Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Glossar

DI *dependency injection*. 8

DTO *data transfer object*. 3, 4

VO *value object*. 4

Teil I

Beschreibung des Projekts

Der vorliegende Programmentwurf beschäftigt sich mit der Visualisierung von Wegfinde-Algorithmen. Die Verfahren werden auf einem Gitter durchgeführt und verfolgen hierbei das Ziel, den kürzesten Weg zwischen zwei Punkten `S` und `Z` zu finden. Die grundlegende Idee dieser Darstellungsmethode ist in Abbildung 1 zu sehen. Die gelb markieren Koordinaten zeigen den kürzesten Weg und die grau eingefärbten Felder stellen Hindernisse dar.

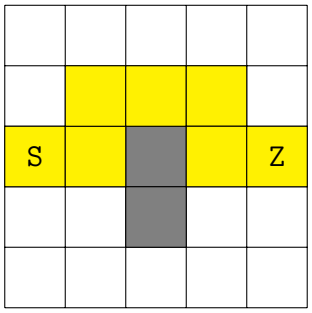

Abbildung 1: Wegfinde-Algorithmus

Es wird unterschieden zwischen gewichteten und ungewichteten Algorithmen. Ein gewichtetes Verfahren kann während der Wegsuche zusätzliche Streckenkosten beachten (z. B. bei einem Stau) und somit nicht nur den kürzesten, sondern auch den günstigsten Weg finden. Kosten / Gewichte können auf dem Gitter durch Zahlen dargestellt werden, wie in Abbildung 2 zu sehen ist.

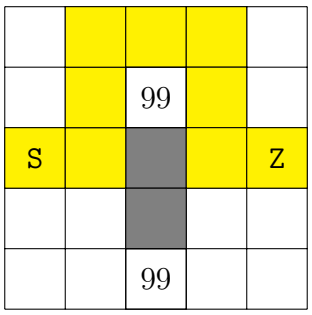

Abbildung 2: Wegfinde-Algorithmus mit Gewicht

Das Projekt besteht aus zwei Teilen: Der API (ASP.NET Core Web API, im Ordner „`api`“) und der Benutzeroberfläche (Vue, im Ordner „`vue`“). Der Quelltext und eine kurze Anleitung zur Projektdurchführung sind über den folgenden Link auf GitHub zu finden.

`https://github.com/JensDll/pathfinding-visualization`

Teil II

Clean Architecture

Die Ordnerstruktur des API Projekts ist orientiert an den Schichten der *Clean Architecture* mit Abhängigkeiten von außen nach innen. Die konkret umgesetzten Schichten sind in Abbildung 3 zu sehen.

