Hochschule RheinMain
Studiengang Medieninformatik
Sommersemester 2022
Prof. Dr. Wolfgang Weitz

Softwaretechnik

Praktikumsblatt 2
29.04.2022

Um das Build-Tool **Gradle** (z.B. zum Management der Zusatzbibliotheken) zu nutzen, was cool und ratsam ist, legen Sie in Eclipse am besten von Anfang an in Eclipse ein „**Gradle Project**“ an. Sie können dann durch einfache Deklaration der benötigten (Bibliotheks-)Abhängigkeiten („dependencies“) in der gewünschten Version bzw. die Nutzung eines Plugins (z.B. für Java FX) in der Konfigurationsdatei **`build.gradle`** die benötigten Jars besorgen und einbinden lassen, was bei schon allein wegen JavaFX wirklich hilfreich ist.

Für die heutige Aufgabe können Sie sich ein **vorbereitetes Rahmen-Projekt** unter folgender Git-URL clonen:

`https://scm.mi.hs-rm.de/rhodecode/2022swttest/2022swttest00/GradleAmpel`

- Wählen Sie in Eclipse, ausgehend vom „File“-Menü:
  „File > Import... > Git > Project from Git (with smart import) > Clone URI“
- Geben Sie die o.g. URL an und authentifizieren Sie sich, wenn gefragt
- Nach Durchklicken der Dialogstrecke ist das Projekt in Ihrem Workspace
- Das Projekt ist git-seitig nach dem Clonen mit der o.g. URL als „origin“ assoziiert. Dorthin können Sie aber keine Änderungen pushen. Wenn Sie ein eigenes (leeres) Repository in Ihrem stud-Ordner angelegt haben, können Sie die push/fetch-URLs (also die Einstellung von „origin“) im Kontextmenü des Projekts unter „**Team / Remote / Configure push/fetch to Upstream...**“ und in der folgenden Dialogbox mit Hilfe des **„Change“-Buttons** neben der URI ändern.

Im vorbereiteten .gitignore wurden die **Eclipse-spezifischen Konfigurationsdateien** `.project` und `.classpath` und der Ordner `.settings` bereits **ausgeschlossen** (was richtig ist - am besten gleich **vor** dem ersten Commit in die „**gitignore**“-Liste aufnehmen), daher müssten Sie nach dem Clonen eines Gradle-Projekts aus dem Git **einmalig** in Eclipse über das Kontextmenü des Projekts „**Configure > Add Gradle Nature**“ ausführen, um die Eclipse-Gradle-Unterstützung für Ihr Projekt zu aktivieren.

Nach **Änderungen** an der `build.gradle` in Eclipse bitte **immer** die Kontextmenü-Funktion „**Gradle > Refresh Gradle Project**“ anstoßen. Wenn Sie das Projekt unter Versionskontrolle stellen (z.B. mit Git), denken Sie bitte daran, neben Ihren Sourcefiles bei Gradle-Projekten auch die Dateien `build.gradle`, `settings.gradle`, `gradlew` und den Ordner `gradle` mitzunehmen. Auf diese Weise könnten spätere Projektpartner nach Clonen des Git-Repositories gleich loslegen.

Die Verwendung von Tools wie Gradle ist für größere Projekte mit vielen Abhängigkeiten (benötigten Bibliotheken) unerlässlich. Bei der Ermittlung der richtigen „Koordinaten“ (Dependency-Name und Versionsnummer) hilft z.B. die Suchmaschine eines der Internet-Repositories, von denen Gradle fehlende Jars automatisch herunterlädt:
`https://search.maven.org`
Wenn Sie die gewünschte Bibliothek (in der gewünschten Version) gefunden haben, finden Sie auf der Detailseite unter „Gradle Groovy DSL“ auch einen „Textschnipsel“, den Sie per Copy&Paste direkt als Dependency-Eintrag in Ihre Datei `build.gradle` übernehmen können.

Aufgabe 1:

Entwickeln Sie bitte als Beispielanwendung für automatisiertes Testen einen Ampel-Simulator für Ampeln mit drei Farben. Die entsprechende Ampel-Klasse darf (nur) diese Methoden realisieren:

1. `isRot()`, `isGelb()` und `isGruen()` zur Abfrage,
   ob die betreffende Farbe gerade leuchtet oder nicht (`boolean`)
2. `reset()` zum Rücksetzen der Ampel auf den Ausgangszustand „rot“
3. `tick()` zum Umschalten auf die jeweils nächste Ampelphase

Es gibt also insbesondere *keine* „Setter“ für die einzelnen Farben (wenn man den Nutzern Ihrer Klasse Setter anbietet, machen die nur Blödsinn mit Ihrer Ampel, und ganz schnell ist das dann keine Lichtzeichenanlage mehr, sondern vielleicht gerade noch eine mittelmäßige Disco-Beleuchtung). Also:

