Die Hausaufgaben müssen von jedem Studierenden einzeln bearbeitet und abgegeben werden. Für die Hausaufgabe sind die aktuellen Informationen vom Blog `https://seblog.cs.uni-kassel.de/ws1920/programming-methodologies/` zu berücksichtigen.

**Abgabefrist ist der 13.02.2020 - 23:59 Uhr**

## Abgabe

Wir benutzen für die Abgabe der Hausaufgaben Git. Jedes Repository ist nur für den Studierenden selbst, sowie für die Betreuer und Korrekteure sichtbar. Für diese und voraussichtlich alle zukünftigen Abgaben wird kein neues Repository benötigt. Nutzt weiterhin jenes, welches mit folgendem Link angelegt wurde:

`https://classroom.github.com/a/X9QUkSjx`

**Nicht, oder zu spät gepushte (Teil-)Abgaben werden mit 0 Punkten bewertet.**

## Vorbereitung

Zur Bearbeitung der Hausaufgabe sollte eine Entwicklungsumgebung verwendet werden. Wir empfehlen aufgrund der Nachvollziehbarkeit die Verwendung von IntelliJ (siehe Aufgabenblatt 3). Die Abgabe muss als **lauffähiges** Projekt abgegeben werden.

## Aufgabe 0 - Evaluation

Zum Ende der Veranstaltung würden wir gerne eine Evaluation durchführen. Diese dient in diesem Fall nicht nur dafür, Feedback für uns einzuholen, sondern dient in diesem Semester ebenfalls als Datengrundlage für die Abschlussarbeit eines eurer Kommilitonen.

Diese Arbeit nutzt die gesammelten Daten dabei vollkommen anonym und nutzt lediglich die Aussagen über die Nutzung von Fulib.org.

Der folgende Link führt zur Umfrage. Wir bedanken uns im Voraus bei allen, die sich die Zeit nehmen, um uns bei der Gestaltung der Lehre zu helfen.

`https://forms.gle/BsKmeiA7Tdh7YxGUA`

## Aufgabe 1 - Attack (20P)

In dieser Aufgabe soll der Kampf, sowie die Übernahme von Houses implementiert werden. Die Regeln, nach denen der Kampf an einem einzelnen House abläuft, lauten wie folgt:

- Summiert die attackValues jeder Partei auf. (Partei = alle Shrooms eines Players am House)
- Die Summe der Partei, der auch das House gehört, wird als Verteidigungswert aufgefasst.
- Bildet die Differenz zwischen dem Verteidigungswert und allen Angriffswerten.
  - Ist die Verteidigung höher als der Angriff, wird die Differenz auf alle Angreifer verteilt
  - Ist der Angriff höher, trifft die Differenz den Verteidiger mit voller Wucht.
- Ist die verteidigende Partei nicht mehr vertreten, das House jedoch noch nicht eingenommen, wird der Schaden einer Partei auf die anderen anwesenden Parteien aufgeteilt.
- Ist nur noch eine feindliche Partei vorhanden, so wird der Schaden von den HP des Houses abgezogen.
- Fallen die HP eines Houses auf 0, so wird dies vom Angreifer eingenommen und dem Spieler zugeordnet.

Anschließend muss die battle()-Methode in den Gameloop eingebunden werden und das Besiegen der Shrooms sollte über Property Change Listener realisiert werden.

```
package de.uniks.pmws1920.builder;

public class  ModelBuilder {

    // ==========================================================================
    // Game Rule Methods
    // ==========================================================================

    public void battle (){
        // put battle  code here
    }
}
```

Abbildung 1: ModelBuilder

## Aufgabe 2 - Wincondition (40P)

In dieser Aufgabe soll die Logik des Gameloops abgeschlossen werden. Am Ende einer Partie sollte ein Gewinner festzustellen sein und auch dementsprechend gewürdigt werden. Erweitert die Anwendung um die Verarbeitung eines Gewinners, sowie die Anzeige dessen.