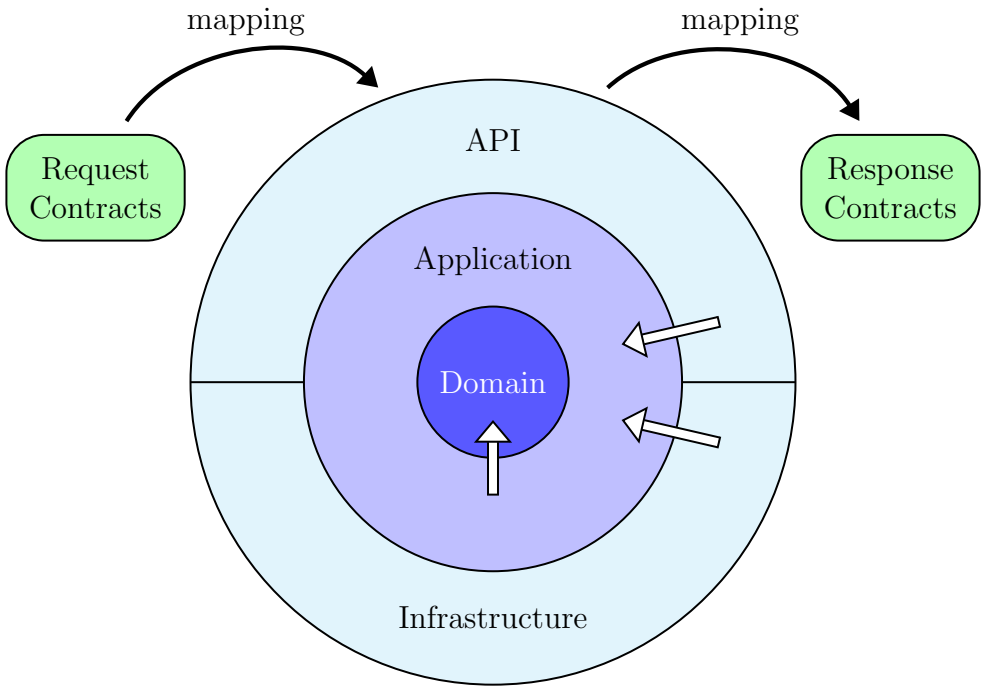

Abbildung 3: Clean Architecture

In der Anwendungsebene werden hauptsächlich Interfaces definiert, welche durch die Infrastrukturebene implementiert werden. Das API Projekt sollte ausschließlich diese Interfaces verwenden und nie direkt ein Objekt der Infrastruktur instanziieren. Um diese

Einschränkung durchzusetzten, hilft es, die Klassen dieser Ebene mit dem Schlüsselwort `internal` zu markieren. Die Domäne enthält anwendungsübergreifende Bausteine, wie Datenstrukturen, Entitäten, Enums und die Implementierung der Wegfinde-Algorithmen. Das Domänen Projekt ist unabhängig von den anderen Schichten. Es gibt außerdem ein fünftes Projekt mit dem Namen `Contracts`. Hier werden alle Verträge beschrieben, die ein Anwender mit der Schnittstelle haben kann. Verträge sind Datentransferobjekte (engl. *data transfer object* (DTO)) und werden mit dem Suffix `Dto` gekennzeichnet. Datentransferobjekte tauchen nur unmittelbar im Bereich der Schnittstelle auf und sollten nie direkt in anderen Teilen der Anwendung verwendet werden. Durch Mapping werden DTOs in Objekte der Domäne umgewandelt und umgekehrt:

$$\text{DTO} \Longrightarrow \textit{domain object} \Longrightarrow \text{DTO} \tag{1}$$

Diese Abbildung sollte immer geschehen, auch wenn sich die Objekte sehr ähnlich sehen. Verträge und Domäne können sich so unabhängig voneinander weiterentwickeln, ohne im Programmcode aufwändige Änderungen vornehmen zu müssen. Im besten Fall muss nur der Mapping Code angepasst werden. Dieser befindet sich auf der Infrastrukturebene.

Teil III

Domain Driven Design

Die Domänensprache des Projekts umfasst die folgenden Begriffe:

- **Grid** ∼ Das Gitter auf dem der kürzeste Weg gesucht und angezeigt wird.
- **GridNode** ∼ Ein Knoten (engl. *node*), der eine Position auf dem Gitter beschreibt. Knoten haben neben primitiven Werten wie Gewicht, außerdem die folgenden Eigenschaften:
  - **GridNodeType** ∼ Der Typ des Knotens mit Werten wie Start, Ziel, Wand oder Standard.
  - **Position** ∼ Die Koordinate des Knotens in der Form $(Zeile, Spalte)$.

Ein Gitter besteht aus einer zweidimensionalen Anordnung von Knoten. Die meisten Be-

griffe wie **Grid** und **GridNode** werden im Programmcode als Entitäten bezeichnet. **Position** hingegen ist ein *value object* (VO). Es macht mehr Sinn Punkte anhand ihrer Werte zu unterscheiden und nicht anhand des gleichen Verweises. VOs können in C# mit dem in Version 9.0 neu eingeführten Datensatztyp (engl. *record type*) sehr bequem definiert werden. Methoden um die Wertgleichheit sicherzustellen, werden durch den Compiler automatisch erzeugt. Eine gekürzte Definition des **Position** VO ist in Quelltext 1 zu sehen.

