Einführung in die Modifizierung von Galimulator mit Hilfe von Starloader

Verfasst von Geolykt

Erste Verfassung, Januar 2022

Inhaltsverzeichnis

1 Hintergrundinformation

1.1 Was ist Starloader überhaupt?

Starloader ist im Grunde genommen eine Reihung von Programmen, die erlauben Galimulator zu modifizieren. Im Herzen dieser Reihung ist der so genannte Starloader-Launcher, welcher das Starten von Java-Programmen mit einen modifizierten Klassenlader erlaubt. In den meisten Fällen wird es aber nur Galimulator sein, welcher von dem Launcher gestartet wird, da Starloader nur auf die Modifizierung von Galimulator ausgelegt ist. Das zweite Hauptelement welches wir in dieser Einführung verwenden werden ist die Starloader-API. Die Starloader-API ist eine Softwarebibliothek, die erlaubt, mit Galimulator zu kommunizieren, ohne dies direkt zu tun. Das direkte Kommunizieren mit Galimulator ist in den meisten Fällen gefährlich, da die Namen von Methoden, Klassen und Felder innerhalb von Galimulator zum Teils “verschlüsselt” sind

und sich mit jedem Update stark verändern können. Ein weiterer Aspekt ist, dass mit einer zunehmenden Anzahl von Modifizierungen die Wahrscheinlichkeit einer Inkompatibilität steigt. Die Starloader-API versucht diese Inkompatibilitäten zu beseitigen. Aus diesen Gründen werden wir in der Einführung stets Starloader-API verwenden, auch wenn es nicht unbedingt nötig wäre.

1.2 Welches Wissen wird erwartet?

Diese Einführung geht davon aus, dass sowohl Starloader als auch Starloader-API installiert sind. Ebenso wird ein Grundwissen von Java vorausgesetzt, fortgeschrittene Kenntnisse sind aber nicht unbedingt notwendig. Zudem wird eine Maven-Installation (siehe Kapitel 2.1) empfohlen, sollte es keine Präferenz zu ein ähnliches Werkzeug geben.

1.3 Konventionen

Konventionen unterscheiden sich oft zwischen Person und Person. Daher ist es Anzumerken, dass dieses Dokument Konventionen verwenden könnte, die für den einen Unüblich ist. Oft sind Konventionen "überflüssig", Sie sollten aber nicht außer Acht gesetzt werden. Es werden auf Konventionen, die in den Kreis des Themas als wichtig empfunden werden, werden in diesem Dokument begründet aufgegriffen.

2 Kleines Hallo Welt

2.1 Mit Starloader kompilieren

Ohne irgendwelche Konfiguration hat der Java-Compiler keine Ahnung, was Starloader ist. Dies ist ein Problem, da der Versuch, Starloader-spezifische Klassen zu verwenden scheitern würde. Dies am Einfachsten mit Maven oder Gradle gelöst werden. Wenngleich es alternativen gibt, so sind diese entweder Zeitaufwendig oder nur auf das eigene System funktionstüchtig. In dieser Einführung wird Maven verwendet, da Maven in der Regel weniger Anfällig für Spontane Probleme ist. Wenn es eine Präferenz zu Gradle gibt, kann auch Gradle verwendet werden, obwohl man wissen sollte, wie sein Werkzeug verwendet wird. Um Maven zu erklären, wie das Projekt zu Kompilieren ist, muss eine pom.xml erstellt werden. Eine pom, die Starloader als Abhängigkeit erklärt, sieht wie folgt aus:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.example</groupId>
  <artifactId>beispiel</artifactId>
  <version>1.0</version>
  <name>Beispiel</name>

  <repositories>
    <repository>
      <id>geolykt-maven</id>
      <url>https://geolykt.de/maven</url>
    </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>de.geolykt</groupId>
      <artifactId>starloader-api</artifactId>
      <version>1.5.0</version>
      <scope>provided</scope>
      <exclusions>
        <exclusion>
          <groupId>org.spongepowered</groupId>
          <artifactId>mixin</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <release>11</release>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Dies sieht Komplex aus, daher wird jetzt alles im Detail erklärt:

- Die ersten vier Zeilen sind für den Anwender nicht interessant, es ist aber Konventionell diese zu haben. Spezifischer ist nur "<project>" notwendig, alles andere ist nur Information für andere Programme, die nicht unbedingt Wissen, was es mit der Datei auf sich hat.

