Dokumentation der Praktischen Arbeit

Zur Prüfung zum mathematisch-technischen Softwareentwickler

Markus Faßbender
Prüflingsnummer: 142-59218
16.05.2014

Inhalt

1 Aufgabenanalyse

1.1 Analyse

Auf einem rechteckigen Feld soll ein Roboter den Boden versiegeln. Damit die frische Versiegelung nicht sofort wieder zerstört wird, muss eine unterbrechungsfreie Route über alle Parzellen (sofern möglich) gefunden werden, die jede Parzelle genau einmal abläuft. Dabei darf jeweils nur in die Richtungen oben, rechts, unten und links bewegt werden. In einem Zug „quer nach unten links“ zu fahren ist also nicht erlaubt. Die abzufahrenden Parzellen werden als Routenplan in einer Liste gespeichert, die anschließend abgearbeitet wird. Der Roboter darf auf der letzten Parzelle stehen bleiben.

Es gibt Hindernisse, die über mehrere benachbarte Parzellen gehen können und die der Roboter nicht überwinden kann. Diese können auch so angeordnet sein, dass der Roboter umzingelt ist oder diese wie eine Wand vor sich hat. Je nach Positionierung von Hindernissen kann ein Feld nicht vollständig abgelaufen werden. Dann soll eine Route gefunden werden, die möglichst viele Parzellen versiegelt.

Angegeben ist eine Uhrzeiger-Strategie, die in jedem Fall eine vollständige oder bestmögliche Route findet. Dabei wird auf jeder Parzelle neu entschieden, in welche Richtung fortgefahren wird. Dabei sind natürlich einige Richtungen nicht sinnvoll und/oder nicht möglich. Das Verfahren ist ein Backtracking-Algorithmus, der zur letzten Parzelle zurückspringt, wenn von einer Parzelle keine Richtung mehr möglich ist. Auf der vorherigen Parzelle wird versucht eine andere Richtung einzuschlagen usw. bis eine Lösung gefunden wurde oder keine vollständige Lösung vorhanden ist.

Ich soll nun einen zweiten Algorithmus entwickeln, der wie die Uhrzeiger-Strategie eine bestmögliche Lösung findet. Das heißt es müssen ebenfalls möglichst viele Parzellen – im Idealfall alle – versiegelt werden. Der Routenplan soll vom Algorithmus erstellt werden, sodass der Roboter diesen nur noch abfahren muss. Hier gilt es eine Strategie zu finden, die mindestens so gut ist wie die Uhrzeiger-Strategie.

1.2 Datenformat und Eingabe

Die Parzellen können als Feld in einem zweidimensionalen Array abgespeichert werden. Als Typ bietet sich *char* an, weil in das Feld einzelne ASCII-Zeichen eingetragen werden (z.B. ^, <, v, >, S, Z, H).

Alternativ könnte man auch einen eigenen Typen (eine Klasse oder ein *Enum*) verwenden. Das Array kann nach dem Einlesen der ersten Zeile alloziert werden, weil die Größe dann fest ist und sich nicht mehr ändert. Daher muss hier auch nicht unbedingt mit einer *ArrayList* o.ä. gearbeitet werden.

Der Routenplan hingegen sollte als Liste implementiert werden, weil anfangs nicht die Anzahl der nötigen Schritte bekannt ist. Eine Route besteht aus einer Menge von Parzellen, die in einer festen Reihenfolge versiegelt werden.

Die Daten werden als Datei eingelesen und zeilenweise ausgewertet. Kommentarzeilen können übersprungen werden. Als erstes wird die Anzahl der Felder ausgelesen, die für das Anlegen des Arrays benötigt wird. Danach kommt die Startposition und dann kommen in mehreren Zeilen die Hindernisse. Die Startposition und Hindernisse können dann direkt in das Datenobjekt aufgenommen werden.

1.3 Ausgabeformat

Das Ausgabeformat besagt, dass die Lösungen nach folgendem Muster ausgegeben werden sollen:

1. Die eingelesenen Anfangsbedingungen
2. Die Lösung mit der Uhrzeiger-Strategie und ihr Routenplan
3. Die Lösung mit der eigenen Strategie und ihr Routenplan
4. Statistik über den Zustand der Parzellen (versiegelt, nicht versiegelt, durch Hindernis besetzt)

Wichtig ist dabei, dass der ausgegebene Routenplan nur aus Parzellen besteht, an denen ein Richtungswechsel stattfindet. So wird die Route im Normalfall kürzer in der Ausgabe und man muss nicht bei 10x10 Parzellen einhundert Schritte ausgeben.

Als Beispiel ist in der Aufgabenstellung folgendes gegeben:

```
Startpunkt (1,1)
  1 2 3 4 5
1 S
2
3
4
5

Uhrzeiger-Strategy
  1 2 3 4 5
1 > > > > v
2 v < v < v
3 v ^ v ^ v
4 v ^ v ^ v
5 Z ^ < ^ <

Routenplan:
1,1 / 1,5 / 5,5 / 5,4 / 2,4 / 2,3 / 5,3 / 5,2 / 2,2 / 2,1 / 5,1

< an dieser Stelle soll der Routenplan in ASCII- und Listenform für eine
zweite Strategie ausgegeben werden, vgl. Aufgabenstellung >

zu versiegelnde Parzellen: 25
Hindernisparzellen: 0
versiegelte Parzellen: 25
nicht versiegelte Parzellen: 0
```

1.4 Anforderung an das Gesamtsystem

Das Programm kann funktional in ein Model, eine View und einen Controller unterteilt werden, die für spezielle Aufgaben zuständig sind. Der Controller liest über die View die Daten ein, speichert diese im Model und führt die Lösungsalgorithmen durch. Anschließend werden die Ergebnisse wieder über die View ausgegeben.

Um beide Algorithmen durchführen zu können, muss ein Grundgerüst an Software bestehen:

1. Interaktion mit Dateien, um Eingabe und Ausgabe zu benutzen

2. Ein Datenmodell, das die Daten vorhält und Zugriffe regelt. Dies sind die jeweiligen Felder der Strategien und der Routenplan.
3. Die Logik des Backtrackings wird vom Controller ausgeführt und benötigt View und Model als Kommunikationspartner.