Quelltext 1: Der Datensatztyp einer Koordinate

```
public record Position(int Row, int Col);
```

Einige Datentransferobjekte werden ebenfalls als *record type* definiert. Eigenschaften von DTOs sollten immer schreibgeschützt, dies ist vor allem deshalb sinnvoll, da Verträge unveränderlich sein sollten und nur durch Mapping in eine ggf. veränderliche Datenstruktur gebracht werden. Wegfinde-Algorithmen sind Gegenstand des *abstraction code* auf der Domänenebene. Sie werden als Teil eines *service* zusammengefasst mit dem Namen `PathfindingService` und über diesen aufgerufen. Da für das Projekt nicht wirklich eine persistente Datenspeicherung nötig ist, werden keine Repositories verwendet.

Teil IV

Programming Principles

Es wurde versucht, während der Entwicklung verschiedene *programming principles* einzuhalten. Im Folgenden werden die aus der Vorlesung behandelten Muster untersucht und beschrieben, wie sie im Projekt umgesetzt wurden.

1 SOLID

1.1 Single Responsibility Principle (S)

Es wird hier als Beispiel die `PathfindingService` Klasse beschrieben, welche in Quelltext 2 zu sehen ist.

Quelltext 2: `PathfindingService` Klasse

```
internal class PathfindingService : IPathfindingService
{
   public PathfindingResult BreadthFirstSearch(Grid grid)
   {
      return new BreadthFirstSearch(grid).ShortestPath();
   }

   public PathfindingResult Dijkstra(Grid grid)
   {
      return new Dijkstra(grid).ShortestPath();
   }

   public PathfindingResult AStar(Grid grid)
   {
      return new AStar(grid).ShortestPath();
   }
}
```

Die Klasse muss nur genau dann geändert werden, wenn ein neuer Algorithmus hinzugefügt wird.

1.2 Open / Closed Principle (O)

Jede Wegfinde-Klasse besitzt eine Methode `ShortestPath` und muss das folgende Interface implementieren.

Quelltext 3: Wegfinde-Algorithmus Interface

```
public interface IPathfindingAlgorithm
{
   PathfindingResult ShortestPath();
}
```

Eine weitere Klasse, die Wegfinde-Funktionalität benötigt, ist dadurch nicht abhängig von einer bestimmten Implementierung. Die folgenden zwei Zeilen sind äquivalent.

Quelltext 4: Abstraktion der Wegfinde-Algorithmen

```
IPathfindingAlgorithm bfs = new BreadthFirstSearch();
IPathfindingAlgorithm dijkstra = new Dijkstra();
```

Neue Verfahren können hinzugefügt werden (*open*), ohne das die Klassen, die Wegfinde-Funktionalität verwenden, angepasst werden müssen (*closed*).

1.3 Liskov Substitution Principle (L)

Die umgesetzten Wegfinde-Algorithmen benötigen eine Funktion, welche von einem gegebenen Knoten aus die nächsten zu besuchenden Knoten auswählt. Die Methode wird `GetNeighbors` genannt und ist definiert in der abstrakten Klasse `Grid`, wie im folgenden Programmausschnitt zu sehen ist.

Quelltext 5: **Grid** Entität

```
public abstract class Grid
{
   public abstract List<GridNode> GetNeighbors(GridNode node);
}
```

Subklassen, die von `Grid` ableiten, müssen diese Methode überschreiben. Für den Algorithmus spielt es keine Rolle, wie die konkrete Umsetzung aussieht. Zum Beispiel könnte

eine Implementierung die horizontal liegenden Nachbarn liefern, eine andere die diagonal liegenden und eine weitere beide. Die folgenden zwei Zeilen sind äquivalent.

Quelltext 6: Abstraktion des Gitters

```
Grid grid = new HorizontalGrid();
Grid grid = new DiagonalGrid();
```

1.4 Interface Segregation Principle (I)

Es können hier die beiden Interfaces gennant werden, welche das in Gleichung 1 gezeigte Mapping beschreiben.

