Präsentation

Ziel dieser Phase war der Entwurf einer objektorientierten Architektur für unser Simulationsprogramm.

Zunächst ein kurzer Nachtrag zur Spezifikation: TODO

Kommen wir zur Programmarchitektur. Unser Entwurf basiert im Wesentlichen auf dem Model-View-Controller Prinzip, besteht also aus drei großen Oberpaketen: View, Controller und Model. Die View informiert den Controller über Interaktionen des Nutzers mit der UI. Dieser verwendet denn gegebenenfalls das Model, um die Benutzeranfragen zu bearbeiten, also etwa um eine Simulation zu starten oder eine Konfigurationsdatei zu laden. Da die Durchführung im Allgemeinen durchaus lang dauern kann, werden Simulationen asynchron gestartet. Der Controller beobachtet dann das Model, um über Neuigkeiten im Simulationsprozess informiert zu werden und gegebenenfalls die View entsprechend zu aktualisieren.  
Wie man sieht sind durch diesen Entwurf View und Model komplett voneinander entkoppelt und wir haben das Model so entworfen, dass es keine Annahmen darüber trifft, wie es verwendet wird; dadurch wird das Model wiederverwendbar.

Als UI-Framework verwenden wir JavaFX, das heißt die GUI ist vollständig in xml beschrieben und die View-Klassen werden zur Laufzeit daraus erzeugt. Mit jedem der Fenster ist ein Controller assoziiert, der zusammen mit dem Fenster erzeugt wird und dessen Nutzereingaben verarbeitet.

* Controller

Die Controller sind in einer baumartigen Hierarchie angeordnet. Oben steht der HeadController. Dieser gehört zum Startfenster und wird damit gleich zu Programmstart erzeugt. Er ist außerdem die Schnittstelle des Controllers zum Model. Darunter befinden sich Controller für die verschiedenen Popup-Fenster wie das Konfigurationsfenster oder die Fenster zur Erstellung von Strategien, Populationen, etc. sowie für die Ergebnisausgabe im Hauptfenster. Wenn der Nutzer in einem der Fenster eine neue Gruppe, Strategie etc. erstellt hat, erzeugt der entsprechende Controller aus den Nutzereingaben eine Instanz des entsprechenden Objekts. Die neu erstellte Instanz soll dann dem Model zur Verfügung gestellt werden, dazu muss also der HeadController benachrichtigt werden. Dazu implementieren diese Controller das generische CreationController-Interface für die von ihnen erzeugte Klasse, die die Registrierung eines callbacks bei der Erzeugung einer neuen Instanz ermöglicht. Der HeadController schreibt die Instanz dann in das zentrale Repository im Model, dazu später nochmal mehr.  
Neben dem Erstellen von Populationen etc. ist natürlich die wichtigste Funktion des Models das Ausführen von Simulationen. Dazu wollen wir zunächst anhand eines Sequenzdiagramms das Starten einer Simulation veranschaulichen.

* Seq.

Die Schnittstelle des Models zum Starten und Stoppen von Simulationen ist der Simulator. Dieser stellt exemplarisch eine startSimulation-Methode zur Verfügung, die eine Instanz der Klasse UserConfiguration entgegennimmt. Diese ist eine direkte Abbildung der Einstellungen, die der Nutzer im Konfigurationsfenster vornimmt, kann also insbesondere auch eine Multikonfiguration sein. Um daraus die zugehörigen elementaren Konfigurationen zu gewinnen, benutzt der Simulator den ConfigurationCreator. Handelt es sich nicht um eine Multikonf., enthält die Collection also nur ein Element. Eine Instanz der Configuration Klasse enthält nun genau die Informationen, die zur Durchführung einer einzelnen Wiederholung benötigt werden.  
//Erstellung SimulationResult, füllt sich im Laufe der Zeit  
Für jede der elementaren Konfigurationen und dann jeweils für jede durchzuführende Wiederholung erzeugt der Simulator eine Instanz der SimulationEngine und startet diese asynchron mit der entsprechenden Configuration.  
…  
Das SimulationResult Objekt bietet die Möglichkeit, einen UpdateHandler zu registrieren, der immer aufgerufen wird, wenn das Ergebnis einer Wiederholung vorliegt.  
…  
Diese Klassen bilden, außer der SimulationEngine und natürlich dem Controller, das Unterpaket Simulator vom Model, hier nochmal ein Klassendiagramm dazu. Wie man sieht, ist der Simulator ein Interface. Wir planen, die Wiederholungsausführung mittels ThreadPools zu parallelisieren, aber natürlich wäre auch eine sequentielle Implementierung denkbar.