Zu Beginn sollte eine checkWin()-Methode zum Modelbuilder wie in Codesnippet 2 hinzugefügt werden. Definiert eine eigene Wincondition als Kommentar oberhalb der Methode. Diese Wincondition sollte logisch gewählt sein, das bedeutet, dass ein Spieler die klare Oberhand gewonnen haben muss. Ob eure Wincondition nun ist, dass alle Houses von einem Player eingenommen wurden und sämtliche Shrooms anderer Spieler besiegt worden sind, oder eine weichere Condition, ist euch überlassen, solange sie klar in Textform definiert wurde.

```
package de.uniks.pmws1920.builder;

public class  ModelBuilder {

    // ================================================================================
    // Game Rule Methods
    // ================================================================================

   // <Define wincondition here>
   public Player checkWin(){
        // check your wincondition
        // return the  player  that  won the game (or null if  no player has won yet)
   }
}
```

Abbildung 2: ModelBuilder Wincondition

Nach Erfüllung der Wincondition soll dem Nutzer angezeigt werden, welcher Player gewonnen hat. Neben der Anzeige des Names und der Farbe des Spielers soll ein Button angezeigt werden. Beim Klick auf den Button gelangt der Spieler zurück in den SetupView und das Model muss resettet werden. Somit kann dann ein weiteres Spiel begonnen werden.

Eine Beispielumsetzung der UI kann der Abbildung 3 entnommen werden. Es darf, wie bereits zuvor, von der Darstellungsform abgewichen werden.

