt = m}
 ⇒ {moves.length - cnt < m + 1}
 ⇒ {moves.length - cnt - 1 < m}

```

    sum := sum + moves[cnt];
    {moves.length - cnt - 1 < m}
    ⇒ {moves.length - (cnt + 1) < m}
    cnt := cnt + 1
    {moves.length - cnt < m}
od;

```

– Lösung Ende –