Deshalb lässt sich mein Programm grob in diese Untermodule einteilen. Die komplette Durchführung besteht aus:

- Einlesen
- Speichern der Daten
- Berechnen der Lösungen (ggf. mehrere, hier 2)
- Ausgabe der Lösungen

Dabei ist es wichtig mögliche Fehler zu behandeln, um Abstürze und unerwartetes Verhalten zu vermeiden. Außerdem wird eine aussagekräftige Meldung ausgegeben, welcher Fehler aufgetreten ist. Die Robustheit des Programms wird mit Testfällen überprüft.

1.5 Grenzfälle

Als Grenzfall gibt es zwei Szenarien:

- Minimal-Szenario
  Die Fläche ist von der Größe 1x1. Der Startpunkt muss also gleich dem Ziel sein.

- Maximal-Szenario
  Die Fläche ist von der Größe 10x10. Hier ist die Laufzeit des Algorithmus maximal (falls keine Hindernisse vorhanden sind).

Weiterhin ist es möglich, dass eine Fläche vollständig durch Hindernisse blockiert ist, sodass nur der Startpunkt frei ist. Dies kann allerdings wieder auf den Minimalfall zurückgeführt werden.

1.6 Sonderfälle

Folgender Fall kann als Worst-Case-Szenario betrachtet werden, da die maximale Anzahl an Rechenschritten nötig ist um dieses Problem zu lösen: Die Fläche ist von der Größe 10x10, aber zwei Hindernissen blockieren eine Parzelle in einer Ecke. Weil es dann keine vollständige Lösung gibt, müssen alle möglichen Lösungen durchlaufen werden. Bei einem 10x10 Feld (mehr ist laut Aufgabenstellung nicht zugelassen) sind 97 von 100 Parzellen abzuarbeiten, weil zwei Parzellen durch Hindernisse blockiert werden und eine nie erreicht werden kann.

Ein zweiter Sonderfall tritt auf, wenn sich zwei oder mehr Hindernisse überlagern.

Es können auch andere *Whitespaces*[1] statt des Leerzeichens als Trennzeichen in einer Zeile auftauchen (z.B. Tabulator).

[1] Java erkennt die *Unicode-Whitespaces*: https://de.wikipedia.org/wiki/Leerraum#Unicode (Stand 16.05.2014)

1.7 Fehlerfälle

Die auftretenden Fehler können in grob in drei Kategorien aufgeteilt werden: technische, syntaktische und semantische Fehler. Technische Fehler liegen vor, wenn entweder die angegebene Datei nicht vorhanden ist oder Zugriffsrechte fehlen. Syntaktische Fehler treten auf, wenn die Eingabedatei die Formatvorgaben nicht korrekt einhält. Bei semantischen Fehlern sind fehlerhafte Werte (z.B. 0 als Reihe) enthalten.

Durch die Analyse der Aufgabenstellung und Eingabeanforderung ergeben sich folgende Fehlerfälle:

Technische Fehlerfälle

- Keine Eingabe-Datei vorhanden
- Keine Zugriffsrechte

Syntaktische Fehlerfälle

- Es fehlt die Startzeile oder die Anzahl der Felder
- Es werden ungültige Trennzeichen benutzt
- Es werden Fließkommazahlen eingegeben
- Ein Hindernis ist unzureichend definiert

Semantische Fehlerfälle

- Es sind mehrere Abmessungen oder Startparzellen in einer Zeile gegeben
- Es sind weniger als 1 oder mehr als 10 Parzellen in eine Richtung gegeben
- Der Startpunkt liegt außerhalb des Feldes
- Die Hindernisse liegen außerhalb des Feldes
- Der Startpunkt ist auch ein Hindernis

2 Verfahrensbeschreibung

Die in der in der Aufgabenstellung beschriebene Uhrzeiger-Strategie ist ein typisches Beispiel für einen Backtracking-Algorithmus. Dieser arbeitet nach dem „*trial and error*“-Prinzip (frei übersetzt: Versuch-und-Irrtum-Prinzip), das jede mögliche Lösung ausprobiert. Existiert eine große Menge an optimalen Lösungen, so ist es wahrscheinlich, dass der Algorithmus relativ schnell eine davon findet. Gibt es nur eine optimale Lösung, so kann man Fälle konstruieren, in der diese zuletzt gefunden wird. Aber sie wird mit Sicherheit immer gefunden. Falls es keine optimale Lösung gibt (hier: mindestens eine Parzelle kann nicht versiegelt werden) so muss der Algorithmus alle Lösungen durchprobieren, um die beste Näherung (hier: möglichst viele versiegelte Parzellen) zu bestimmen.

Meine Variante der vorgegebenen Strategie zur Lösung des Problems kann ebenfalls mit dem Backtracking-Algorithmus als Basis arbeiten. Daher bietet es sich an, den Backtracking-Algorithmus mit einer abstrahierten Strategie zu implementieren und die konkreten Strategien unabhängig durch geeignete Entwurfsmuster umzusetzen. Hier ist das *Strategy-Pattern* der Gang of Four besonders geeignet, das genau diese Anforderungen erfüllt. Zuerst wird nur die Schnittstelle definiert und die konkreten Implementierungen unterschiedlicher Strategien liegen gekapselt vor. Dadurch kann eine einzelne Strategie leicht ausgetauscht oder erweitert werden, ohne die anderen zu verändern. Dies entspricht auch dem *Open-Closed-Prinzip* der Softwaretechnik.

Laut Aufgabenstellung sollen in einem Durchlauf beide Algorithmen nacheinander ausgeführt und die Ergebnisse im direkten Vergleich hintereinander ausgegeben werden. Daher werden die konkreten Strategien im Folgenden näher erläutert.

2.1 Meine Strategie

Meine Strategie greift ebenfalls auf das Backtracking-Verfahren zurück, um eindeutig entscheiden zu können, ob es überhaupt eine gültige, vollständige Lösung gibt. Auf jeder Parzelle wird versucht zunächst die vorherige Richtung beizubehalten. Sollte dies nicht möglich sein, wird gegen den Uhrzeigersinn nacheinander die nächste Richtung probiert. Falls z.B. die letzte Richtung „nach links“ war, wird danach „nach unten“, dann „nach rechts“ und dann „nach oben“ probiert. Beim Start wird mit „nach oben“ begonnen. Bei dieser Strategie wird also verhältnismäßig selten die Richtung geändert, sodass ein relativ kurzer Routenplan erstellt wird. Außerdem lassen sich dadurch oft unnötige Schritte vermeiden, weil die vorherige Parzelle, von der man gekommen ist und die dann schon versiegelt ist, frühestens als vorletzter Fall betrachtet wird.