- "<groupId>org.example</groupId>" definiert die Gruppe des Artefakts. Der Wert ist für den Normalen Anfänger unwichtig, wird aber wichtig sobald das Projekt veröffentlicht werden soll. Die Gruppe sollte meistens eine Domain in umgekehrter Reihenfolge sein. Also "example.org" wird "org.example" und "subdomain.google.com" wird "com.google.subdomain". Da viele nicht im Besitz einer Domain oder Subdomain sind, verwenden viele Werte wie "me.musternname", dies kann unter seltenen Umständen aber zu Probleme führen.

- "<artifactId>beispiel</artifactId>" definiert das Kennzeichen des Artefakts. Der Wert sollte nur aus Kleinbuchstaben und den Zeichen "-" und "_" bestehen. Leerzeichen sind nicht erlaubt. Standardmäßig ist der Wert der Name des Projekts in Kleinbuchstaben, wobei mehrere Wörter mit "-" verbunden werden.

- "<version>1.0</version>"definiert die Version des Artefakts. Auch dies ist erst wirklich Wichtig, wenn das Projekt in der Öffentlichkeit ist. In den Meisten Fällen sollte der Wert die Semantische Versionierung befolgen, aber auch andere Schemas sind möglich.

- Der Abschnitt zwischen "<repositories>" und "</repositories>" ist wiederum Wichtig, da es Maven erklärt, wo die Abhängigkeiten zu finden sind. In diesem Fall kann Maven die Abhängigkeiten unter der URL "`https://geolykt.de/maven`" auffinden. Maven zudem such auch auf Maven Central sowie dem eigenen PC nach Abhängigkeiten. Sollte dieser Abschnitt wegfallen wüsste Maven unter Umständen nicht mehr, wo die Abhängigkeiten sind und die Kompilation würde scheitern.

- Der Abschnitt zwischen "<dependencies>" und "</dependencies>" erklärt Maven welche Abhängigkeiten verwendet werden. Mit den Anweisungen zwischen "<exclusions>" und "</exclusions>" wird eine Abhändigkeit heraus genommen, die nur für fortgeschrittene Benutzer interessant wird und für andere nur ein Stolperstein ist. In unserem Falle ist die Starloader-API eine Abhängigkeit, wobei alle Abhängigkeiten von dieser Abhängigkeit miteinbezogen werden. Dadurch müssen wir nicht explizit Starloader-Launcher und libGDX als Abhängigkeit spezifizieren.

- Der letzte Abschnitt zwischen "<build>" und "</build>" definiert, dass Maven mit Java 11 kompiliert. Das führt dazu, dass Java 10 und niedriger nicht im Stand ist das Projekt zu kompilieren. Java 12 und höher sind aber im Stande. Grund für diesen Abschnitt ist, dass ein großer Teil von

Starloader auf Java 11 läuft und eine Kompilation mit Java 8 eine sinnlose Verschwendung von Features, die zwischen Java 8 und 11 gekommen sind, wäre.

Wenn das Maven-Projekt mit mvn packagecompiliert wird, sollte nun der Compiler wissen, dass Starloader als Abhängigkeit zu verwenden ist.

2.2 Eine Extension deklarieren

Die pom.xml ist noch nicht alles, damit Starloader-Launcher die Mod als Mod entdecken kann. Zudem gibt es überhaupt kein Programm, das Starloader ausführen könnte. Daher werden die Ordner "src/main/resources" und "src/main/java" erstellt. Innerhalb "src/main/resources" wird die Datei "extension.json" erstellt. Der Inhalt dieser Datei sollte ungefähr so aussehen:

```
1 {
2         "entrypoint": "org.example.beispiel.Beispiel",
3         "name": "Beispiel",
4         "version": "1.0",
5         "dependencies": ["StarloaderAPI"]
6 }
```

In diesem Fall gibt es weniger uninteressantes, da alle Einträge von Starloader-Launcher verwendet werden.

