

Test Driven Development Trainer

Systembeschreibung

1 GUI UND CONTROLLER (ALESSANDRA)

1.1 LAYOUT

Das wesentliche Layout wird in der FXML-Datei `layout.fxml` spezifiziert. Die Basis bildet ein `BorderPane`. Die einzelnen Bereiche werden wie folgt belegt:

- **TOP** In `<top>` befindet sich eine Menu- und Anzeigenleiste. Diese besteht aus folgenden Buttons, die mit entsprechenden Actions versehen sind:
 - **COMPILE**: Dient dazu, den Code testweise zu kompilieren (ohne den Modus zu wechseln)
 - **NEXT**: Code wird kompiliert und Nutzer geht in den nächsten Modus über
 - **BACK TO RED**: Befindet sich der User gerade im Modus GREEN, kommt er mittels diesem Button wieder zurück zu RED
 - **END REFACTOR**: Refactoring beenden
 - **TRACKING**: Ruft Tracking auf.

Außerdem beinhaltet die Leiste noch Anzeigen für den derzeitigen Modus, sowie die Zeit bei BabySteps.

- **Left**: In `<left>` befindet sich das Auswahlmenu der Aufgaben. Dieses wurde als `TableView` realisiert, welche Instanzen der Klasse `MenuEntry` als Einträge verwaltet. Die Darstellung erfolgt in 2 Spalten. Zum einen wird der Name der Aufgabe angezeigt. Zum anderen wird angezeigt, ob die Funktion `BabySteps` bei der Aufgabe eingeschaltet ist oder nicht. (Wir hatten uns hierbei dazu entschlossen, die Funktion `BabySteps` aufgabenspezifisch einzubauen. D.h. in den Konfigurationen ist festgelegt, bei welchen Aufgaben `BabySteps` eingeschaltet ist.). Nettes Extra bei `TableView`: Bei einem Klick auf den jeweiligen Spaltenkopf werden automatisch Sortierungen vorgenommen.
- **Center**: Hier befindet sich ein `TextArea`, das für den jeweiligen Code einer Aufgabe verwendet wird (als Anzeige und zum Editieren). Zu Beginn kann man hier noch nicht hineinschreiben (wird erst nach Auswahl einer Aufgabe und wenn die entsprechende Phase erreicht wird freigegeben)
- **Right**: Analog für den Test Code ein `TextArea`.
- **Bottom**: `TextArea`, das Nachrichten an User übergibt (Kompilierfehler, etc.)

1.2 MENUENTRY

Die Instanzen dieser Klasse dienen dazu, die Listeneinträge des Aufgabenauswahlmenus zu realisieren. Neben den Attributen für die Anzeige von Titel und BabySteps wird hier auch die jeweilige `Exercise` gespeichert, damit bei einem Klick auf den jeweiligen Menu-Eintrag entsprechen im Controller reagiert werden kann. Die zunächst redundant erscheinenden Getter und Setter, sowie der Default-Konstruktor werden für das Einbinden in die `TableView` benötigt (ohne kompiliert der Spaß nicht). Generell habe ich mich bei dieser Konstruktion an dem Beispiel auf http://docs.oracle.com/javafx/2/ui_controls/table-view.htm orientiert.

1.3 CONTROLLER

Die Klasse `Controller` implementiert das Interface `Initializable`. Dadurch wird beim Aufruf der entsprechenden Datei direkt die Methode `initialize` aufgerufen: Hier wird die anfängliche Phase auf RED

gesetzt (egal, welche Aufgabe der Nutzer auswählt: gestartet wird im Modus RED). Dann wird der XML-Katalog mittels `ConfigParser` geparsed. Erhalten dadurch eine Liste mit Exercises, die dann über eine `foreach`-Schleife direkt in das Aufgabenmenu eingehängt werden.