Wenn der von mir erstellte Algorithmus nicht weiter in die vorherige Richtung gehen kann, muss geprüft werden, ob alle Felder (außer den Hindernissen) abgearbeitet wurden. Falls ja, terminiert er, in allen anderen Fällen muss ein Schritt zurückgegangen werden. Dies wird solange wiederholt, bis eine gültige Lösung erreicht wurde oder alle Felder in allen Richtungen abgelaufen worden sind. Das macht den Algorithmus allerdings sehr aufwändig, wenn es keine vollständige Lösung gibt.

Nach Abschluss des Algorithmus kann auf die beste gespeicherte Routenliste als Ergebnis zurückgegriffen werden. Unabhängig davon, ob der Algorithmus mit oder ohne vollständige Lösung terminiert, erreicht die beste gespeicherte Routenliste die bestmögliche Versiegelung von Parzellen.

Erstellt wird die Liste während des Backtrackings nach folgendem Prinzip:

- Bei jedem Schritt wird die betrachtete Parzelle in die aktuelle Routenliste eingetragen.
- Übersteigt die Anzahl der durch die Routenliste abgedeckten Parzellen die Anzahl der durch die bisher als Beste gespeicherte Routenliste, wird die aktuelle Routenliste als neue beste Routenliste abgespeichert.
- Wenn von einer Parzelle ein Rückschritt erfolgt und die Parzelle damit unversiegelt ist, muss dieser aus der aktuellen Routenliste entfernt werden.

Im Worst-Case-Szenario hat dieser Algorithmus daher eine Laufzeit in der Klasse $O(4\hat{}k)$, wobei k die Anzahl aller Parzellen ist. In dem unter „Grenzfälle“ definierten Worst-Case-Szenario müssen also $4^{97} = 2{,}51 * 10^{58}$ Schritte gemacht werden.

2.2 Uhrzeiger-Strategie

Wie bereits skizziert, läuft die Uhrzeiger-Strategie so lange, bis ein vollständiges Ergebnis gefunden wurde oder alle Parzellen in alle Richtungen abgearbeitet wurden. Dieser Lösungsansatz entspricht dem Backtracking-Verfahren.

Bei jeder besuchten Parzelle wird versucht, in einer festen Reihenfolge der Richtungen weiter zur nächsten Parzelle zu gehen. Die Reihenfolge entspricht dem Uhrzeigersinn und beginnt immer mit der Richtung „nach oben“, egal aus welcher Richtung der Roboter vorher kam. Sind weitere Richtungen möglich werden alle von diesen ausprobiert. Wenn keine Richtung möglich ist und keine vollständige Lösung gefunden wurde, wird die Parzelle wieder als unversiegelt markiert und die vorherige Parzelle wird betrachtet. Ist in der vorherigen ebenfalls keine weitere Richtung mehr möglich, muss diese ebenfalls wieder als unversiegelt markiert werden usw.

Wenn eine vollständige Lösung gefunden ist, bricht die Rekursion ab. Sonst läuft diese alle Möglichkeiten ab und bestimmt daraus mithilfe eines Routenplans die Beste. Dies war nicht konkret

2.3 Beispiel mit unterschiedlichen Lösungen

Dass die beiden Strategien zu unterschiedlichen Ergebnissen kommen, zeigt folgendes Beispiel.

```
; Abmessung der Fläche
3 3
; Startparzelle
2 2
; Eckparzelle der Hindernisse
```

Startparzelle

	S	

Uhrzeiger-Strategie

Z	>	V
^	^	V
^	<	<

Meine Strategie

v	<	Z
v	^	^
>	>	^

Offensichtlich werden unterschiedliche Lösungen gefunden, die aber beide eine vollständige Route ohne Unterbrechungen finden. Daher ist im Sinne der Aufgabenstellung eine Strategie weder besser noch schlechter, solang die gleiche Anzahl an Parzellen versiegelt wird. Dieses Beispiel wird an späterer Stelle (s. Abschnitt 4.5) schrittweise erläutert um alle Details aufzeigen zu können.

2.4 Grenzfälle

Die oben genannten Grenzfälle werden wie folgt behandelt:

Die Fläche besteht aus einer einzigen Parzelle.
Das Programm wird normal ausgeführt.

Die Fläche besteht aus 10x10 Parzellen.
Das Programm wird normal ausgeführt.

2.5 Sonderfälle

Die oben genannten Sonderfälle werden wie folgt behandelt:

Worst-Case-Szenario: Die Fläche besteht aus 10x10 Parzellen mit einer nicht erreichbaren Ecke.
Das Programm wird normal ausgeführt, aber wahrscheinlich nicht in absehbarer Zeit terminieren.

Zwei oder mehr Hindernisse überlagern sich.
Das Programm wird normal ausgeführt.

Es werden *Whitespaces* als Trennzeichen benutzt.
Das Programm wird normal ausgeführt.

2.6 Fehlerfälle

Die oben genannten Fehlerfälle werden wie folgt behandelt:

Es ist keine Eingabedatei vorhanden.
Das Programm gibt eine Fehlernachricht aus und beendet sich.

Es sind nicht genügend Rechte im Dateisystem vorhanden.
Das Programm gibt eine Fehlernachricht aus und beendet sich.

Es fehlt die Startzeile oder die Anzahl der Felder.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

Es sind weniger als 1 oder mehr als 10 Parzellen in eine Richtung gegeben.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

Es werden ungültige Trennzeichen benutzt.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

Es werden Fließkommazahlen eingegeben.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

Ein Hindernis wird nicht ausreichend definiert.
Das Programm wird ohne das Hindernis ausgeführt, hängt aber eine Warnung an die Ausgabe.

Es sind mehrere Abmessungen oder Startparzellen in einer Zeile gegeben.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

Der Startpunkt liegt außerhalb des Feldes.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

Der Startpunkt ist auch ein Hindernis.
Das Programm schreibt einen Fehler in die Ausgabedatei und beendet sich.

