Spiellogik automatisiert testen

Einleitung:

Testen, bevor es implementier ist?

Ja das geht und hat sogar einen eigenen Namen: "Test Driven Development".

Auf der Basis der Signaturen der Methoden der Spiellogik implementieren Sie zuerst die Tests. Diese werden zunächst alle fehlschlagen.

Dann implementieren Sie die Spiellogik… bis alle Tests erfolgreich sind.

Und ja, wahrscheinlich werden Ihnen währen der Implementierung weitere Tests einfallen, die Sie ergänzen.

Ziel

- ⇨ Ich kann eine Aufgabe verfeinern und Anforderungen formulieren.
- ⇨ Ich kann Anforderung nach den Kriterien formulieren, überprüfen und verbessern. (vollständig, verständlich, atomar, identifizierbar, nachprüfbar, konsistent)
- ⇨ Ich kann einen Ablauf entwerfen, zeichnen und umsetzen.
- ⇨ Ich kann ein Testkonzept mit Testfällen entwerfen.
- ⇨ Ich kann einen Test durchführen.

Inhalte

Aufgabe 1: Klassen und Methoden mit Signaturen übernehmen

Damit wir eine gemeinsame Basis für das Test Drive Development haben, gebe ich Ihnen die Signatur der Methode der Klasse Spiellogik vor.

Zudem brauche ich eine Klasse Crime, die für das "Secret" und die "Suggestion" verwendet wird.

Vorbereitung:

Übernehmen Sie die Klassen in Ihr Projekt.

- model/Crime.java
- logic/GameLogic.java

Verstehen Sie, was die Klassen machen?

Sie werden nun zuerst die Tests schreiben und erst danach ergänzen Sie die Implementierung von der GameLogic. (Test Driven Development)

Aufgabe 2: Ersten Test gemäss Bespiel schreiben und ausführen

Schreiben Sie einen ersten Test zur Methode:

```
public boolean evaluateSuggestion(Crime suggestion, Crime secret,
   int numberOfSuggestion, int maxNumberOfSuggestions){
```

Wie man das macht?

https://www.jetbrains.com/help/idea/create-tests.html

https://www.jetbrains.com/help/idea/performing-tests.html

https://www.jetbrains.com/help/idea/viewing-and-exploring-test-results.html

Schreiben Sie einen ersten Test gemäss diesem Bespiel und probieren Sie obiges damit aus.

```
@Test
void evaluateSuggestion() {
   Crime suggestion = new Crime();
   Crime secret = new Crime();
   int numberOfSuggestion = 0;
   int maxNumberOfSuggestions = 8;

   //setup secret
   secret.setActor(0);
   secret.setWeapon(0);
   secret.setScene(0);

   /* setup suggestion with same values as secret
    * so I expact to win :-)
    */
   suggestion.setActor(0);
   suggestion.setWeapon(0);
   suggestion.setScene(0);

   GameLogic gameLogic = new GameLogic();

   //return true is expected
   assertEquals( expected: true, gameLogic.evaluateSuggestion(suggestion, secret, numberOfSuggestion, maxNumberOfSuggestions));
}
```

Aufgabe 3: Wissensaufbau: Wie funktioniert JUnit?

Lösen Sie die Aufgabe " 18.1_EinführungZuJUnit.pdf" und lernen Sie den ersten Umgang mit JUnit.

Input: @BeforEach

Grundsatz: Jede Testmethode darf nur etwas testen.
Damit gibt es am Ende sehr viele einzelne Tests.

Doch einige Teile braucht man für jeden Test. Das kann man zusammenfassen.

Beispiel zur Spiellogik: Setup, das jeder Test **erneut** und **neu** braucht.

```
class GameLogicTest {
    2 usages
    private GameLogic gameLogic;
    5 usages
    private Crime suggestion;
    5 usages
    private Crime secret;
    2 usages
    private int numberOfSuggestion;
    2 usages
    private int maxNumberOfSuggestions;

    @BeforeEach
    private void setupBevorEachTest(){
        gameLogic = new GameLogic();
        suggestion = new Crime();
        secret = new Crime();
    }
```

Macht den Test schlanker

```
    @Test
    void evaluateSuggestion() {
        numberOfSuggestion = 0;
        maxNumberOfSuggestions = 8;

        //setup secret
        secret.setActor(0);
        secret.setWeapon(0);
        secret.setScene(0);

        /* setup suggestion with same values as secret
         * so I expact to win :-)
         */
        suggestion.setActor(0);
        suggestion.setWeapon(0);
        suggestion.setScene(0);

        //return true is expected
        assertEquals( expected: true, gameLogic.evaluateSugg
    }
```