Wie wir jetzt schon gesehen haben, kapselt die SimulationEngine Klasse die Durchführung einer einzelnen Wiederholung. Wie genau sie das tut, wollen wir wieder anhand eines Sequenzdiagramms vorstellen.  
🡪Seq.  
Zur Initialisierung der Agenten holt sich die Engine bei der Konfiguration die Segmente. Da für die Simulation nur relevant ist, wieviele Agenten und wie initialisiert werden sollen, ist die darüberliegende Gruppen- und Populationsstruktur irrelevant. Die Segmente werden einer Instanz …  
Die variablen Events im Simulationsablauf, namentlich der Paarbildungsalgorithmus,… sind jeweils hinter einem Interface versteckt und werden der Engine gemäß des Strategie-Entwurfsmusters von der Configuration übergeben.  
…  
Diese Klassen bilden das Unterpaket SimulationEngine vom Model. Zu den Interfaces PairBuilder, etc. gibt es natürlich die entsprechenden Implementierungen, die bereits im Pflichtenheft beschrieben wurden, also bspw. … . Um den Simulationsablauf möglichst frei konfigurierbar zu machen, ermöglichen wir es dem Nutzer, eigene Implementierungen der Interfaces zu schreiben und per Plugin zu integrieren. Dazu haben wir folgendes Pluginsystem entworfen 🡪 Pluginklassendiagramm

Das Problem, auf das wir zunächst gestoßen sind, ist das die per Plugin hinzugefügten Klassen auch parametrisierbar sein sollen und der Nutzer diese Paramter im Konfigurationsfenster festlegen können soll. Dazu reicht es eben nicht, eine direkte Implementierung der Interfaces als Plugin zu verwenden. Unsere Lösung ist es, als Plugin eine Wrapper-Klasse zu verwenden, welche auf Anfrage parametrisierte Instanzen der Klasse zur Verfügung stellt, die die tatsächliche Funktionalität implementiert. Um die Parametrisierung zu ermöglichen, enthält die Wrapper-klasse zudem eine Liste von nötigen Parametern. Ein Parameter wird durch einen Namen, eine Beschreibung und einen Wertebereich beschrieben.  
Die nächste Frage war, wie wir die Konfiguration des Plugins im Konfigurationsfenster ermöglichen, wenn jedes Plugin eine eigene Parametrisierung haben kann. Dazu stellt das Plugin einen PluginRenderer zur Verfügung. Dabei handelt es sich um eine abstrakte Fabrik, welche Plugin-Controls erzeugt. Diese sind direkte Unterklassen des javafx.layout.panes und können damit dynamisch vom Controller in die UI eingebettet werden. Wird nun etwa ein durch ein Plugin eingebundener Paarbildungsalgorithmus ausgewählt, so wird unter dem Dropdown-Menü das entsprechende PluginControl objekt eingebettet, das dann bspw. Textfelder und Slider zur Festlegung der nötigen Parameter beinhaltet. Auf Anfrage gibt das PluginControl objekt die Werte der eingestellten Parameter zurück, und zwar in der Reihenfolge, wie sie dem Plugin übergeben werden können, um eine entsprechend parametrisierte Instanz zu erhalten. Damit man nicht für jedes Plugin einen eigenen Renderer schreiben muss, wird standardmäßig ein generischer Renderer zurückgegeben, der für jeden Parameter ein Textfeld mit entsprechendem Namen anzeigt.  
Die Plugins werden beim Start vom Pluginloader über die Java-Serviceloader-API geladen.

Die geladenen plugins werden in einem zentralen Repository abgespeichert.

* Repository

Das Repository ist als Singleton implementiert. Neben den Plugins speichert es auch alle im System hinterlegten Strategien, Spiele, Gruppen und Populationen. Das Repository-Paket beinhaltet zudem auch noch den FileLoader. Dieser wird beim Start verwendet, um Strategien, Populationen, Gruppen und Spiele aus einer dafür angelegten Ordnerstruktur zu laden. Außerdem wird er verwendet, wenn der Nutzer zur Laufzeit Konfigurationen, Gruppen etc. lädt oder speichert.

Warum nicht direkt PluginControl vom Plugin: So können PluginRenderer wiederverwendet werden und eine Standard-Implementierung angeboten werden. Außerdem wird dadurch die Initialisierung des layout panes aus dem Plugin ausgelagert