Danach wird noch der Event-Handler für das Aufgabenmenu gesetzt: Hierbei musste etwas getrickst werden, weil es ansonsten Probleme bzgl. der Klickauswahl gab: Es beginnt erstmal recht klassisch. Der angeklickte Eintrag wird über das `SelectionModel` von `TableView` zurückgegeben. Nun kann es aber sein, dass nur ein paar Aufgaben in der Liste sind und der User statt einer Aufgabe einen leeren Eintrag anklickt. In diesem Fall kriegen wir null zurück, was natürlich abgefangen werden muss. Haben wir tatsächlich einen `MenuEntry` als Rückgabe, so können wir hier die entsprechende Exercise erhalten, aus der wir die Konfigurationsdaten (Templates, BabySteps) lesen. Zunächst hatten wir geplant, mehr als eine Klasse pro Aufgabe zu erlauben. Dies wurde aber letztlich umgeändert (in der Aufgabenstellung ist dies nicht explizit festgelegt), sodass es pro Aufgabe genau eine Klasse für Code und eine Klasse für die Tests gibt (für Info1 dürfte das größtenteils durchaus reichen ;)). Wir haben die Listenstrukturen allerdings noch beibehalten, damit der Parser dafür nicht extra umgestellt werden muss. Die entsprechenden Templates werden in die dafür vorgesehenen TextAreas geladen und Funktionalitäten von BabySteps werden aufgerufen (dazu: siehe Bericht zu BabySteps). Nachdem das Auswerten des Klicks vollzogen wurde, wird die Markierung aufgehoben. Dies das eben erwähnte Tricksen, denn wenn die Markierung auf dem Feld beibehalten wird, erhält man Probleme, wenn man beispielsweise auf die Spaltenköpfe zum Sortieren klickt (entspricht dann einem erneuten Klick auf den markierten Eintrag). Außerdem werden die jeweiligen Buttons und das Textfeld für die Tests freigegeben und entsprechend gefärbt.

1.4 WEITERE

Haben noch weitere Methoden im Controller:

`compileCode`: Wird bei einem Klick auf den `COMPILE` Button sowie beim Versuch eines Phasenwechsels aufgerufen. Versucht mittels `CompilerManager` den Code zu kompilieren. Gab es beim Kompilieren Fehler, so werden diese im unteren Textfeld angezeigt.

`next`: Aufruf bei einem Klick auf den `NEXT`-Button. Der User beantragt damit einen Phasenwechsel. Dafür wird zunächst einmal `compileCode` aufgerufen. Danach wird je nach aktueller Phase (`RED` oder `GREEN`; bei Refactor ist `NEXT` nicht anklickbar) anders reagiert: `GREEN`: Wenn alle Tests erfolgreich waren, hat der User 2 Möglichkeiten: Entweder er geht in den Refactor-Modus oder er beginnt wieder bei `RED`. Der User entscheidet sich über ein Alert-Fenster. Entscheidet er sich für Refactor, so wird zusätzlich die TextArea für die Tests freigegeben und die Phase optisch eingeleitet. Zusätzlich wird der Button für das Beenden vom Refactoring freigegeben. Ansonsten wird über `initRedMode` der Modus `RED` gestartet. Waren die Tests nicht erfolgreich, so wird je nach Fehlerfall (Code nicht kompilierbar oder Tests noch nicht erfolgreich) eine Nachricht in der unteren TextArea ausgegeben. `RED`: Der User darf nur in den Modus `GREEN` wechseln, wenn der Code nicht kompiliert oder wenn genau ein Test fehlschlägt. Der Wechsel wird mit `initGreenMode` vollzogen. Ansonsten wird eine entsprechende Meldung an den User weitergegeben.

`endRefactor`: Überprüft, ob der Code noch kompiliert und ob alle Tests laufen. (Ansonsten darf er Refactor noch nicht verlassen.) Ist dies der Fall, so wird `initRedMode` aufgerufen und die Anklickbarkeit der Buttons modifiziert (d.h. ein Phasenwechsel zu `RED` wird vollzogen).

`initRedMode`, `initGreenMode` und `backToRed` initialisieren einen entsprechenden Phasenwechsel. Dabei werden die Ränder und die Editierbarkeit der TextAreas entsprechend angepasst. Außerdem werden Dinge für BabySteps getan (siehe Bericht BabySteps). In `initGreenMode` wird zudem noch der Text, der sich bei Phasenwechsel in der CodeArea befindet, in einer File zwischengespeichert. Bei `backToRed` wird der zuvor zwischengespeicherte Text wieder in das Textfeld geladen.

