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 2023/2024

Prof. Dr. Sabine Glesner
Julian Klein
Simon Schwan

Hausaufgabenblatt UML/OCL

Ausgabe: 30.1.

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 Benutzer:innen (*User*) ermöglichen, gegeneinander oder gegen Bots beliebige Spiele zu spielen. Dafür soll eine Server-Anwendung sämtliche Funktionalität bereit stellen, auf die Benutzys über ein Web-Interface zugreifen können. Benutzer:innen müssen sich am System registrieren, bevor sie sich anmelden können. Dazu müssen sie einen beliebigen eindeutigen Namen, einen Anzeigenamen und ein Passwort angeben. Nach dem Registrieren erfolgt eine erste Anmeldung automatisch.

Benutzer:innen können ein neues Spiel beginnen. Dabei wählt man aus, wie viele der Gegenspieler:innen durch Bots besetzt werden. Die minimale und maximale Anzahl von Spieler:innen für ein Spiel wird dabei durch das *konkrete Spiel* vorgegeben. Nach der Erstellung des Spiels werden die Bots als Spieler:innen direkt zugewiesen, und auf menschliche Gegenspieler:innen wird gewartet. Sobald genügend Spieler:innen zugewiesen sind, wird das Spiel gestartet. Die/die Spieler:in, die/der das Spiel erstellt hat, ist zuerst am Zug.

Ein:e Benutzer:in kann außerdem einem Spiel beitreten. Wenn zu diesem Zeitpunkt bereits Spiele erstellt wurden, die noch auf Spieler:innen warten, wird er/sie dem Spiel, das am längsten wartet, als Spieler:inn zugewiesen. Sollte kein Spiel auf Spieler:innen warten, wird die Anfrage mit einer entsprechenden Meldung an die Benutzer:innen beendet.

Ein:e Benutzer:in kann in maximal acht Spielen, die noch nicht beendet sind, als Spieler:in zugewiesen sein. Benutzer:innen können sich jederzeit die Spiele auflisten, bei denen

mitspielt wird, 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 Spieler:innen", ,,Aktiv", ,,Beendet", ,,Beendet - Unentschieden", ,,Beendet - Aufgegeben". Intern kann dies auch detaillierter modelliert werden.

Sobald Spieler:innen an der Reihe sind, gibt es folgende Optionen: Man 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 der/die Spieler:in entsprechend als Gewinner:in bzw. Verlierer:in markiert. Ein:e Spieler:in kann außerdem ein Unentschieden vorschlagen, welches für es für dieses Spiel vermerkt wird. Wenn alle Spieler:innen dies getan haben, wird das Spiel als Unentschieden beendet. Ein:e Spieler:in kann außerdem aufgeben, in diesem Fall wird er zur Verlierer:in und alle anderen Spieler:innen haben gewonnen. Der Status ist dann ,,Beendet - Aufgegeben".

Der/die Benutzer:in 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.

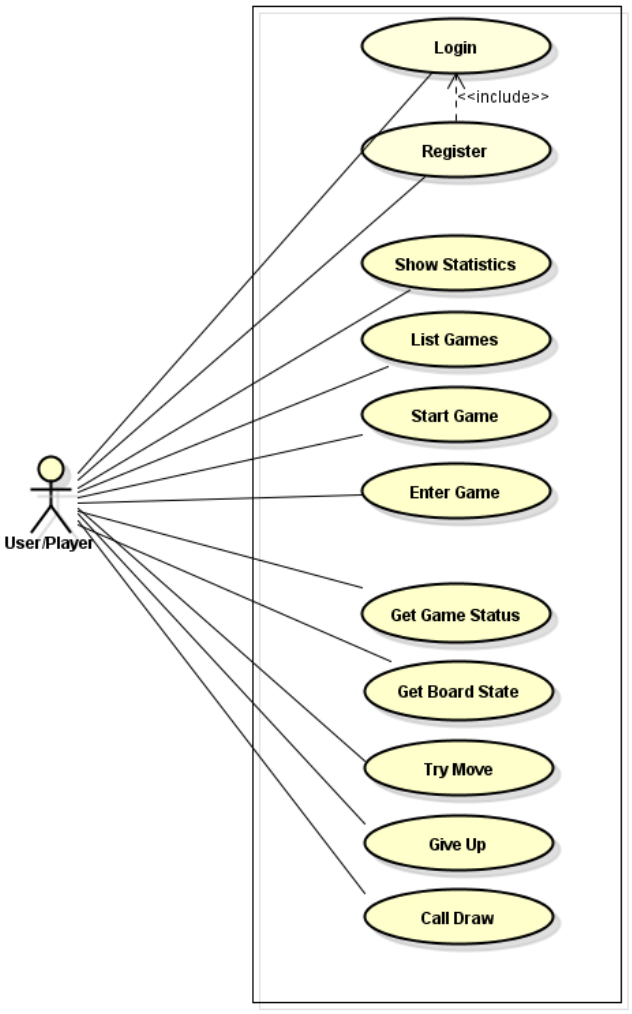
Das Web-Interface ermöglicht die Kommunikation von Benutzer:innen mit dem System. Es implementiert dabei keine eigene Funktionalität, sondern reicht alle Anfragen von Benutzer:innen direkt an das System weiter. Es kann davon ausgegangen werden, dass alle Daten, z.B. Spielstand, Züge und ggf. Textmeldungen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