2.7 Gesamtsystem

Ich habe den gesamten Quellcode in Java ohne Klassen oder Bibliotheken von Drittanbietern entwickelt.

2.7.1 Main-Funktion

Die *main*-Funktion liest die benötigten Pfade aus den Übergabeparametern aus und initialisiert damit den Controller. Anschließend startet sie den Controller. Weitere Operationen erfolgen im *Controller*.

2.7.2 Model

Die Daten werden in der Klasse *Model* verwaltet. Dabei kann nicht direkt auf die Daten, sondern nur über die Schnittstellenmethoden auf diese zugegriffen werden. Dies erlaubt eine leichte Handhabung für Erweiterungen und hält die Kommunikation übersichtlich.

2.7.3 View

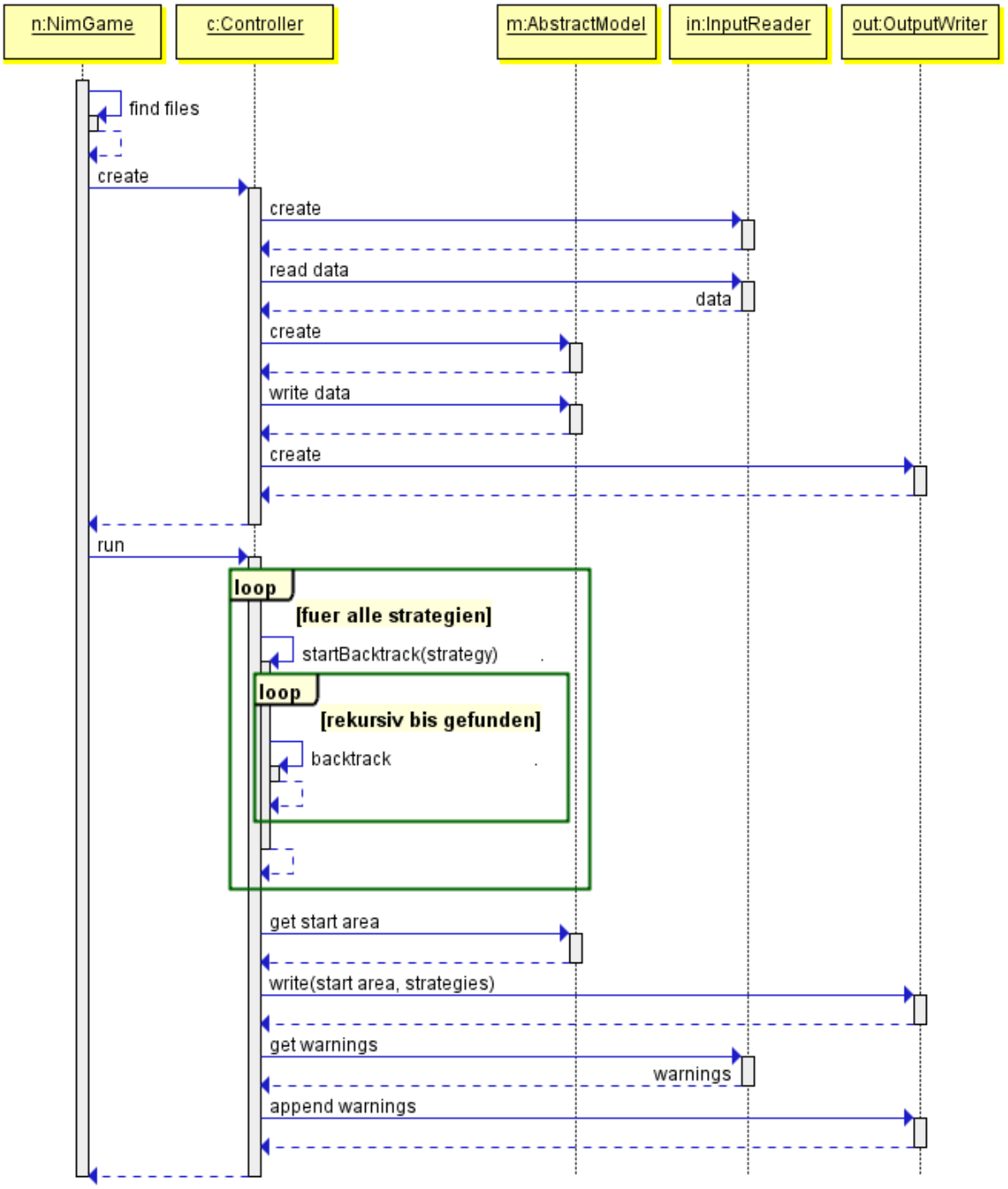
Zur *View* gehören sowohl Eingabe als auch Ausgabe. Hier werden also Methoden zum Einlesen von Daten (Klasse *InputFileReader*) und zur Ausgabe (Klasse *OutputFileWriter*) von der Statistik bereitgestellt. Auch Warnungen und Fehler können ausgegeben werden.

2.7.4 Controller

Der *Controller* beinhaltet die eigentliche Logik des Programms. Dieser startet das Einlesen aus der Datei, schreibt die Daten in das Model und führt alle Berechnungen und Durchläufe aus. Abschließend wird das Ergebnis in die Ausgabe geschrieben.

Die Strategien werden hier verwaltet und ausgeführt, sodass der gesamte Prozess abgekapselt ist.