`startTracking`: Handshake mit Tracking (Weitergabe der Daten, Öffnen der Grafiken)

Sonstiges: Bei sämtlichen Situationen, in denen die Phase gewechselt wird, wird die Zeit gestoppt und eine Liste, die an Tracking weitergegeben wird, gebaut

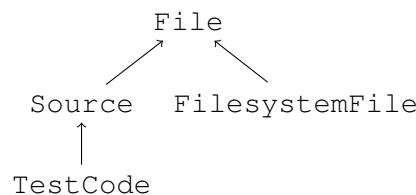
2 PARSER UND COMPILER (PASCAL)

2.1 KLASSEN UND IHRE ZUSTÄNDIGKEIT

2.1.1 Parsen der Konfigurationsdatei

Dateien, Code und Tests

Die Ableitung der Klassen entspricht dem unten dargestellten Schema.



Ferner existiert noch ein Interface `ReallyExistingFile`, welches hauptsächlich für Unit-Tests eingeführt wurde. Seine einzige Methode besteht in der Rückgabe einer Instanz von `org.xml.sax.InputSource`.

Die Klasse `FilesystemFile` kann auf Dateien zugreifen, die wirklich auf dem Dateisystem vorhanden sind. Eine Klasse, die einzig für Unit-Tests eingeführt wurde ist `DummyFile`. Diese beiden Klassen implementieren jeweils das Interface `ReallyExistingFile`.

Die Klasse `Source` repräsentiert dabei eine bereits kompilierte Quellcode-Java-Datei. Sie enthält eine Liste der Fehler, die aufgetreten sind, den Klassennamen sowie deren Inhalt.

Die Klasse `TestCode` ist dabei für die Test-Dateien vorgesehen. Sie enthält zusätzlich zu den Daten von `Source` noch die Anzahl der Tests, sowie die Tests, die fehlgeschlagen sind.

Aufgaben

Jede Aufgabe wird durch eine Instanz der Klasse `Exercise` repräsentiert. Diese enthält Informationen wie den Namen, eine Beschreibung (nicht verpflichtend) und die vorgefertigten Klassen- und Test-Templates. Alle aus einem Aufgabenkatalog gewählten Aufgaben sind in einem Objekt der Klasse `Catalog` gespeichert.

Der Parser

Die Konfiguration des TDDT wird im XML-Format verfasst. Die Hauptkomponente für diesen Bereich ist die Klasse `ConfigParser`. Diese interpretiert die Konfigurationsdatei unter Verwendung der mit Java mitgelieferten XML-Bibliothek. Aufgabe dieser Klasse ist das Lesen der Konfigurationen und Aufbauen einer Liste mit `Exercise`-Objekten.

2.1.2 Der Compiler Manager

Diese Klasse kümmert sich um die Steuerung der mitgelieferten Compiler-Bibliothek. Dem Konstruktor werden dabei zwei Objekte vom Typ `File`, eines für die Klasse, das andere für die Tests übergeben.

Die Methode `compile` interagiert anschließend mit der Compiler-Bibliothek und erzeugt Objekte vom Typ `Source` und `TestCode`, welche Compiler-Fehler und evtl. fehlgeschlagene Tests enthalten.

2.2 INTERAKTION ZWISCHEN DEN KLASSEN

2.2.1 Der Parser

Bei der Erzeugung eines neuen Parsers wird eine Eingabedatei erwartet. Darunter ist ein Objekt zu verstehen, welches das Interface `ReallyExistingFile` implementiert.

Gestartet wird der Parser mit der Methode `parse`. Dieser geht nun die gesamte Datei durch und baut nach und nach die Liste aller Aufgaben (`Exercise`) auf. Diese Methode wirft im Fehlerfall entweder eine Ausnahme vom Typ `ConfigParserException` oder Ausnahmen, die der Dokumentation von SAX zu entnehmen sind.

2.2.2 Der Compiler Manager

Der Compiler Manager erwartet im Konstruktor zwei Objekte vom Typ `File` (Code und Tests) sowie eine Angabe über die aktuelle Phase des Entwicklungszyklus (`enum Cycle`). Die Methode `run` startet den Übersetzungsvorgang. Sollten währenddessen Fehler auftreten, d.h. Compiler-Fehler oder aber eine falsche Anzahl Tests schlägt fehl, so gibt diese Methode den Wert `false` zurück - andernfalls `true`.