- Bearbeiten Sie nachfolgend die Ampel-Klasse im Package `de.hsrm.mi.swt.ampel` (unter `src/main/java`). Verwenden Sie bitte während der Entwicklung ein neues Git-Repo in Ihrem stud-Ordner auf unserem Git-Server (die o.g. URL ist für Sie nur lesbar).
- Dazu gehört (unter `src/test/java`) im Package `de.hsrm.mi.swt.ampel` eine **Testklasse** `AmpelTest`, welche die (noch nicht ausgefüllte) `Ampel`-Klasse möglichst weitgehend überprüfen soll (viele, spezialisierte, zielgerichtete Testmethoden - *nicht* „eine komplexe“).
- Implementieren Sie bitte nun Ihre **`Ampel`-Klasse**. Lassen Sie dabei bitte häufig Ihre Testfälle durchlaufen (Testklasse-Kontextmenü : Run as... -> JUnit Test).
- Nach **1000-maligem Umschalten** sind die Glühbirnen der Ampel kaputt und müssen gewechselt werden. Bitte ergänzen Sie Ihre Ampel geeignet, dass sie nach 1000 Schaltvorgängen so lange bei jeder Operation eine `AmpelPuttException` wirft, bis die (neue) Methode `birnenWechsel()` aufgerufen und dann ein `reset()` ausgeführt wurde (bei den beiden in dieser Reihenfolge gibt es dann keine Exception). Schreiben Sie bitte wieder zunächst einen **Test**, der das gewünschte Verhalten dokumentiert, und ergänzen Sie dann Ihre Ampel-Klasse.
- Lassen Sie Ihre Tests auch einmal **außerhalb der Entwicklungsumgebung** aus der Shell mit Gradle ausführen (`./gradlew test`). Sie finden dann im von Gradle angelegten `build`-Ordner (ggf. mehrere Stufen) unterhalb des Unterordners „**`reports/tests/test`**“ eine `index.html`-Datei mit einem generierten Test-Report einschließlich verlinkter Detailseiten. Bitte probieren Sie dies auch mit einigen funktionierenden und (noch) fehlgeschlagenen Tests aus (ggf. zum Ansehen mal Fehler „einbauen“).

Aufgabe 2:

Realisieren Sie bitte zusätzlich (im gleichen Projekt) eine beampelte `AmpelKreuzung` (vier Ampeln). Verwenden Sie bitte Ihre Ampelklasse und die zugehörigen Tests weiter und legen Sie zum gemeinsamen Testen von Ampel- und Kreuzungsklasse bitte weitere Testklasse(n) in dem Java-Package `tests` an. Die `AmpelKreuzung` sollte *zumindest* eine `reset()`-Methode zur Herstellung eines definierten Anfangszustands, eine `tick()`-Methode zum gleichzeitigen Weiterschalten der Kreuzungsampeln und natürlich eine vernünftige Möglichkeit zum Zugriff auf die zugehörigen Ampeln haben, damit der Ampelkreuzungstest überprüfen kann, ob die Kreuzung insgesamt korrekt beampelt ist.
Versuchen Sie, mit möglichst wenig Aufwand eine vollständige Testung eines kompletten Kreuzungs-Schaltzyklus zu bewerkstelligen. Zur Einfachheit brauchen Sonder-Schaltphasen für Linksabbieger nicht berücksichtigt zu werden, und Fußgänger gibt es in diesem Szenario auch nicht. Durch eine einfache Symmetrieüberlegung können Sie relativ viel Aufwand einsparen. Jedes Mal alle Ampeln mit allen anderen zu vergleichen ist weder zum vollständigen Durchtesten nötig noch optimal.

Wenn Sie fertig getestet und noch etwas Zeit haben, können Sie Ihre JavaFX-Kenntnisse auffrischen, indem Sie der Ampel eine kleine GUI verpassen (drei „Lampen“ mit einem „reset“ und einem „tick“-Button). Hier eine build.gradle, die Testing und FX abdeckt:

```
plugins {
    id 'application'
    id 'org.openjfx.javafxplugin' version '0.0.12'
}
mainClassName = "de.hsrm.mi.swt.FxAmpelApp"
// Konfiguration des JavaFX-Plugins: Version und benötigte Module
javafx {
    // version = "17"
    modules = [ 'javafx.controls', 'javafx.fxml' ]
}
// Abzufragendes Internet-Repository für Bibliotheken (z.B. JavaFX)
repositories {
    mavenCentral()
}
// Abhängigkeiten (benötigte und einzubindende Bibliotheken)
dependencies {
    //implementation     'org.python:jython-standalone:2.7.2'
    testImplementation 'org.junit.jupiter:junit-jupiter-api:5.8.2'
    testRuntimeOnly    'org.junit.jupiter:junit-jupiter-engine:5.8.2'
}
// Initialisierung für JUnit-Testing
test {
    useJUnitPlatform()
}
// Bei Änderungen Projekt-Update nicht vergessen (Eclipse: rechte Maustaste - Gradle - Refresh Gradle Project)
```