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 konnten das Model so entwerfen, 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. Der HeadController 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 sowie für die Ergebnisausgabe im Hauptfenster.  
Wenn der Nutzer eine neue Gruppe, Strategie etc. erstellt, erzeugt der entsprechende Controller aus den Nutzereingaben eine Instanz des entsprechenden Objekts. Die neu erstellte Instanz soll dann dem Model zur Verfügung gestellt werden, wozu also der HeadController benachrichtigt werden muss. 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. Wie eine solche Interaktion mit dem Model aussieht, wollen wir anhand eines Sequenzdiagramms vorstellen. 2:42

* Seq.

Die Schnittstelle des Models zum Starten und Stoppen von Simulationen ist der Simulator. Dieser stellt eine Methode startSimulation 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. Eine Instanz der Configuration Klasse enthält dann nur noch die Informationen, die zur Durchführung einer einzelnen Wiederholung benötigt werden.  
Der Simulator erstellt dann eine Instanz der Klasse SimulationResult. Diese ist zunächst leer und füllt sich im Laufe der Zeit mit den Ergebnissen der einzelnen Wiederholungen.  
Für jede der elementaren Konfigurationen und dann jeweils für jede durchzuführende Wiederholung erzeugt der Simulator nun eine Instanz der SimulationEngine und startet diese asynchron mit der entsprechenden Configuration. Diese führt eine einzelne Wiederholung zu der gegebenen Konfiguration aus und gibt deren Ergebnis dann als IterationResult zurück.  
Nachdem der Simulator alle Wiederholungen gestartet hat, gibt er das wie gesagt zunächst leere SimulationResult Objekt zurück.  
Dieses bietet die Möglichkeit, einen callback zu registrieren, der immer aufgerufen wird, wenn das Ergebnis einer Wiederholung vorliegt.  
Ist nun also eine Wiederholung abgeschlossen, schreibt der Simulator deren Ergebnis in das entsprechende SimulationResult objekt, welches dann, falls welche registriert wurden, die entsprechenden callbacks aufruft. 4:50  
Diese Klassen bilden, außer der SimulationEngine und natürlich den Konfigurationsklassen, 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.  
Für die Initialisierung muss die Engine wissen, wie viele Agenten und jeweils mit welcher Strategie- und Kapitalverteilung sowie mit welcher Gruppenzugehörigkeit erzeugt werden müssen. Diese Information steckt vollständig in den Segmenten, die die Engine bei der Configuration-Instanz abfragt. Die Segmente werden einer Instanz der AgentInitializer-klasse übergeben, die die Agenten der Segmente erzeugt und initialisiert.  
Die Engine erzeugt dann eine Instanz der SimulationHistory-Klasse. Diese speichert stets die Ergebnisse der bisherigen Spiele des aktuellen Adaptionsschritts. Das ist notwendig, da etwa Strategien wie tit-for-tat oder grim darauf bezug nehmen, wie sich die anderen Agenten in vorangegangenen Spielen verhalten haben.  
Die variablen Bausteine im restlichen Simulationsablauf, namentlich der Paarbildungsalgorithmus,… sind jeweils hinter einem Interface versteckt und werden der Engine gemäß des Strategie-Entwurfsmusters von der Configuration übergeben.  
Nachdem die Initialisierung abgeschlossen ist, werden so lange Adaptionsschritte ausgeführt, bis ein Gleichgewicht oder die maximale Zahl durchzuführender Adaptionsschritte erreicht ist. Die Ausführung eines Adaptionsschrittes haben wir der Übersichtlichkeit halber ausgelagert. Wie sie bereits wissen, besteht ein Adaptionsschritt aus einer festen Anzahl von Runden. In jeder Runde werden dann die Agenten zunächst gemäß des gegebenen Paarbildungsalgorithmus zu Paaren zusammengefasst. Ein solches Matching wird repräsentiert durch eine Collection von AgentPair-Objekten. Für jedes dieser Paare wird dann das zugrundeliegende Stufenspiel gespielt.  
Dazu werden bei dem AgentPair-Objekt die beiden Agenten und dann bei den Agenten deren Strategien abgefragt. Die Strategie ist ein Interface, das die Methode isCooperative zur Verfügung stellt. Ein solcher Aufruf gibt zurück, ob Agent p1 bei einem Spiel gegen Agent p2 bei Benutzung dieser Strategie bei dem gegebenen bisherigen Verlauf des aktuellen Adaptionsschrittes kooperieren würde.   
Nachdem auf diese Weise die Kooperationsentscheidungen beider Agenten bestimmt wurden, fragt die Engine bei der Konfiguration das Spiel ab und lässt die Agenten mit den gegebenen Entscheidungen spielen. Das Ergebnis wird als Instanz der GameResult Klasse zurückgegeben. Diese speichert die beiden Agenten, deren Kooperationsentscheidungen und die erhaltenen Auszahlungen. Dieses Ergebnis wird dann noch der SimulationHistory hinzugefügt.  
Nachdem alle Runden abgeschlossen sind werden die Agenten nach ihrem Erfolg in diesen Runden sortiert und ihre Strategien gemäß des gegebenen Adaptionsmechanismus angepasst. Zuletzt wird noch geprüft, ob ein Gleichgewicht erreicht wurde.  
Sind alle Adaptionsschritte abgeschlossen, also weil ein Gleichgewicht oder die Maximalzahl erreicht wurde, werden die Ergebnisse der Wiederholung in einer Instanz der Klasse IterationResult gespeichert. Sie beinhaltet die Agenten sortiert nach finalem Rang, den Verlauf des letzten Adaptionsschrittes, ob sich ein Gleichgewicht eingestellt hat und falls ja dessen Effizienz sowie die Zahl durchgeführter Adaptionsschritte. Dieses Ergebnis wird dann zurückgegeben.  
  