2.8 Gesamtablauf – Sequenzdiagramm

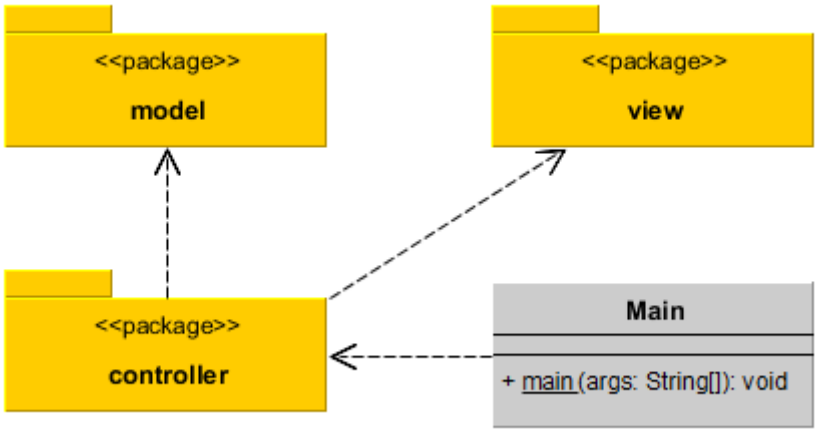


3 Programmbeschreibung

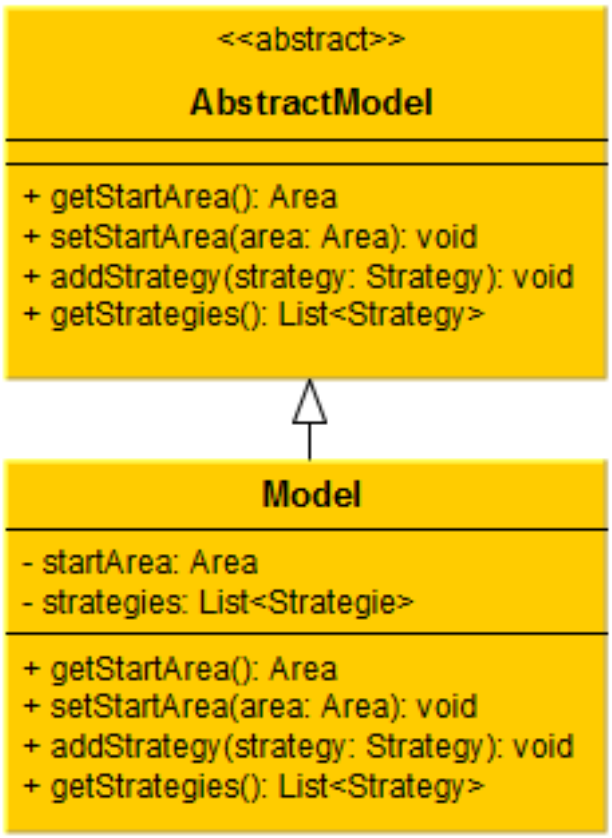
3.1 Pakete

Das Programm wird in drei Pakete unterteilt: *Model*, *View* und *Controller*. Dabei sind Teile des Controllers, nämlich die Strategien, als eigenes Unterpaket zusammengefasst. Für die Model und die View wurden abstrakte Klassen als Schnittstelle definiert. Diese bieten alle wichtigen Funktionen und werden jeweils in einer Unterklasse konkret implementiert. Dadurch bleibt das Programm leicht erweiterbar.

Die UML-Gesamtübersicht aller Klassen und Methoden kann aufgrund der hohen Auflösung und Bildgröße leider nicht an dieser Stelle dargestellt werden. Sie ist aber unter folgendem Pfad auffindbar: *diagrams/uml/UMLDetailed.png*.



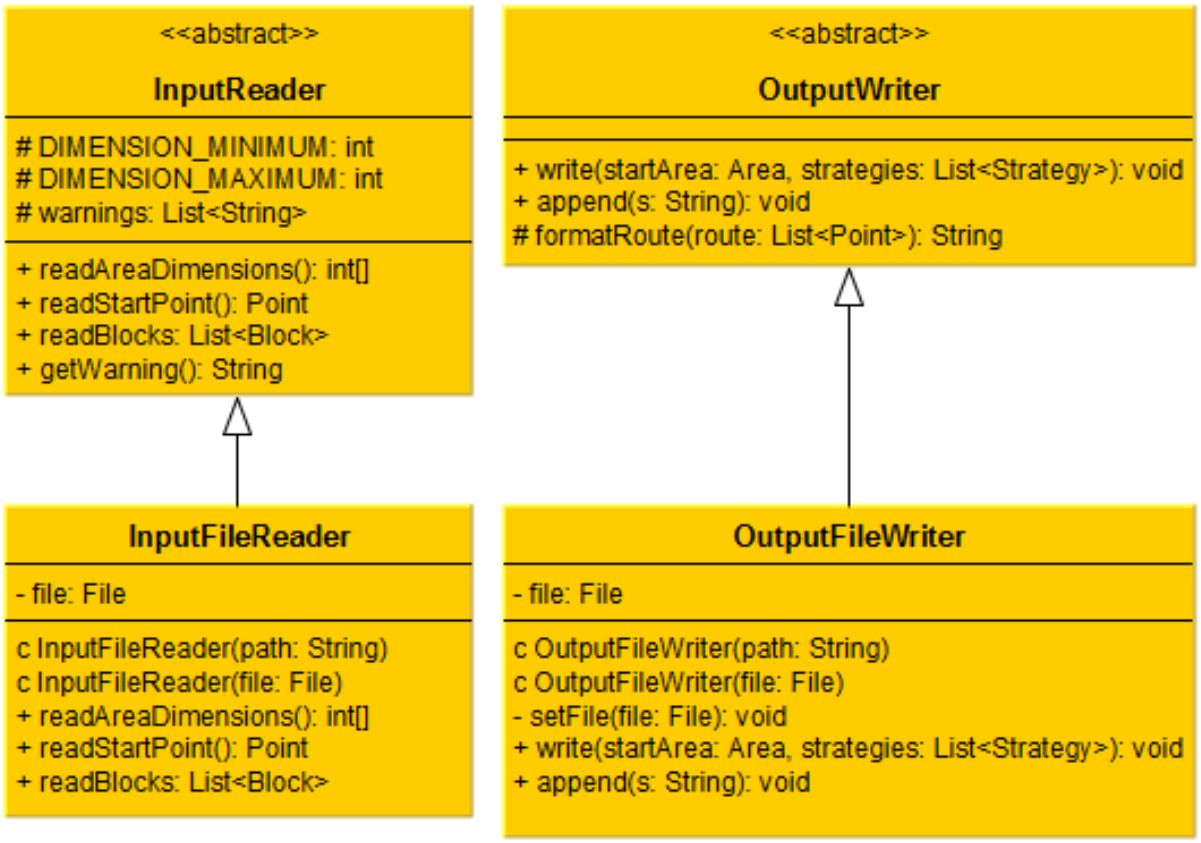
3.1.1 Model-Klassen



AbstractModel
Definiert eine Schnittstelle, über die alle benötigten Daten gespeichert und geholt werden können.