Input: Sprechende Namen für die Tests

Tests brauche sprechende Namen:

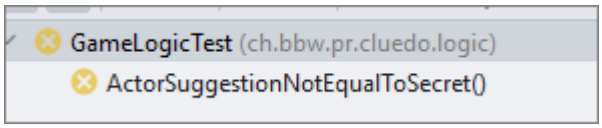
Nicht so gut:

```
23      @Test
24      void evaluateSuggestion() {
```

Besser

```
44      @Test
45      void ActorSuggestionNotEqualToSecret() {
```

Weil man dann gleich sieh, was fehlgeschlagen ist:



Aufgabe 4: Tests für Anforderungen implementieren

Nun müssen Sie weitere Tests, um die Methode *evaluateSuggestion* der Spiellogik damit zu testen.

Mit den Tests werden Sie die untenstehenden Anforderungen abdecken.

Pro Test darf nur etwas getestet werden. Es gibt also je nach Anforderung ein oder mehrere Tests.

Der Namen der Testmethoden muss angeben, was getestet wird. (Siehe dazu Input oben)

Beispiel einer solchen Testmethode:

```
23      @Test
24      void ActorWeaponSceneNotEqualThenReturnFalseAndHistory0() {
25          //there are still suggestions possible
26          numberOfSuggestion = 0;
27          maxNumberOfSuggestions = 8;
28
29          //setup secret
30          secret.setActor(0);
31          secret.setWeapon(0);
32          secret.setScene(0);
33
34          //setup suggestion
35          suggestion.setActor(1); //not equal secret
36          suggestion.setWeapon(1); //not equal secret
37          suggestion.setScene(1); //not equal secret
38
39          //as secret is not uncovered, return false is expected
40          assertFalse(gameLogic.evaluateSuggestion(suggestion, secret, numberOfSuggestion, maxNumberOfSuggestions));
41          //message in history's last entry is expected
42          assertEquals( expected: "Correct: 0 Try again", suggestion.getHistory().get(suggestion.getHistory().size()-1));
43      }
```

Auf der nächsten Seite finden Sie die Anforderungen und die vorgeschlagenen Namen der Testmethoden.

Anforderungen und Testmethoden:

22. Ist keines der geratenen Teile (Täter, Tatwaffe, Tatort) zutreffend, so wird in der Historie 0 vermerkt.

```
ActorWeaponSceneNotEqualThenReturnFalseAndHistory0(..
```

23. Ist eines der geratenen Teile (Täter, Tatwaffe, Tatort) zutreffend, so wird in der Historie 1 vermerkt.

```
ActorEqualWeaponSceneNotEqualThenReturnFalseAndHistory1(..
WeaponEqualActorSceneNotEqualThenReturnFalseAndHistory1(..
SceneEqualWeaponSceneNotEqualThenReturnFalseAndHistory1(..
```

24. Sind zwei der geratenen Teile (Täter, Tatwaffe, Tatort) zutreffend, so wird in der Historie 2 vermerkt.

```
ActorWeaponEqualSceneNotEqualThenReturnFalseAndHistory2(..
ActorSceneEqualWeaponNotEqualThenReturnFalseAndHistory2(..
WeaponSceneEqualActorNotEqualThenReturnFalseAndHistory2(..
```

25. Sind drei der geratenen Teile (Täter, Tatwaffe, Tatort) zutreffend, so wird in der Historie "gewonnen" vermerkt.

```
ActorWeaponSceneEqualThenReturnTrueAndHistoryWin(..
```

26. Ist das Rätsel nach 8 Versuchen nicht korrekt erraten, so wird in der Historie "verloren" vermerkt und es sind keine weiteren Rateversuche möglich.

```
MaxNumberOfSuggestionReachedAndNotWinThenReturnFalseAndHistoryNoneLeft(..
```

21. Die Details der Rateversuche und das Ergebnis jedes Rateversuches werden in der Historie gespeichert.

Selbst ergänzen, sofern nicht bereits in den obigen Tests enthalten.

Aufgabe: Spiellogik implementieren

⇨ Die Integration der Spiellogik in das GUI ist (noch) kein Thema.
Dank den Tests, können Sie die Spiellogik dennoch testen.

Sie folgen dem Prinzip von *Test Driven Develpment*
→ zuerst die Tests dann die Implementierung.

- Entwerfen Sie nun den Ablauf für die Spiellogik.
- Implementieren Sie die Spiellogik, schrittweise.
- Testen Sie Ihre Implementierung laufend, es sollten immer mehr Tests erfolgreich verlaufen.
- Erkennen Sie, dass ein Test fehlt? Dann ergänzen Sie den Test.