Quelltext 7: Request Mapper

```
public interface IPathfindingRequestMapper
{
    Grid MapPathfindingRequestDto(PathfindingRequestDto pathfindingRequestDto);
}
```

Quelltext 8: Response Mapper

```
public interface IPathfindingResponseMapper
{
    PathfindingResponseDto MapPathfindingResult(PathfindingResult pathfindingResult);
}
```

Anstatt ein großes Allzweck-Interface (wie z. B. `IPathfindingMapper`) umzusetzen, werden zwei kleinere Interfaces verwendet, getrennt nach Anfrage und Antwort.

1.5 Dependency Inversion Principle (D)

Gelöst durch die in Teil II beschriebene Architektur. API (*high level*) und Infrastrukturebene (*low level*) sind beide abhängig von den Abstraktionen auf der Anwendungsebene.

2 GRASP

2.1 Information Expert

Für eine neue Aufgabe ist derjenige zuständig, der schon das meiste Wissen für die Aufgabe mitbringt. Beispiel: Manhattan Distanz zwischen zwei Punkten bestimmen. Diese Funktion ist Teil der **Position** Entität, da hier schon die Hälfte des benötigten Wissens vorhanden ist.

Quelltext 9: Manhattan Distanz

```
public record Position(int Row, int Col)
{
   public int ManhattanDistance(Position p)
   {
      var v = (this - p).Absolute;
      return v.Row + v.Col;
   }
}
```

2.2 Geringe Kopplung

Die Funktionsweise von Klassen sollte durch Interfaces beschrieben werden. Klassen haben keine direkten Abhängigkeiten, sondern werden durch *dependency injection* (DI) im Konstruktor zur Verfügung gestellt. Im Fall von einer Änderung kann so ein Austausch der Implementierung mit minimalem Aufwand (eine Zeile) erfolgen. Eine geringe Kopplung wird nach diesem Prinzip erreicht.

2.3 Hohe Kohäsion

Es werden für dieses Prinzip die Klassen der Wegfinde-Algorithmen genannt und hier am Beispiel von `BreadthFirstSearch` erklärt.

Quelltext 10: `BreadthFirstSearch` Algorithmus

```
public class BreadthFirstSearch : IPathfindingAlgorithm
{
   private readonly Grid _grid;

   public BreadthFirstSearch(Grid grid)
   {
      _grid = grid;
   }

   public PathfindingResult ShortestPath()
   {
      // ...
   }
}
```

Die Instanzvariable von `BreadthFirstSearch` wird in allen (in diesem Fall einer) Methode verwendet.

Teil V

Refactoring

Es wurden im Laufe des Projekts verschiedene Refactorings durchgeführt. Viele waren eher kleinerer Natur, doch einige haben auch größere Änderungen mit sich gebracht. In den folgenden Abschnitten wird jeweils ein Problem beschrieben und gezeigt, wie es durch eine Anpassung der Programmstruktur gelöst werden kann.

1 Lange Parameterliste

Um einen Wegfinde-Algorithmus zu verwenden, wurden zuvor die drei Parameter `grid`, `startPosition` und `searchDiagonal` benötigt. Manche Verfahren verwendet zusätzlich außerdem noch die Zielposition. Der folgende Quelltext zeigt einen Ausschnitt aus alten der `PathfindingService` Klasse (vgl. Quelltext 2 für die neue Implementierung).

Quelltext 11: Die alte `PathfindingService` Klasse

```
internal class PathfindingService : IPathfindingService
{
   public PathfindingResult Dijkstra(GridNode[][] grid,
      Position startPosition, bool searchDiagonal)
   {
      return new Dijkstra(GetSearchType(searchDiagonal))
         .ShortestPath(grid, startPosition);
   }

   private static IGetNeighbors GetSearchType(bool searchDiagonal) =>
      searchDiagonal
         ? new GetNeighborsDiagonal()
         : new GetNeighborsHorizontal();
}
```

Um die Parameterliste zu verkürzen und die Wegfinde-Algorithmen einheitlich zu halten, wurden die verschiedenen Eigenschaften in der Klasse `Grid` zusammengefasst. Die Polymorphie der `GetNeighbors` Methode ist jetzt nicht mehr über das Interface `IGetNeighbors` gegeben, sondern über die Subklassen der abstrakte Klasse `Grid` (vgl. Liskov Substitution Principle 1.3). Die Änderungen wurden in Pull Request #148 durchgeführt.