Model
Konkrete Implementierung eines Models mit Variablen. Die Daten werden also im Arbeitsspeicher gehalten.

3.1.2 View-Klassen



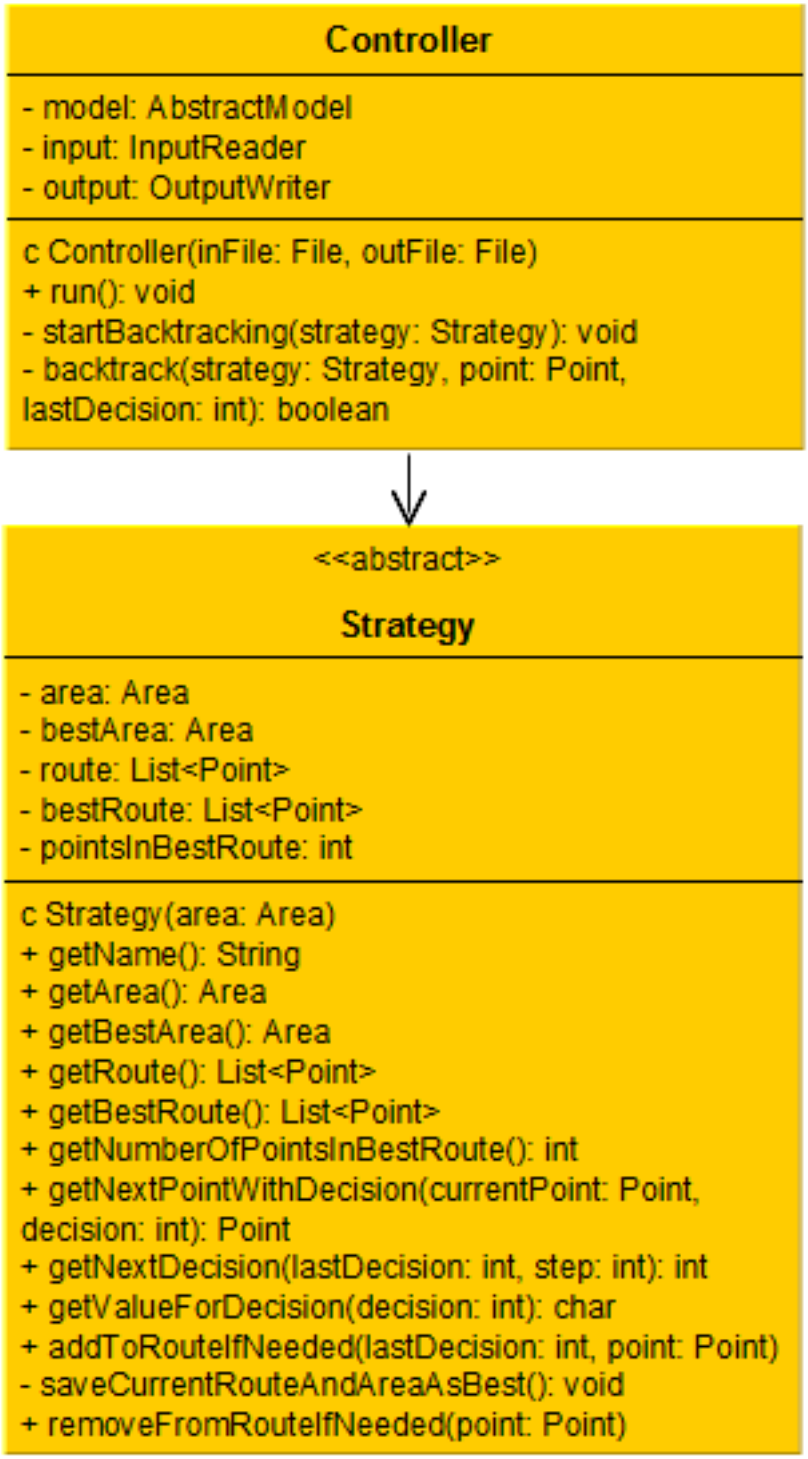
InputReader
Schnittstelle zum Einlesen von Eingabedaten.

InputFileReader
Konkrete Implementierung zum Einlesen über eine Datei.

OutputWriter
Schnittstelle zum Ausgaben von Daten.

OutputFileWriter
Konkrete Implementierung zur Ausgabe in eine Datei.

3.1.3 Controller-Klassen



Controller

Startet das Programm und beinhaltet die Logik des Lösungsalgorithmus, in diesem Fall das Backtracking.

