Petri Netz Simulator Implementierungsbeschreibung

Michael Birkhoff

Petri_Net_State

Vor der Implementierung des Petri Netz Projektes überlegte ich mir eine geeignete Datenstruktur, um einzelne Netze ihren Eigenschaften zu speichern.
Ein Netz sollte einen Namen und je eine Menge aus Knoten, Transitionen, Kanten und *Properties* enthalten, wofür sich eine HashMap anbot um einfach auf die einzelnen Elemente zugreifen zu können.
Die Menge der Kanten habe ich aufgeteilt um Verwechslungen von ein- und ausgehenden Kanten zu vermeiden, außerdem erleichterte es Debugging, da direkt zwischen den beiden Kantenarten unterschirden werden konnte.

```
{:name "Name", :properties #{}, :vertices {}, :transitions {}, :edges_in #{}, :edges_out #{}}}
```

Eine Anforderung an die Netze war es zudem zwei verschiedene Netze zu einem neuen zu vereinigen, was potentiell zu Problemen führen könnte sollte es identische Namen von einzelnen Knoten oder Transitionen geben.
Anfangs Prefixte ich dazu die Namen der einzelnen Netze mit dem Ursprungsnetznamen. Allerdings kamen so nach schon zwei Vereinigungsoperationen Namen wie A_B_Netz_A_v-a zustande. Daraufhin entschied ich mich dazu intern einen Hash-Wert für die referenzierung zu nutzen. Die Darstellung eines Knoten änderte sich somit von einem Tupel zu einem Triple

anfangs: ['v-a' 10] nach der Änderung: [:1599435 'v-a' 10]

Somit musste ich bei einer Vereinigung von zwei Netzen nicht auf die Herkunft eines Knoten oder einer Transition achten sondern konnte direkt alle Mengen vereinigen und musste lediglich am Ende einmal über alle Knoten und Transitionen einen neuen Hash-Wert mappen um zu gewährleisten, dass das neue Netz eigene Hash-Werte für seine Mengen besitzt.
Zudem konnte man Namen von Transitionen oder Knoten im Nachhinein einfach ändern ohne die Funktionalität des Netzes zu beeinflussen.

Der State wird in einem Atom gespeichert welcher eine HashMap von Petri Netzen enthält. Die Wahl fiel auf Atome da ich nicht plante die Anwendung von mehreren Benutzern oder Threads gleichzeitig zu nutzen.

Die Funtkionalität des petri_net_state umfasst das Erstellen und Hinzufügen von Petri Netzen, Knoten, Kanten und Transitionen. Zudem können einzelne Netze entfernt, vereinigt oder kopiert werden. Das Hinzufügen und Verarbeiten von Properties habe ich in den Simulator ausgelagert, da ich empfand, dass diese Funktionalität über die Basis Eigenschaften von Petri Netzen hinausging und diese außerhalb des Simulators auch keine Funktionalität erfüllen.
Zudem stand deren finale Form der Darstellung lange nicht fest stand.

Simulator

Der Simulator beinhaltet die Funktionalität Transitionen zu feuern und Properties einem Netz hinzuzufügen und auszuwerten.
Zuerst habe ich Properties als gequotete Funktionen übergeben, was allerdings keine gute Idee ist, da sich Funktionen nach einer Vereinigung von zwei Netzen als ungeeignete Datenstruktur erwiesen um nachträglich Einträge zu ändern. Dadurch wurde dieser Ansatz zu Gunsten einer weiteren HashMap verworfen die den Typ und die Argumente der Funktion speichern.

```
{:type :transition_alive, :args ("y")}
```

Der Vorteil war nun, dass man keine eigene DSL einführen musste um logische Junktoren einzubauen sondern lediglich mit Multimethods den Typ ableiten konnte.

```
{:type :or,
 :args   ({:type :non_empty, :args ("a")} {:type :non_empty, :args ("b")})}
```

Mein Programm habe ich stets mit Tests abgesichert, was das Refactoring sehr simpel gestaltete.

GUI

Nach meiner Erfahrung mit Swing war ich äußerst positiv überrascht wie gut Clojure mithilfe von Seesaw einen guten Wrapper zur GUI Programmierung zur Verfügung stellt.
Es gab allerdings oft Momente in denen ich es nicht offensichtlich fand, wann manche Aufrufe ausgewertet wurden und andere nicht, was leider zu vielen Workarounds führte, deren Ursache hauptsächlich Unkenntnis des Frameworks zugrunde lag.
Dadurch hätten viele Aspekte meiner Implementierung mit mehr Erfahrung deutlich besser realisiert werden können.
Trotz alle dem habe ich bisher noch nie so eine angenehme Swing Erfahrung genossen.

GAME

Dieses Clojure Programm entstand zunächst aus purer Spielerei allerdings wurde mir anhand dessen erst die Tragweite der Genialität Clojures vollends bewusst.
Die Möglichkeit Funktionen zur Laufzeit hinzuzufügen und zu ändern ist einfach das Killer Feature! So wird es ermöglicht ohne vorher das Programm neu zu kompilieren Elemente direkt hinzuzufügen, ohne dadurch wieder im Initialzustand zu landen.
Man kann direkt Code ändern und sehen was sich zur Laufzeit ändert, was gerade bei grafischen Oberflächen sehr komfortabel ist, da in diesem Bereich automatisierte Tests nur schwer realisierbar sind.
Zudem ist Multi Threading einfach zu benutzen wie nie und man kann somit sehr schnell Schleifen erzeugen die Seiteneffekte verursachen. Auch wenn Seesaw kein geeignetes Framework ist um ein Spiel darzustellen konnte ich mit wenig Aufwand sich bewegende Sprites erzeugen.