Zu beachten ist, dass diese Modellierung nur eine abstrakte Sicht auf Spiel und Spieler:innen 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 Benutzer:innen und Spieler:innen als externe Aktoren genannt. Daher wäre es auch ok die Use-Cases auf zwei Akteur:innen aufzuteilen. Da dies aber die gleiche Person ist haben wir uns für nur eine:n Akteur:in 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.

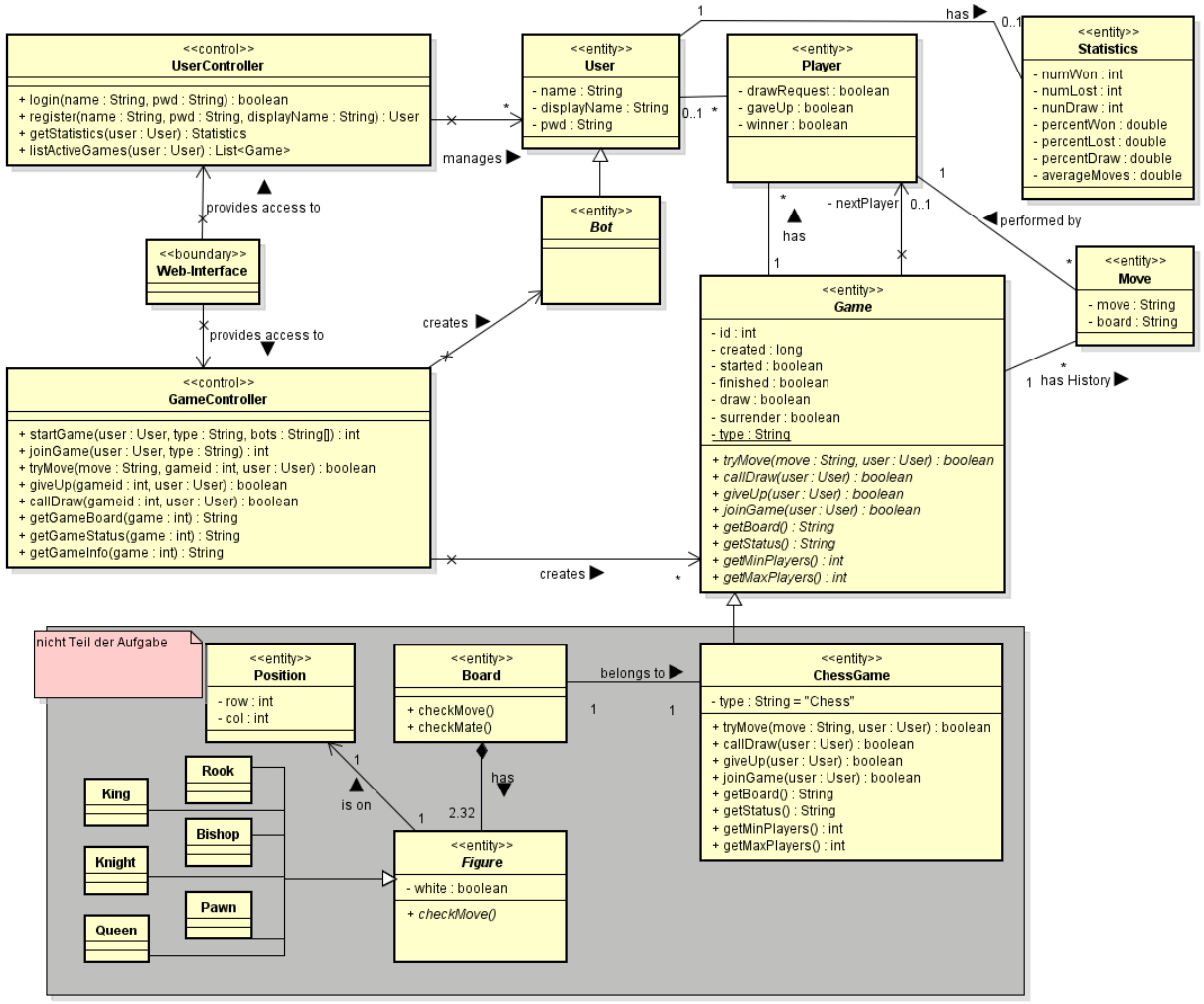
– **Lösung Ende** –

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 eine:n Akteur:in (Benutzer:in/Spieler:in) 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 Zugiffe auf ein bestimmtes Spiel beinhalten.

