AiSystem 2.0

Coding Guidelines

1) Wo möglich, sollte immer das Objekt mit den Daten dazu, die Aufgabe erledigen.

Beispiel: Statt das Game2048 und die Informationen dazu separat zu behandeln, existiert Informed2048, da das Spiel selbst am besten wissen sollte, welche Informationen dazu auftreten.

2) Es kann immer nur ein Objekt der Besitzer eines anderen Objektes sein.

Unzulässig wäre demnach:

```
val x = X()
val y = Y()
val z = Z(x)
```

In diesem Beispiel müsste man ein Objekt YZ kreieren, dass den Teilen x oder Teile von x zuspielt. X muss nicht unbedingt als Kopie verteilt werden, auch wenn dies sicherer wäre, da dies ineffizienter wäre.

3) Abstrakte Klassen sind verboten und stattdessen sollen Interfaces benutzt werden. Klassen sind stets final.
4) Es soll möglichst deklarativ vorgegangen werden. Statt „get“, „calc“ oder „play“ soll man eher eine Benennung verwenden, die das Resultat beschreibt und nicht, wie man dorthin kommt. Das hat den Hintergrund, dass es der Klasse überlassen werden soll, wie sie zum Resultat kommt und dies nicht durch die Benennung festlegen oder suggerieren soll.
5) Es wird wie folgt kommentiert:

Klassen:

```
/**
 * - mutable <reason>
 * - thread safe
 *
 * <description>
 * @param
 * @throws
 */
```

Methoden:

```
/**
 * - mutable <reason>
 * - thread safe
 *
 * <description>
 * @param
 * @return
 * @throws
 */
```

- Sofern eine Methode oder Klasse destruktiv agiert, wird die Begründung hinter „- mutable“ geschrieben.
- Wenn eine Methode oder Klasse Thread safe ist, so wird dies mit „- thread safe“ signalisiert, sofern diese Methode/Klasse auch mutable ist. Immutable Klassen/Methoden benötigen dies nicht, da sie ohnehin Thread safe sind.
- Es folgt eine Beschreibung dessen, was die Methode macht/wofür die Klasse gedacht ist.

- Es werden die Parameter aufgezählt und welchem Zweck sie dienen. Falls Invarianten vorliegen, werden sie hier erwähnt, sofern sie nicht im „@throws“ auftauchen
- Es wird gesagt, was zurückgegeben wird.
- Es wird angegeben, welche Fehler bei welchen Invarianten geworfen werden.

Sollte eine Eigenschaft nichtzutreffend sein (immutable, oder keine Parameter), so wird der Punkt weggelassen. Außerdem wird nicht auf Zwang kommentiert. Es soll das erwähnt werden, dass nicht eindeutig ist.

6) Klassen und Methoden sollen klein gehalten werden. Als Faustregel bietet sich folgendes an:

max 4 Parameter je Methode, 10 Zeilen je Methode, 5 (mutable) Variablen je Klasse, 5 (public) Methode je Klasse

Um Klassen zusätzliche Eigenschaften zu verleihen, bietet sich Komposition an. Dies ist an vielen Stellen im Projekt erkennbar (Game -> GameSequence -> TimedSequence). Um Punkt 3 nicht zu verletzen, soll das neue Objekt eine Instanz des alten Objektes beinhalten und sein Interface erfüllen, statt davon zu erben.

7) Klassen sollen stets nur von Interfaces abhängig sein und nicht von konkreten Klassen.

Struktur

Das System besteht aus den folgenden drei Schichten:

1. Base: Enthält das Anwendungsfeld, an das sich der Agent anpassen soll. Das kann z.B. ein Spiel wie Schach sein. Da es denkbar ist, dass ein Anwendungsfeld von Dritten übernommen werden kann und eventuell nicht losgelöst von seiner UI ist, kann eine Aufteilung davon in Logic und UI nicht immer gewährleistet werden. Deshalb gilt dies als separate Schicht.
2. Logic: Das Zentrum hiervon bilden die Agenten, die von anderen Komponenten unterstützt werden. Zu finden sind hier auch die Wrapper, um die Komponenten aus Base universal benutzen zu können. Es sei angemerkt, dass die Logic für sich eine textliche Repräsentation besitzt, sodass man dies einfach ohne eine eigens programmierte UI in der Konsole ausgeben kann.
3. UI: Zur Visualisierung wird hier derzeitig JavaFX benutzt und das nicht weil JavaFX besonders gut ist, sondern vielmehr aufgrund mangelhafter Alternativen.

Bei den Schichten ist es wichtig, dass diese von ihrer darüberliegenden Schicht komplett losgelöst sind, somit komplett unterschiedlichen Entwurfskonzepten zugrunde liegen können und austauschbar sind.

