Reflexion, Plan vs Realität

Gegenüberstellung Klassendiagram und Implementierung:

Überraschender Weise ist vieles ident bei der Modellierung als auch der tatsächlichen Implementierung, es sollte zwar gegeben sein, persönlich bin ich allerdings davon ausgegangen das eventuell viel geändert werden müsse. Nach einem größtenteils überraschend positiven Abgabegespräch für Teil 1, habe ich versucht mich dem Modell so gut es ging zu folgen.

Karten-package Klassen sind komplett ident zum Modell, allerdings mussten außer bei den Enumeration Klassen überall zuzügliche Variablen bzw Konstanten hinzugefügt werden um die gewünschte Funktionalität in dem Bereich oder auch außerhalb zu verbessern.

Als nächstes wurde das MVC Pattern und die wichtigeren Klassen für den Spielbetrieb: Netzwerk, GameLogic, GameStatus(Name wurde nachträglich geändert um Verwirrungen mit Serverobjekten zu vermeiden) und die CLI-Ausgabeklassen.

In der Netzwerkklasse ist lediglich ein Wegfallen der ConDat-Klasse passiert, die anderen Methoden wurden mehr oder weniger wie im Modell implementiert. GameLogic und GameStatus wurden um einiges aufwendiger implementiert als im Klassendiagram angezeigt um die nötige Funktionalität zu schaffen. Des Weiteren wurde eine Converter Klasse ebenfalls in das ControllerPackage hinzugefügt um Server objekte von der Netzwerkklasse für den lokalen Client verwendbar zu machen vice versa. Das Implementieren des MVC Patterns stellte sich anfänglich als schwieriger heraus als gedachte aber am Ende wurde die CLI Klassen auch mehr oder weniger wie im Modell implementiert.

Im AI Package in welchen die Dijkstra Pfadfindung und das GoalSetting implementiert sind schaut in der Realität doch deutlich anders aus. Außer das der Umfang beider Klassen ein wenig mehr umfasst existiert auch eine Konvertierungsklasse in dem Packet, welche es dem Programm erlaubt, ermittelte Pfade des Pathfinders in eine Liste an EDirection enumerations umzuwandeln, um diese wiederverwertbar zwischenzuspeichern zu können.

Y-Statements:

- Im Kontext von der Art und Weise wie die Spielkarten technisch implementiert werden habe ich mich angesichts von Phase/Interface-Pattern für zwei semantisch getrennte Kartenarten welche von ein gemeinsames Interface implementieren entschieden und eine einzelne Kartenklasse vernachlässigt, um Kohärenz und Single Responsibility zu erreichen, unter Inkaufnahme von einem anfänglich höheren Auwand bei der Implementierung,weil die besagten Vorteile erstens überwiegten und es den Code zusätzlich für die einzelnen (Pre-) Spielphasen erleichterte.
- Im Kontext von Skalierbarkeit und Single Responsibility habe ich mich angesichts vieler Überlegungen für das Interface Pattern für die Pathfinderklasse(Dijkstra) entschieden und ein direktes verwenden der individuellen Pathfinderklasse vernachlässigt, um einerseits die Kopplung als auch die Verbesserung der Erweiterbarkeit in meinem Code zu erreichen, unter Inkaufnahme von dem das die Dijkstraklasse zur Zeit noch alleine das Interface implementiert, weil zuvor noch eine (fürs Testen) Random-Pathfinderklasse auch implementiert wurde, im Zeichen von YAGNI diese wieder verworfen wurde, da diese fürs Endprodukt nicht mehr relevant wurde.
- Im Kontext von der Architektur habe ich mich angesichts von der Synchronisierung der arbeitenden Klassen für die Implementierung des MVC Patterns entschieden und nicht das Äquivalent des Observer Patterns vernachlässigt, um die besagten Klassen bei Änderungen von Zuständen weitläufig zu zusammenspielen zu lassen, unter Inkaufnahme von erhöhtem Implementieraufwand, da eine großzahl an weiteren Klassen, Packages dazukommen musste und man sich bzw. weitere Implementierungen zuerst Gedanken machen, bevor man diese in der richtigen Komponente hinzufügte.