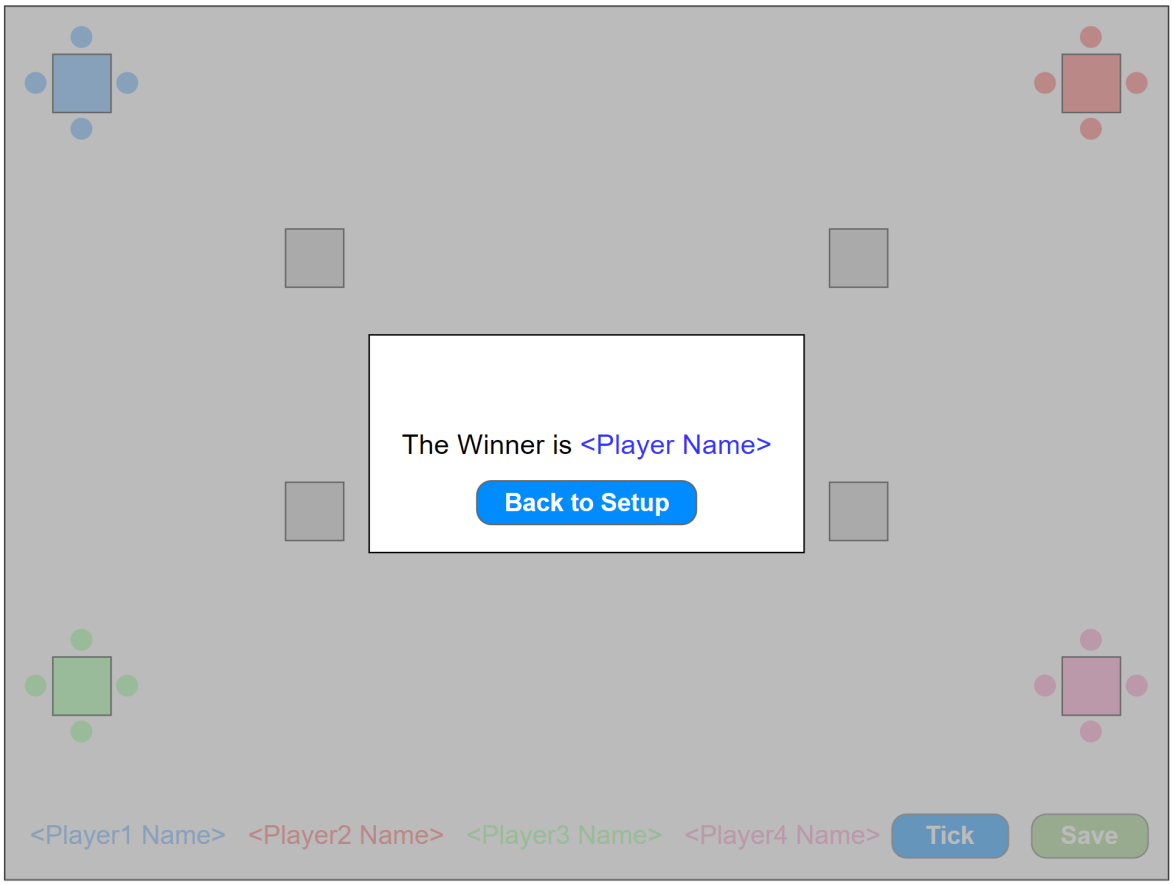


Abbildung 3: Anzeige des Gewinners

## WinTest

Eine Funktionalität, wie das Gewinnen eines Spieles, durch das Durchführen eines Testspieles zu prüfen, erscheint weder angenehm noch vom Arbeitsaufwand rational. Dies ist ein klassischer Fall, um die Funktionalität mithilfe eines JUnit-Tests zu prüfen.

Die Aufgabe besteht darin, einen Test zu formulieren, der sowohl die Funktionalität der checkWin()-Methoden als auch die Anzeige des Gewinners in der UI in zwei separaten Test-Methoden überprüft. Die in Codesnippet 4 vorgegebene Struktur darf hierfür übernommen werden.

```
package de.uniks.pmws1920.gui;

public class  WinTest extends ApplicationTest {

    @Override
    public void start(Stage stage)  {
        ShroomWars wars = new ShroomWars();
        wars.start(stage);
    }

    @Test
    public void showWinningPlayerTest() {
        // manage to display the ingame screen
        // manipulate the the model to make one player win the game
        // test  the  displayed Ui (minimum two Asserts)
    }

    // This can be a  Simple Model Test without the usage of any UI
    // If  you want to move this method into another tests that's  fine
    @Test
    public void winConditionTest() {
        //  initialize   a  normal game
        // make a objectdiagram-dump that displays the Model after the games init
        // manipulate the the model to make one player win the game
        // test  the  Model for the defined win-condition (minimum two Asserts)
        // make a objectdiagram-dump that displays the Model after the win-condition holds
    }
}
```

Abbildung 4: WinTest

## Aufgabe 3 - Singleplayer (40P)

Das ist die letzte reguläre Aufgabe der Hausaufgaben in diesem Semester. Lasst uns also die bisherige Form der Aufgabenstellung aufbrechen.

Kernpunkt dieser Aufgabe ist es, allein anhand der Beschreibung die gewünschte Funktionalität umzusetzen. Es gibt keine Codesnippets oder andere Hilfestellungen, an welche Stelle die Methodenaufrufe gehören oder wie viele Methoden es geben sollte. Der bisher geschriebene Code und die Struktur des Programms sollte also verinnerlicht worden sein, um diese Aufgabe lösen zu können.

Bisher konnten wir jeden Player befehligen, was nicht wirklich viel Sinn in einem Spiel macht. Ändert die Logik des Programms so ab, dass es zu einem Singleplayerspiel wird. Es muss also im SetupView erkennbar sein, welcher Player man selbst ist, oder es muss möglich sein, festzulegen, welchen der Player man spielt. Im IngameView darf man ausschließlich eigene Shrooms befehligen dürfen. Es sollte also egal sein, in welcher Reihenfolge man die Houses anklickt, es sind nur Bewegungen möglich, die von einem eigenen House ausgehen.

---

Wer am Ende tatsächlich ein waschechtes Spiel haben möchte, kann sich noch damit auseinandersetzen, wie man ein Gegnerverhalten einbauen könnte. Hierzu könnte man beispielsweise eine neue Klasse erstellen, die sich Bot nennt und random, oder nach einer Strategie, Spielzüge ausführt.
**Das Implementieren eines solchen Gegnerverhaltens ist optional!**