- Um bei einem Spiel mitzuspielen, könnte `Bot` auch von `Player` erben. Die hier gewählte Modellierung von Bots als *Benutzer:in* ermöglicht es aber, den Bot in allen Situationen wie gleichwertige menschliche Spieler:innen zu behandeln. Z.B. erhält der Bot so einen Namen, der im Web-Interface als Gegenspieler:in 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.

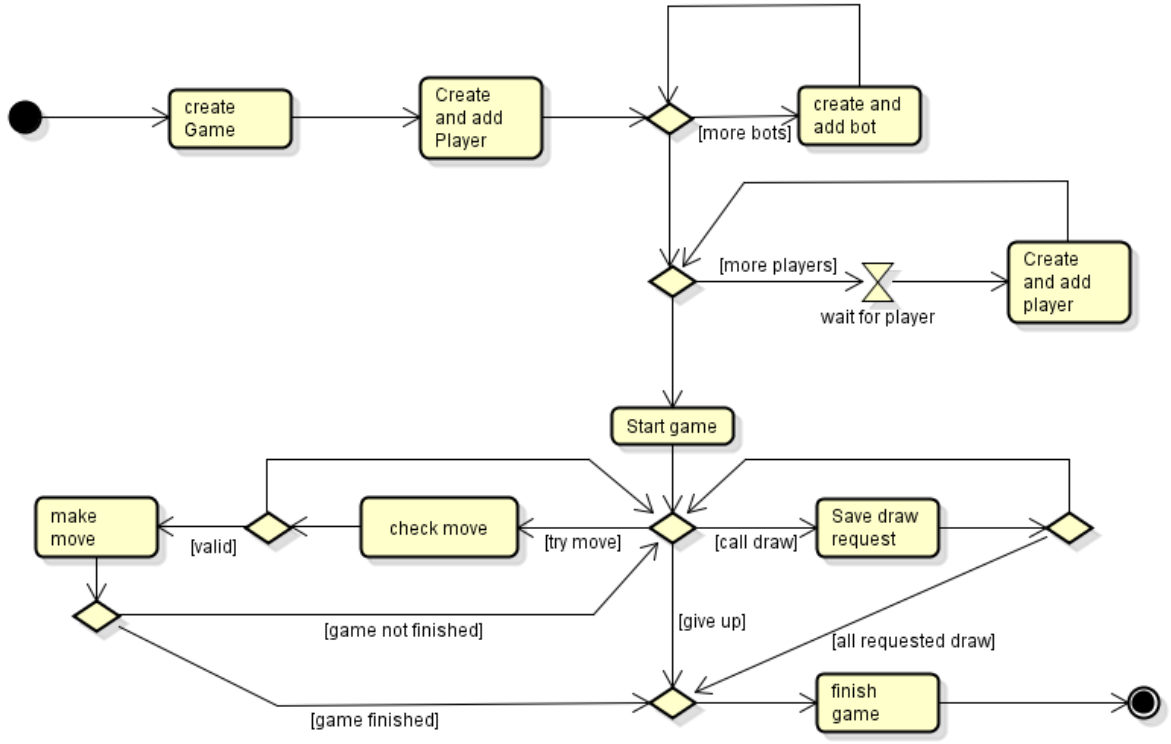
- Die Statistik könnte theoretisch auch bei jeder Anfrage neu berechnet werden und muss nicht statisch "existieren" wie hier im Klassenmodell. An dieser Stelle dient es dazu, die im Text beschriebenen Anforderungen im Modell zu repräsentieren.