- Der "entrypoint"-eintrag weist auf den Haupteinstiegspunkt der Mod. Die Klasse auf der zugewiesen wird, wird später genauer untersucht. Der Wert sollte stets den Package der Klasse sowie den Klassennamen haben, der Default-Package wird zwar unter Starloader erlaubt, es wird aber abgeraten, ihn zu verwenden, da die Wahrscheinlichkeit einer Kollision sich drastisch erhöht, wenn Packages entfallen.

- "name" definiert den Namen der Mod. Es sollte unter keinen Umstand den Namen einer anderen Mod teilen, da es nicht möglich ist, zwei Mods mit dem selben Namen gleichzeitig zu Verwenden.

- "version" definiert die Version der Mod. Es ist größtenteils Ästhetisch, kann aber für den Benutzer der Mod hilfreich sein.

- "dependencies" gibt die Abhängigkeiten der Mod an. Abhängigkeiten werden stets vor der Mod gestartet. Es ist auch wichtig zu wissen, dass eine Abhängigkeit immer vor der Abhängigen Mod heruntergefahren wird. Da in dieser Einführung die Starloader-API verwendet wird, wird diese als Abhängigkeit definiert.

- Die extension.json-Datei erlaubt auch ein paar andere Einträge, diese sind aber nur für Fortgeschrittene Benutzter brauchbar.

Nachdem die Datei angelegt ist kann Starloader erkennen, dass es sich um eine Mod handelt, dennoch würde die Mod nicht laden, da der Programmcode fehlt. Hierfür muss die Klasse definiert werden, welche der Haupteinstiegspunkt der Mod sein soll. Daher wird innerhalb des "src/main/java" Ordners die Klasse "org.example.beispiel.Beispiel" angelegt, wobei der Name sich nach dem "entrypoint" Eintrag richten muss. Diese Klasse muss "public" sein und die Klasse de.geolykt.starloader.mod.Extensionerweitern. Da in Starloader-Launcher 1.1 (welche von der Starloader-API gegeben wird) die Methoden **void** initialize () und **void** terminate() abstrakt sind, müssen diese implementiert werden. Die Klasse darf zudem keine Parameter im Konstruktor haben. Zum Schluss sollte die Klasse ungefähr wie folgt aussehen:

```
1  package org.example.beispiel;
2
3  import de.geolykt.starloader.mod.Extension;
4
5  public class Beispiel extends Extension {
6      @Override
7      public void initialize() {
8
9      }
10
11     @Override
12     public void terminate() {
13
14     }
15 }
```

Die **void** terminate() gilt als unsicher und sollte in fast jeder Situation gemieden werden. Dies kommt zu Stande, da diese Methode häufig nur Aufgerufen wird, wenn die JVM sich im Shutdown befindet. Würde es in dieser Methode zu einem Deadlock kommen, würde die JVM nicht herunterfahren können. Zudem ist die Starloader-API innerhalb dieser Methode nicht verwendbar, da der Launcher die Starloader-API schon heruntergefahren hat.

Wenn das Projekt jetzt mithilfe von mvn package kompiliert wird, kann die Jar aus dem "target" Ordner verwendet werden, um eine Mod zu installieren.

2.3 Logging unter Starloader

Derzeit tut diese Mod aber nichts. Eine einfache Art und Weise ein Lebenszeichen zu Zeigen ist das sogenannte logging. Eine populäre Art und Weise dies zu tun ist System.out.println. Auch wenn dies funktioniert, so ist die empfohlene Methode de.geolykt.starloader.mod.Extension.getLogger. Dies hat den Vorteil, dass der Anwender weiß, woher ein Logeitrag kommt. Es ist auch leicht ablesbar, wann der Eintrag stattfand und wie schwerwiegend er ist. Der Code der benötigt wird, um "Hallo Welt" auszugeben ist also getLogger().info("Hallo␣Welt");.

