

# Inhaltsverzeichnis

<b>I</b>	<b>Beschreibung des Projekts</b>	<b>1</b>
<b>II</b>	<b>Clean Architecture</b>	<b>2</b>
<b>III</b>	<b>Domain Driven Design</b>	<b>3</b>
<b>IV</b>	<b>Programming Principles</b>	<b>4</b>
<b>V</b>	<b>Refactoring</b>	<b>6</b>

# Abbildungsverzeichnis

1	Wegfinde-Algorithmus . . . . .	1
2	Wegfinde-Algorithmus mit Gewicht . . . . .	1
3	Clean Architecture . . . . .	2

# Quelltextverzeichnis

1	Der Datensatztyp einer Koordinate . . . . .	4
2	Die <b>Grid</b> Entität-Klasse . . . . .	5
3	Wegfinde-Algorithmus Interface . . . . .	5
4	Abstraktion der Wegfinde-Algorithmen . . . . .	6

# Glossar

- DI** *dependency injection.* 4
- DTO** *data transfer object.* 3
- 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.

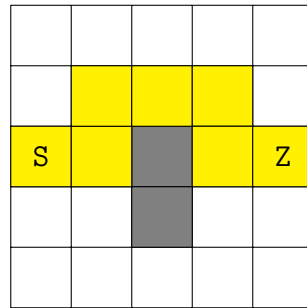


Abbildung 1: Wegfinde-Algorithmus

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.

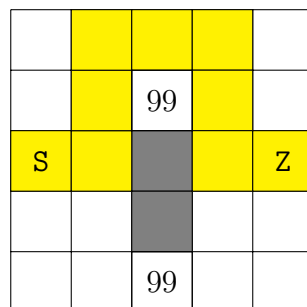


Abbildung 2: Wegfinde-Algorithmus mit Gewicht

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.

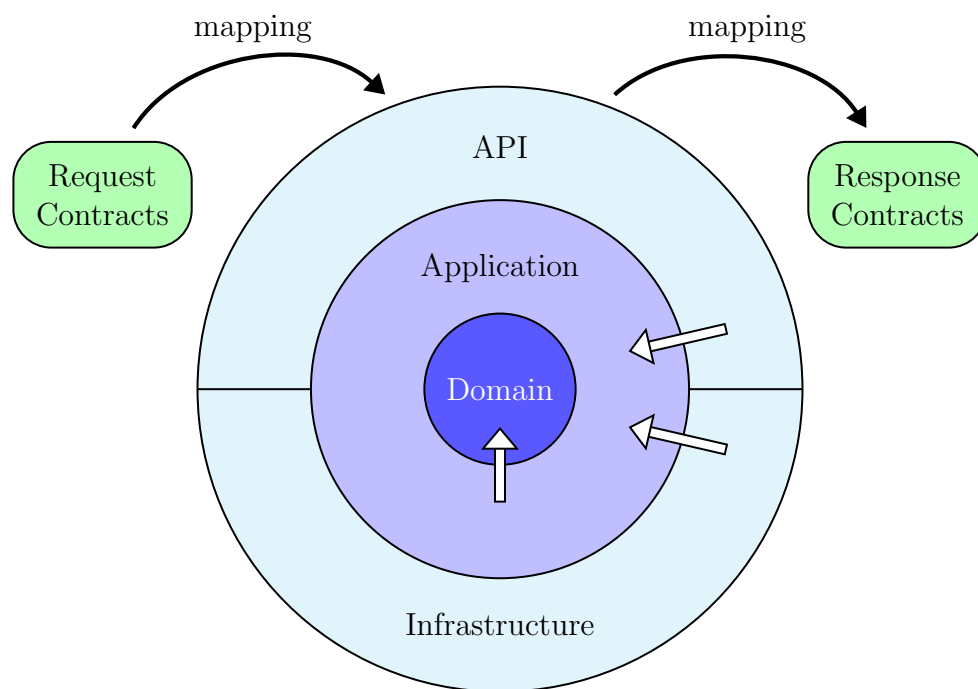


Abbildung 3: Clean Architecture

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 instanziiieren. Um diese

Einschränkung durchzusetzen, 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:

$$\text{DTO} \implies \text{domain object} \implies \text{DTO}$$

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** ~ Knoten (engl. *node*), welcher eine Position auf dem Gitter beschreibt. Ein Gitter besteht aus mehreren Knoten. 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*).

Die meisten Begriffe 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);
```

Datentransferobjekte werden ebenfalls als *record type* definiert. 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. Vor allem das Erreichen von geringer Kopplung ist mit ASP.NET Core eine leichte Aufgabe. Die Funktionsweise von Klassen sollte durch Interfaces beschrieben werden. Durch den bereits standardmäßig vorhandenen *dependency injection container* (DI-Container) können diese Interfaces der gesamten Anwendung zur Verfügung gestellt werden. 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. SOLID und DRY Prinzipien sollten auf Seiten des API Projekts ebenfalls trivial umgesetzt sein. Das Verwenden von DI hilft auch in diesen Bereichen Verstößen

vorzubeugen. Ein Beispiel kann gegeben werden für das **L** (Liskov Substitution Principle) in SOLID. Die umgesetzten Wegfinde-Algorithmen benötigen eine Funktion, um von einem Knoten aus die nächsten zu besuchenden Knoten auszuwählen. Die Methode wird **GetNeighbors** genannt und ist definiert in der abstrakten **Grid** Entität-Klasse, wie im folgenden Programmausschnitt zu sehen ist.

#### Quelltext 2: Die **Grid** Entität-Klasse

```
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. Der Algorithmus bleibt bei einem Austausch gleich. Ähnliches Verhalten kann beobachtet werden, wenn es um die Wegfinde-Verfahren selbst geht. 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 beiden letzten Zeilen in [Quelltext 4](#) sind äquivalent.

#### Quelltext 4: Abstraktion der Wegfinde-Algorithmen

```
Grid grid = new Subklasse();  
IPathfindingAlgorithm bfs = new BreadthFirstSearch(grid);  
IPathfindingAlgorithm dijkstra = new Dijkstra(grid);
```

## Teil V

# Refactoring