– **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 Spieler:innen 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 Benutzer:innen angemeldet sind.

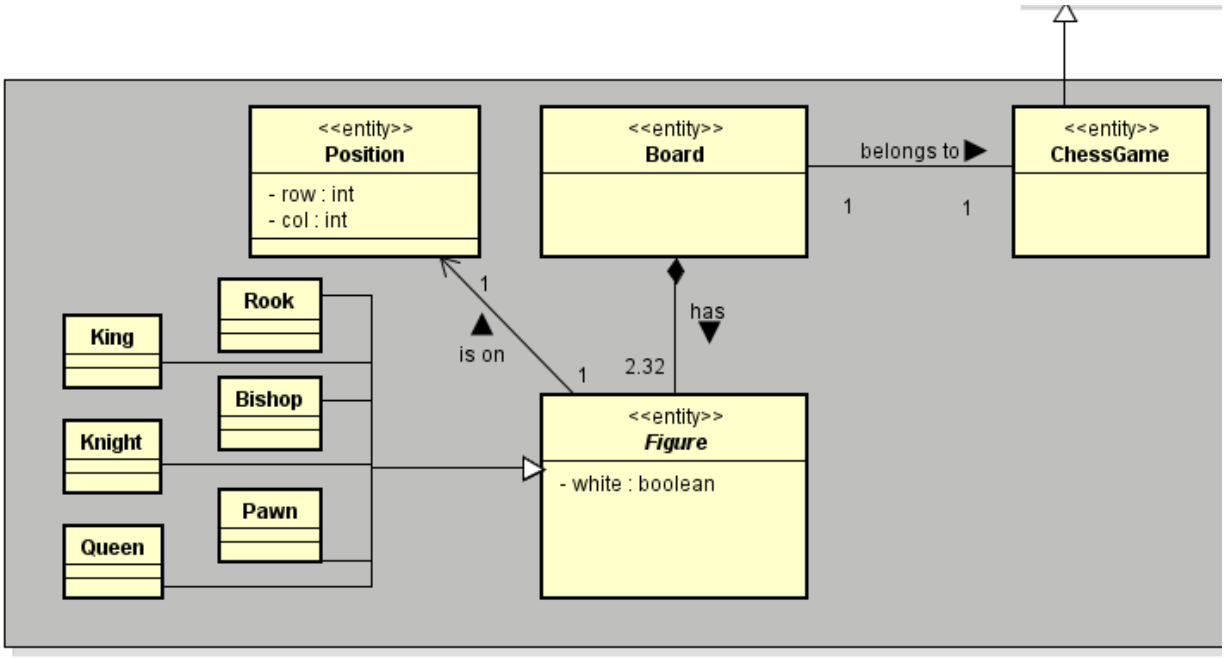
– **Lösung Anfang** –



– **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:

a) Die Benutzer:innen haben eindeutige Kennungen.

b) Zu jedem Spiel gehört mindestens ein:e Spieler:in. Zu jedem gestarteten Schachspiel gehören zwei Spieler:innen.

c) Kein:e Benutzer:in darf mehrere Spieler:innen in einem Spiel kontrollieren.

d) Benutzer:innen dürfen nicht gleichzeitig in mehr als acht Spielen, die nicht beendet sind, teilnehmen.

e) Die Anzahl der gewonnenen Spiele in der Statistik von Benutzer:innen entspricht den beendeten Spielen, für die dieses Benutzer:innen tatsächlich als Gewinner:in vermerkt ist.

f) Bevor ein Schachspiel startet, stehen auf dem Spielfeld 16 Figuren jeder Farbe.

g) Auf einer Position dürfen niemals zwei Figuren stehen.

h) Wenn alle Spieler:innen ein Unentschieden gefordert haben, ist das Spiel als Unentschieden beendet.