Würde man dies während der Initialisierung, also in der **void** initialize ()-methode tun, würde die initalize Methode wie folgt aussehen:

```
1     @Override
2     public void initialize() {
3         getLogger().info("Hallo Welt");
4     }
```

Hiermit wäre das einfache Hallo-Welt-Programm fertig.

3 Events

Häufig soll ein Programm nur etwas tun, wenn etwas Bestimmtes passiert. Events (wörtlich Ereignisse) ermöglichen dies in einer Recht einfachen Art und Weise. Die Starloader-API bietet Zahlreiche Events an, auf der die Mod antworten kann.

3.1 Auf Events Lauschen

Um auf einen Event zu lauschen muss Zuerst eine neue Klasse erstellt werden, die de.geolykt.starloader.api.event.Listener implementiert. Diese Klasse muss **public** sein. Innerhalb dieser Klasse muss eine Methode deklariert werden, die auch **public** ist und nur einen einzigen Parameter hat. Der Typ des Parameters muss identisch mit dem Datentyp des Event sein, auf den gelauscht werden soll. Zudem muss die Methode mit de.geolykt.starloader.api.event.EventHandler annotiert sein. Um auf den echten Start der Applikation zu lauschen wird de.geolykt.starloader.api.event. lifecycle .ApplicationStartEvent verwendet. Eine Klasse, die auf diesen Event lauscht sieht wie folgt aus:

```
1  package org.example.beispiel;
2
3  import de.geolykt.starloader.api.event.EventHandler;
4  import de.geolykt.starloader.api.event.Listener;
5  import de.geolykt.starloader.api.event.lifecycle.*;
6
7  public class BeispielLauscher implements Listener {
8    @EventHandler
9    public void lauscheStart(ApplicationStartEvent event){
10     // Hier wird gelauscht
11   }
12 }
```

Damit die Starloader-API weiß, dass dieser Lauscher auch Lauschen soll muss er Registriert sein. Dafür ist die Methode "de.geolykt.starloader.api.event .EventManager#registerListener(Listener)" geeignet. In dem Fall des Einfüh-

rungsbeispiel würde dies als "EventManager.registerListener(new BeispielLauscher ());" innerhalb der "Beispiel#initialize" Methode manifestieren.

3.2 Events Abbrechen

Einige Events implementieren die de.geolykt.api.event.Cancellable Interface. Diese Events können „abgebrochen" werden, wobei dann in der Regel der Grund, weshalb der Event verteilt wurde, nicht mehr stattfindet. Das heißt, wenn ein Lauscher den StarOwnershipTakeoverEvent (wörtlich SternEigentümerÜbernommenEreignis - wobei kritisch betrachtet der Name etwas falsch ist, da sich der Eigentümer eines einzigen Sterns sich ändert und der Eigentümer selbst nicht übernommen wird) lauscht und ihn Abbricht, so findet kein wechsel des Eigentümer statt. Um also alle Wechsel des Eigentümers zu verhindern und somit eine sehr langweilige Welt zu erschaffen wird eine Methode wie folgt benötigt:

```
1   @EventHandler
2   public void lauscheWechsel(
3           StarOwnershipTakeoverEvent evt) {
4     event.setCancelled(true);
5   }
```

Der Fakt, dass der Parameter der Methode auf einer Zeile für sich ist ist irrelevant und ist nur ein Kompromiss, da wenn der Kopf der Methode in einer einzigen Zeile wäre, es vollkommen grässlich für den Leser wäre. In einem echten Programm sollte so etwas gemieden werden.

3.3 Eigene Events definieren

Von Zeit zu Zeit ist es vorteilhaft selber Events zu definieren. Dafür ist als erstes eine Klasse benötigt, die als Datentyp des Events fungiert. Diese Klasse muss de.geolykt.starloader.api.event.Event (oder eine Subklasse von dieser Klasse) als Superklasse haben. Die Klasse für den Event „BeispielEreignis" würde ähnlich wie folgt aussehen:

```
1 package org.example.beispiel;
2
3 import de.geolykt.starloader.api.event.Event;
4
5 public class BeispielEreignis extends Event {
6   public BeispielEreignis() {
7     super();
8   }
9 }
```

Technisch gesehen ist der „super()" Aufruf nicht benötigt, ist aber aus Gründen der Vollständigkeit in dem Beispiel beinhaltet. Die Klasse kann auch beliebig

viele (sofern von Java erlaubt) Felder und Methoden besitzen. Häufig ist es sogar angebracht, etwas mehr als das hier ersichtliche Minimale zu haben.