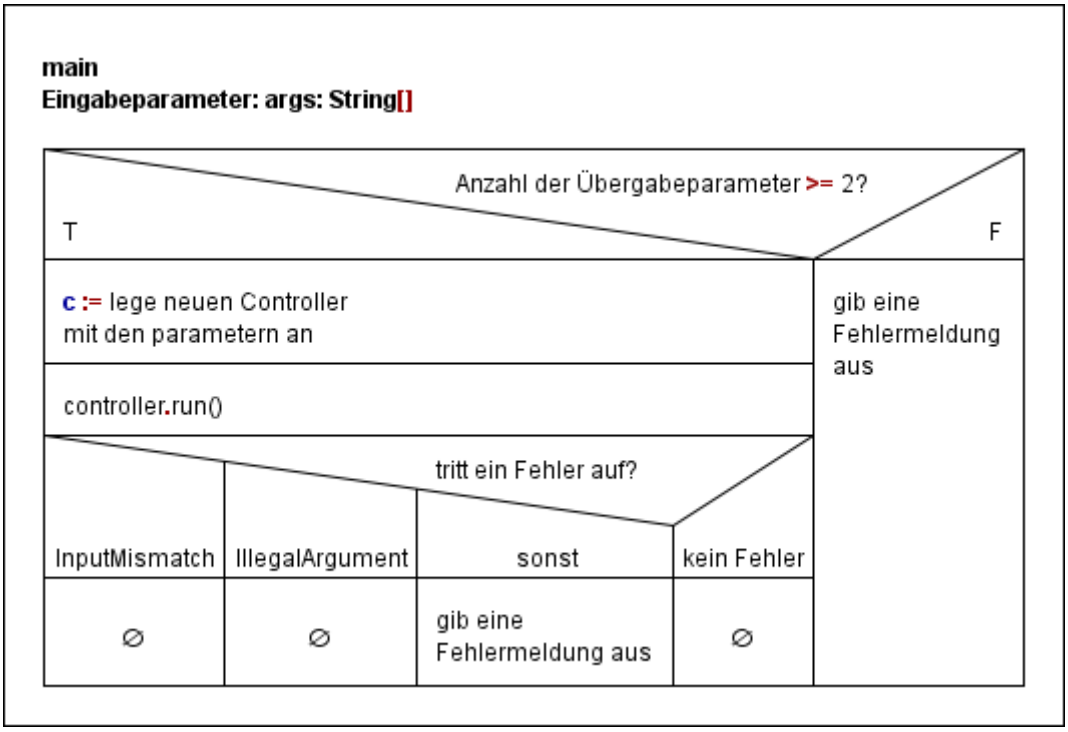
Strategy

Stellt eine Schnittstelle zum Ausführen einer möglichen Strategie bereit.

3.2 Präzisierungen

Die wichtigsten Präzisierungen in Form von Nassi-Shneidermann-Diagrammen sind nachfolgend aufgelistet. Weitere Diagramme zum Einlesen von Daten, Nachhalten der Route usw. sind unter folgendem Pfad als Originaldatei und Bild abgelegt: *diagrams/structograms*.

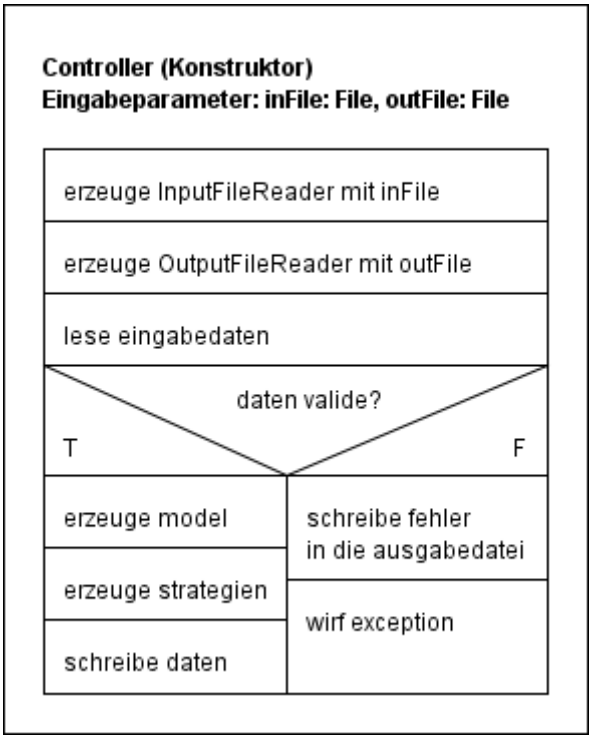
3.2.1 Präzisierung – main



Diese Methode wird bei der Ausführung des Programms aufgerufen. Anfangs wird geprüft ob zwei Übergabeparameter vorhanden sind, nämlich die Pfade der Eingabe- und Ausgabedatei. Falls zu wenige Parameter mitgegeben wurden, wird eine Fehlernachricht ausgegeben, dass mindestens zwei Parameter benötigt werden. Wenn beide Parameter existieren, wird ein neuer Controller mit diesen initialisiert und das Programm wird ausgeführt.

Wenn während dem Initialisieren oder der Ausführung ein Fehler auftritt (sog. *Exception*) wird dieser nur behandelt, wenn es keine *InputMismatch*- oder *IllegalArgumentException* ist. Bei den beiden Spezialfällen werden schon vorher die Fehlermeldungen in die Ausgabedatei geschrieben und das Programm wird korrekt beendet.

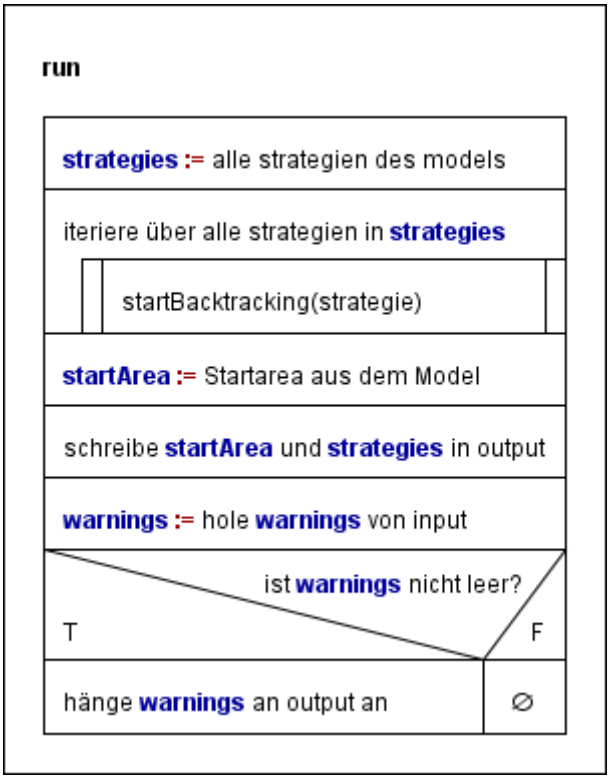
3.2.2 Präzisierung – Controller-Konstruktor



In dem Konstruktor des *Controllers* werden die grundlegenden Objekte erzeugt und initialisiert. Zuerst werden die Views erzeugt, also Ein- und Ausgabe. Die Ausgabe muss direkt am Anfang erzeugt werden, um diesen im Fehlerfall ausgeben zu können.