Die Klassen und Interfaces die hier vorgekommen sind Teil des Unterpakets SimulationEngine vom Model. Zu den Interfaces PairBuilder, etc. gibt es natürlich die entsprechenden Implementierungen, die bereits im Pflichtenheft beschrieben wurden.

Um den Simulationsablauf nun aber möglichst frei konfigurierbar zu machen, wollen wir es dem Nutzer ermöglichen, zusätzlich eigene Implementierungen der Interfaces zu schreiben und per Plugin zu integrieren, also zb einen eigenen Paarbildungsalgorithmus 🡪 Folienwechsel  
Das erste Problem, auf das wir dann gestoßen sind, ist dass die per Plugin hinzugefügten Algorithmen und Mechanismen auch parametrisierbar sein sollen und der Nutzer diese Parameter im Konfigurationsfenster festlegen können soll. Dazu reicht es nun eben nicht, die Implementierung des Interfaces direkt als Plugin zu verwenden, da dann nur eine konkret parametrisierte Instanz ins Programm geladen wird und es dann schwierig ist, zur Laufzeit andere Parametrisierungen zu erzeugen. Unsere Lösung ist es, als Plugin eine Wrapper-Klasse zu verwenden, die auf Anfrage konkrete Parametrisierungen zur Verfügung stellt. Das Plugin-System, das diese Idee realisiert, befindet sich im plugin-Paket innerhalb im model. Das Plugin ist die besagte Wrapper klasse, die auf Anfrage eine Liste aller nötigen Parameter zurückgibt und parametrisierte Instanzen erzeugt.  
Die nächste Frage die wir uns gestellt haben 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. Das ist eine abstrakte Fabrik, die wiederum 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 enthält. Auf Anfrage gibt das PluginControl objekt die eingestellten Werte der Parameter zurück, die dann verwendet 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.  
 in einem zentralen Repository abgespeichert.

* Repository

Dieses ist als Singleton implementiert und speichert neben den Plugins 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.

8:17

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