Neben die Definition der Klasse muss der Event auch ausgegeben werden. Auch hierfür ist EventManager zuständig aber hierbei hat der Entwickler zwei Möglichkeiten. entweder die „handleEvent“ wird verwendet oder die „handleEventExcept“ wird verwendet. Im Idealfall machen beide Methoden das selbe. Der Unterschied ist aber spürbar, wenn ein Fehler innerhalb eines Lauschers auftritt (oder der Lauscher selbst falsch definiert wurde). Bei handleEvent werden alle Fehler innerhalb aller Lauscher in der Konsole ausgegeben, die Applikation ignoriert also die Fehler. Hingegen wird bei handleEventExcept schon bei dem ersten Fehler abbgebrochen und der Fehler wird als Rückgabewert ausgeben. Erfolgt kein Fehler ist der Rückgabewert **null**. „handleEvent“ gibt kein Rückgabewert aus, also ist es nicht möglich, zu wissen ob ein Fehler ausgetreten ist. Dadurch, dass dies oft nicht benötigt wird, wird „handleEvent“ häufiger verwendet. Demzufolge gibt man den Beispiel-Ereignis mit

EventManager.handleEvent(new BeispielEreignis());

oder Ähnlich aus. Um ein Event abbrechbar zu machen kann ganz einfach die Cancellable-Interface implementiert werden. Dabei ist jedoch die implementierende Klasse zuständig, die Interface richtig zu implementieren, andernfalls würde es nicht wirklich abbrechbar sein.

3.4 Wie funktionieren Lauscher?

Derzeit (also in SLAPI 1.5) werden die Methoden der Lauscher-Klasse mithilfe von Java's „Reflection“ API entdeckt und aufgerufen. Da eine dynamische Entdeckung der Lauscher-Methoden sehr Rechenintensiv ist, werden die Methoden nur einmal für jeden Lauscher entdeckt. Die entdeckten Methoden sowie der Lauscher werden in einer Liste gespeichert und wenn später ein Event ausgegeben wird, wird ein Baum generiert, der alle registrierten Methoden zusammenführt. Genauer genommen werden sechs Bäume generiert, eins für jede Priorität. Wenn ein Event ausgeben wird, wird mit Hilfe jedes dieser Bäume kontrolliert ob es für die Klasse des Events einen Lauscher gibt. Wenn ja, wird dies zugehörige Methode aufgerufen. Um auf Subklassen lauschen zu können, wird während der Generierung der Bäume auch die Superklassen der Lauscher-Methoden in Betracht genommen. Unpassende Klassen werden still ignoriert, weshalb es nicht möglich ist, auf Interfaces oder Klassen, die keine Events sind, zu lauschen. Dies führt jedoch dazu, dass mit einer hohen Anzahl an Lauscher die Zeit, die benötigt wird, um die Bäume zu generieren steigt. Um dies zu verhindern sollte die Registrierung von Lauschern nach dem ApplicationStartEvent vermieden werden.

Es kann jedoch sein, dass später effizientere Mechanismen verwendet werden, jedoch müssten diese Mechanismen kompatibel mit der derzeitigen API sein. Dass die privaten Methoden der EventManager-Klasse unwahrscheinlich Fortbestehen werden ist aber mittlerweile klar. Zum Beispiel könnte eines dieser späteren Mechanismen Lambdas anstatt 100% Reflections verwenden. Es kann aber auch möglich sein, dass Mithilfe von Bytecode-Transformer die Bäu-

me innerhalb der Events verlagert werden, dennoch wird sicher genommen, dass die Prozeduren, wie sie in Kapitel 3.3 auf Seite 8 beschrieben werden, ihre Gültigkeit beibehalten.