Logic

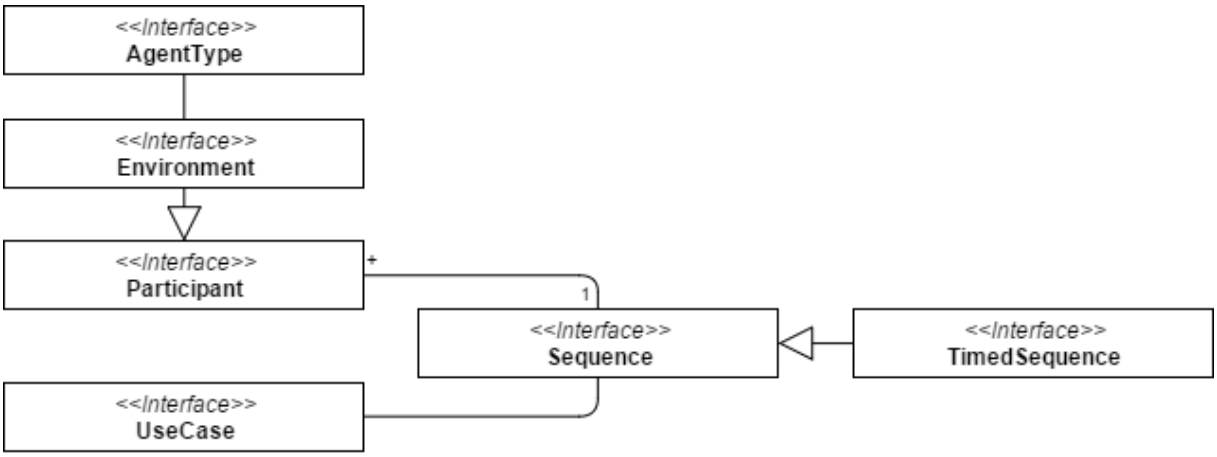

Abbildung 1: Interfaces für Hauptfunktionalitäten

UseCase: In Abbildung 1 wird es zwar als Interface angezeigt, kann aber auch eine konkrete Klasse sein. Für die Logic Schicht ist es unerheblich, wie der UseCase letztendlich realisiert wurde.

Sequence: Dient als einheitliche Schnittstelle, die den UseCase mit dem Environment vereint. Während es bei beispielsweise Spielen häufig der Fall ist, dass „auf dem Spiel“ gespielt wird, man somit selbst kontrollieren muss, dass der richtige Spieler wieder dran ist, wird hierbei die Kontrolle dem Spiel selbst in Form von der Sequence realisiert. Sequence nimmt somit eine bestimmte Anzahl an Spielern (Participant) an und kontrolliert dann selbst den Ablauf. Von außen kann man lediglich sagen, wie viele Züge vergangen sind und sich das Resultat geben lassen, wobei dies eher zu Testzwecken gedacht ist.

```
Sequence2048(
    Game2048(), Environment2048()
).step(1) // one step was done
```

Environment: Dient als Bindeglied zwischen Anwendungsfall und Agent. Im Ablauf würde Environment als Spieler in der Sequence auftreten und von dieser die Spielrelevanten Informationen beziehen. Diese würde das Environment dann für den Agenten umwandeln. Damit ist das Environment sowohl vom Anwendungsfall, als auch vom Agenten Typ (AgentType) abhängig.

AgentType: Hier steckt die künstliche Intelligenz. Es sind dabei verschiedene Typen von Agenten denkbar, wie z.B. Reinforcement Learning Algorithmen oder welche, die Spiele simulieren können müssen. Daraus ergeben sich unterschiedliche Anforderungen an das Environment, das diese alle enthalten müsste. In diesem Fall ist es sogar denkbar ein konkretes Environment für einen einzigen konkreten Agenten zu erstellen.

Ein Agent kann von anderen Agenten abhängig sein, bzw. diese enthalten, weshalb er den Zug nicht ausführt, sondern lediglich sagt, welchen er durchführen würde, sodass dies von einem anderen Agenten weiterverarbeitet werden kann.

TimedSequence: Bietet die Möglichkeit die Sequence in der Anzahl und Dauer anzupassen. So lässt sich sagen, die Sequence soll 1.000 Spiele Durchlaufen und dabei je Zug mindestens 0.3 Sekunden brauchen, sodass man den Ablauf besser verfolgen kann.

Information

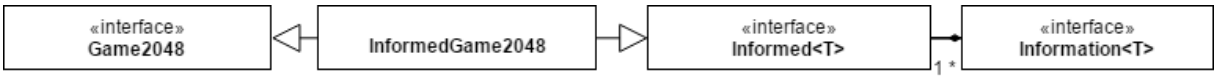

Abbildung 2: Decorator für Information

Die Anwendung bietet die Möglichkeit Information bzgl. der Komponenten anzuzeigen. Für diesen Fall gibt es Decorator, die noch zusätzlich das „Informed“ Interface implementieren. Das Spiel Game2048 hätte demzufolge ein Interface (falls es in Base kein solches gäben sollte, kann man dies in der Logic Ebene erstellen und einen Adapter benutzen), dass der Decorator implementiert und zusätzlich Informed. Dadurch erhält man als Anwender die Möglichkeit der konkreten Klasse Informed eine Reihe an konkreten Informationen zu geben. Das könnte wie folgt aussehen:

```
Sequence2048(
        InformedGame2048(
                Game2048Binary(),
                HighestTile(),
                AvgTile(),
                MovesTaken(),
                AvgScore()
        ),
        Environment2048(
                Agent()
        )
)
```

Eine Information arbeitet aktiv. Das bedeutet eine Klasse die Informed ist, unterscheidet sich insofern von der normalen Klasse, als dass sie zur normalen Funktionalität Aufrufe an die Informations tätigt. Diese werden ihrerseits ihre Information bestimmen und wollte man nun ohne direkte UI das Ergebnis haben, so könnte man einfach toString bei Sequence2048 aufrufen, was wiederrum auf die eigenen Teile verweist und InformedGame2048 würde darin die Informationen vereinen.

```
enum class Event {
    INITIAL,
    PRE_RUN,

    POST_GAME,

    PRE_MOVE,
    POST_MOVE,

    POST_INFO
}
```

Das Event Enum zeigt auf, welche verschiedenen Phasen es gibt, zu dem die Informations aufgerufen werden und welches Event sie bekommen. Dies geschieht nur aufgrund von Effizienz.

Glossar