2 Divergierende Änderungen

Besonders zu Beginn der Entwicklung gab es noch einige Klassen im Projekt mit mehr als einer oder unklarer Aufgabe. Ein Beispiel ist die `AlgorithmService` Klasse, welche im folgenden Quelltext zu sehen ist.

Quelltext 12: `AlgorithmService` Klasse

```
public class AlgorithmService : IAlgorithmService
{
   public List<GridNode> GetNeighbors(GridNode[][] grid,
      (int row, int col) point)
   {
      // ...
   }

   public List<GridNode> GetNeighborsDiagonal(GridNode[][] grid,
      (int row, int col) point)
   {
      // ...
   }

   public void ConstructShortestPath(GridNode node, List<GridNode> shortestPath)
   {
      // ...
   }
}
```

Die Klasse besitzt drei Methoden, wobei diese nicht unbedingt etwas miteinander zu tun haben und wahrscheinlich besser aufgehoben wären in einer eigenen Entität mit verwandter Bedeutung. Die Methoden, um benachbarte Knoten zu bestimmen, wurden wie in 1.3 bereits beschrieben, in die Klasse **Grid** ausgelagert. Da für die Berechnung des kürzesten Wegs immer ein Ausgangsknoten benötigt wird, wurde diese Funktion als Teil der **GridNode** Entität aufgenommen (vgl. Information Expert 2.1). Die zu der Klasse hinzugefügten Methoden sind in Quelltext 13 zu sehen.

Quelltext 13: **GridNode** Entität

```
public class GridNode
{
   // Eigenschaften der Klasse ...

   public List<GridNode> ConstructShortestPath()
   {
      var shortestPath = new List<GridNode>();
      ConstructShortestPathImpl(this, shortestPath);
      return shortestPath;
   }

   private void ConstructShortestPathImpl(GridNode node, List<GridNode> shortestPath)
   {
      if (node == null) return;
      ConstructShortestPathImpl(node.PreviousGridNode, shortestPath);
      shortestPath.Add(node);
   }
}
```

Die Änderungen wurden in Pull Request #103 und #148 durchgeführt.

Teil VI

Entwurfsmuster

Wegen der zur Zeit eher geringen Komplexität des Projektes hat es sich nicht ergeben, dass eine Umsetzung von Entwurfsmustern sinnvoll erscheint. Es ist unklar, ob die erzwungene Verwendung eines Musters die Lesbarkeit des aktuellen Programmcodes verbessern würde. Auf Seiten der Unittests kann aber dennoch ein Beispiel gegeben werden. Es wird hier eine Form des Erbauermusters verwendet, um komplexe Objekte für den Einsatz in Testmethoden zu erzeugen. Konkret handelt es sich um die `GridFactory` Klasse, welche anhand einer Zeichenkettenmatrix `Grid` Objekte erzeugt. Die statische Klasse besitzt eine Methode `Produce` und nimmt entgegen ein `string[]` sowie den Typ des zu erstellenden Gitters. Die gekürzte Version der Klasse ist in Quelltext 14 zu sehen.

Quelltext 14: `GridFactory` Klasse

```
public static class GridFactory
{
   public static Grid Produce(string[] stringGrid, GridType gridType)
   {
      // ...
      return grid;
   }
}
```

Um beispielsweise das Gitter aus Abbildung 2 zu erzeugen, wird die Funktion mit den folgenden Parametern aufgerufen.

Quelltext 15: Erzeugen eines Gitters durch die `GridFactory` Klasse

```
Grid grid = GridFactory.Produce(new string[]
{
   "1 1 1 1 1",
   "1 1 99 1 1",
   "S 1 W 1 F",
   "1 1 W 1 1",
   "1 1 1 1 1"
}, GridType.Horizontal)
```