Durch Aufruf der Methoden `getSource` und `getTestCode` können die vom Manager erzeugte Objekte für die Klasse und die Tests zurückgegeben werden. Diese enthalten detaillierte Beschreibungen der Fehler.

3 BABYSTEPS (EYYÜP)

3.1 OFFENE FRAGEN

- Die Bearbeitungszeit für BabySteps wird in der Konfiguration einer Aufgabe in Sekunden angegeben. Dort wird auch angegeben, ob für eine Aufgabe die BabySteps- Funktion eingeschaltet ist.
- Der Timer hat das Format `mm:ss`.

3.2 KLASSEN UND IHRE ZUSTÄNDIGKEITEN

3.2.1 BabyStepsConfig

Diese Klasse ist zuständig für die Initialisierung und Aktualisierung eines Timer-Labels oder anders ausgedrückt für all das, was keine Komponenten des Controllers benutzt. Mit der `init()`-Methode wird ein Label an eine `StringProperty` (`sp=Klassenvariable`) gebunden und zu einer gegebenen Sekundenanzahl im Format `mm:ss` initialisiert. Mit der `count()`-Methode wird der Timer dann runtergezählt. Dabei wird ein neuer Thread erstellt, damit parallel Aktionen möglich sind. Dieser wird eine Sekunde angehalten, der Timer um eine Sekunde reduziert und die aktuelle Zeit berechnet sowie über die `StringProperty` aktualisiert.

3.2.2 StopUhr

Diese Klasse stellt Methoden zur Verfügung, mithilfe derer man eine Startzeit sowie eine Endzeit setzen kann, sodass man mithilfe einer weiteren Methode die Zeit dazwischen messen kann.

3.3 INTERAKTIONEN ZWISCHEN DEN KLASSEN

Die Option `BabySteps` wird im Controller durch die Methode `babyStepsHandling` erfasst. Sie wird jeweils in der `initialize`-Methode und der `next`-Methode aufgerufen. Ausgeführt wird die Option, wenn in der Konfiguration einer Aufgabe `BabySteps` eingeschaltet ist und man sich nicht in der Refactor-Phase befindet (d.h. wenn `cycle != REFACTOR`). Sie ruft ihrerseits die Methoden `init` zur Initialisierung des Timer-Labels sowie zur Bindung des Labels an die `StringProperty` `sp` und dann `count` zum Runterzählen der Zeit in einem Neben-Thread auf. Dabei werden auch die Inhalte der TextAreas `testArea` oder `codeArea` in der `Stringvariable` Puffer zwischengespeichert, je nach dem in welcher Phase man sich befindet (*RED* oder *GREEN*). Parallel dazu läuft die Methode `babyStepsAbbruch`, die in einem weiteren Thread prüft, ob die Zeit des Timer-Labels abgelaufen ist, also ob `0:00` erreicht wurde. Wenn dies der Fall ist, wird die für die aktuelle Phase maßgebende TextArea auf den im Puffer zwischengespeicherten Wert zurückgesetzt. Die Boolean-Klassenvariablen `successfullCompiling`, `refactoring` und `finished` der Controller-Klasse sowie die Boolean-Klassenvariable `stopThread` sorgen dafür, dass die beiden Threads aus `BabyStepsConfig.count()` und `Controller.babyStepsAbbruch()` rechtzeitig abgebrochen werden, bspw. wenn zwischenzeitlich in die Refactoring- Phase gewechselt wurde oder wenn erfolgreich kompiliert wurde. `StopThread` sorgt dafür, dass frühere Threads abgebrochen werden. `Finished` sorgt dafür, dass der Thread in `Controller.babyStepsAbbruch()` stoppt, wenn der Thread in `BabyStepsConfig.count` abbricht, weil dann die Überprüfung der ablaufenden Zeit überflüssig wird. In der Main-Klasse wird über `setOnCloseRequest` sichergestellt, dass alle Threads stoppen, wenn das Fenster geschlossen wird.