i) Zu langes Warten auf Zuweisung von Gegenspielern bzw. Gegenspielerinnen soll verhindert werden: Es kann kein Spiel mehr als ein:e menschliche:r Spieler:in haben solange ein früher erstelltes Spiel noch auf Spieler:innen 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 Benutzer:innen durch das Web-Interface am System angemeldet wurde und die Benutzer:in-Daten als Objekt zur Verfügung stehen.

a) Benutzer:in registrieren

b) Aufgeben

c) Unentschieden vorschlagen

d) Spiel starten

e) Eine Funktion `calculateAverageMoves`, in der die durchschnittliche Anzahl der Züge aller beendeten Spiele eines/einer Benutzer:in 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.oclIsNew() and
            u.oclIsTypeOf(User) and result=u)

context GameController::
  startGame(user:User, bots:Bag(String)):Integer
    pre: true
    post: game->exists(g| g.oclIsNew() and
            g.player.user->includes(user) and
            g.player.user->
              select(oclIsKindOf(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
```

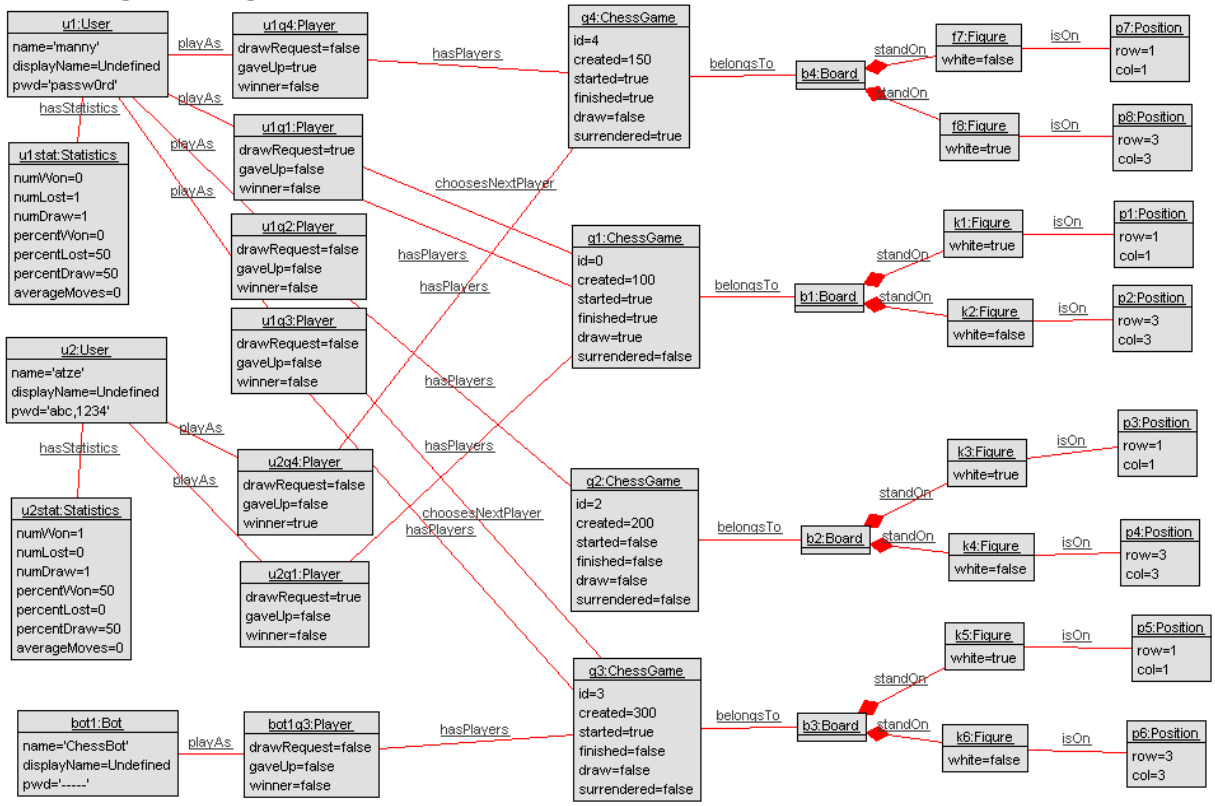
```
            player->forAll(user<>u implies (not gaveUp and winner)))
           and result=true

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 Spieler:innen 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
```
$Q\{(moves.length = 0 \land erg = 0) \lor (moves.length > 0 \land erg = \frac{\sum_{i=0}^{moves.length-1} moves[i]}{moves.length})\}$

– **Lösung Anfang** –

```
calculateAverageMoves(int[] moves):
```
$\{moves.length >= 0\}$
```
    sum := 0;
```
$\{moves.length >= 0 \land 0 = 0 \land 0 = 0 \land 0 = 0\}$
```
    cnt := 0;
```
$\{moves.length >= 0 \land sum = 0 \land 0 = 0 \land 0 = 0\}$
```
    erg := 0;
```
$\{moves.length >= 0 \land sum = 0 \land cnt = 0 \land 0 = 0\}$

$\{moves.length >= 0 \land sum = 0 \land cnt = 0 \land erg = 0\}$

$\{moves.length >= 0 \land sum = \sum_{i=0}^{0-1} moves[i] \land cnt = 0 \land erg = 0\}$

$\{moves.length >= 0 \land sum = \sum_{i=0}^{cnt-1} moves[i] \land cnt = 0 \land erg = 0\}$

$\{moves.length >= 0 \land sum = \sum_{i=0}^{cnt-1} moves[i] \land cnt <= moves.length \land erg = 0\}$
```
    while cnt < moves.length do
```
$\{cnt < moves.length \land moves.length >= 0 \land sum = \sum_{i=0}^{cnt-1} moves[i] \land cnt <= moves.length \land erg = 0\}$

$\{cnt < moves.length \land moves.length >= 0 \land sum = \sum_{i=0}^{cnt-1} moves[i] \land erg = 0\}$

$\{cnt < moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt} moves[i] - moves[cnt] \wedge erg = 0\}$

$\{cnt < moves.length \wedge moves.length >= 0 \wedge sum + moves[cnt] = \sum_{i=0}^{cnt} moves[i] \wedge erg = 0\}$

```
sum := sum + moves[cnt];
```

$\{cnt < moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt} moves[i] \wedge erg = 0\}$

$\{cnt + 1 <= moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt+1} moves[i] - moves[cnt+1] \wedge erg = 0\}$

```
cnt := cnt + 1
```

$\{cnt <= moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt} moves[i] - moves[cnt] \wedge erg = 0\}$

$\{cnt <= moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt-1} moves[i] \wedge erg = 0\}$

```
od;
```

$\{cnt >= moves.length \wedge cnt <= moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt-1} moves[i] \wedge erg = 0\}$

$\{cnt = moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{cnt-1} moves[i] \wedge erg = 0\}$

$\{cnt = moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{moves.length-1} moves[i] \wedge erg = 0\}$

$\{cnt = moves.length \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{moves.length-1} moves[i]\}$

```
if moves.length >0 then
```

$\{moves.length > 0 \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{moves.length-1} moves[i] \wedge cnt = moves.length \wedge erg = 0\}$

$\{moves.length > 0 \wedge sum = \sum_{i=0}^{moves.length-1} moves[i] \wedge cnt = moves.length \wedge erg = 0\}$

$\{moves.length > 0 \wedge sum/cnt = \frac{\sum_{i=0}^{moves.length-1} moves[i]}{moves.length} \wedge erg = 0\}$

```
erg := sum / cnt
```

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

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

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

```
else
```

$\{moves.length <= 0 \wedge moves.length >= 0 \wedge sum = \sum_{i=0}^{moves.length-1} moves[i] \wedge cnt = moves.length \wedge erg = 0\}$

$\{moves.length = 0 \wedge sum = \sum_{i=0}^{moves.length-1} moves[i] \wedge cnt = moves.length \wedge erg = 0\}$

$\{moves.length = 0 \wedge erg = 0\}$

```
skip
```

$\{moves.length = 0 \wedge erg = 0\}$

$\{(moves.length = 0 \wedge erg = 0) \vee \bot\}$

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

```
fi
```

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

Terminierungsfunktion: moves.length - cnt

a) $\{cnt <= moves.length\} \Rightarrow moves.length - cnt \geq 0$

b)

```
while cnt < moves.length do
```

$\{moves.length - cnt = m\}$

$\Rightarrow \{moves.length - cnt < m + 1\}$

$\Rightarrow \{moves.length - cnt - 1 < m\}$

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

– **Lösung Ende** –