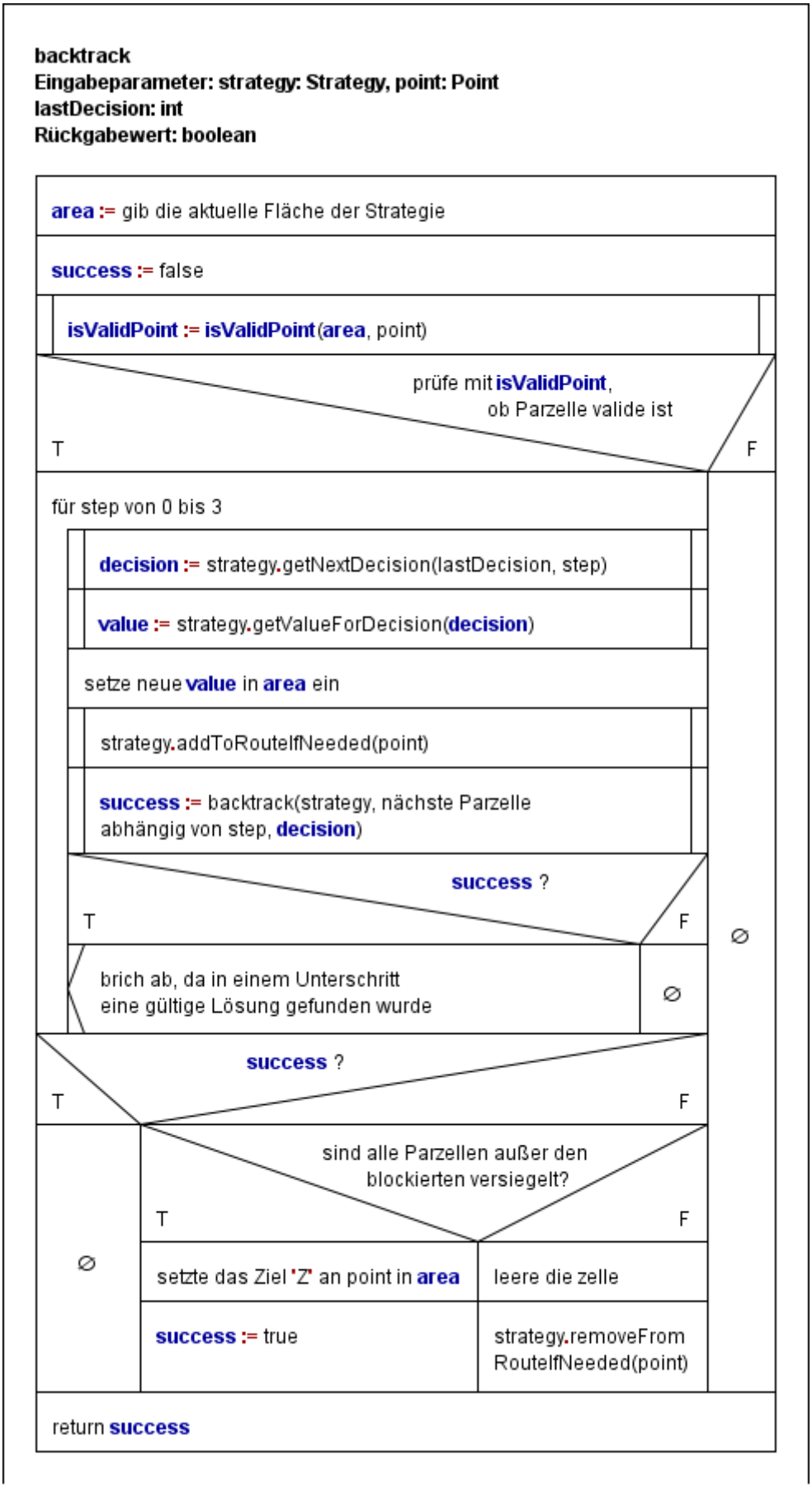
Dann werden die Daten eingelesen und mit den daraus erzeugten Strategien in das Model geschrieben. Falls *Exceptions* auftreten, wird die Nachricht in die Ausgabedatei geschrieben und die Exception wird weitergeworfen, um zu signalisieren, dass kein gültiges Objekt erzeugt werden kann.

3.2.3 Präzisierung – run



Startet das eigentliche Programm. Dazu werden alle Strategien abgearbeitet, *indem startBacktracking(strategie)* für jede Strategie einmal ausgeführt wird. Anschließend werden die Ergebnisse in die Ausgabe geschrieben und falls Warnungen existieren, werden diese an das Ergebnis anhängt.

3.2.4 Präzisierung – backtrack



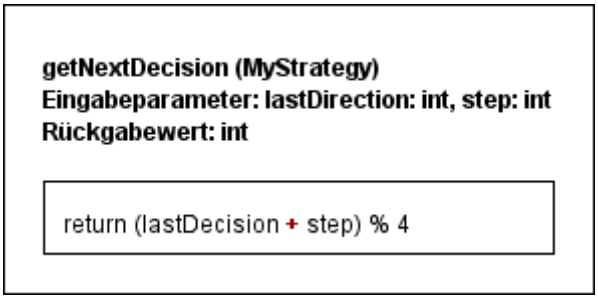
In dieser Methode wird eine Strategie komplett ausgeführt, sodass ein Lösungsweg gefunden wird. Dieser ist nach der Ausführung vollständig, sofern dies möglich ist, oder versiegelt so viele Parzellen wie möglich.

Zuerst werden einige Prüfungen (*isValidPoint*) durchgeführt, um festzustellen, ob von der jeweiligen Parzelle aus der Lösungsweg weitergesucht werden kann. Falls nicht, wird zum Ende gesprungen. Falls ja, werden alle vier weiteren Schritte durchgeführt. Welcher Schritt in welche Richtung geht muss die konkrete Strategie entscheiden (*getNextDecision(lastDecision, step)*), für das Backtracking an sich ist es erst einmal unwichtig. Wenn in einem Unterschritt, d.h. in einer gegebenen Richtung, eine vollständige Lösung gefunden wurde, wird die Schleife abgebrochen.

Wenn keine vollständige Lösung gefunden wurde, werden alle weiteren Schritte (für die weiteren Richtungen) genauso durchgeführt. Falls danach immer noch keine vollständige Lösung gefunden ist, muss geprüft werden, ob die aktuelle Lösung bereits die vollständige ist. Wenn ja wird die aktuelle Position als Ziel markiert und *success* wird auf *true* gesetzt. Dadurch bricht in allen unterliegenden Rekursionsschritten auf dem Stack die Schleife ab und der Algorithmus terminiert. Wenn noch nicht alle Parzellen versiegelt sind, wird der letzte Schritt rückgängig gemacht, sodass die Parzelle durch einen anderen Weg versiegelt werden kann.

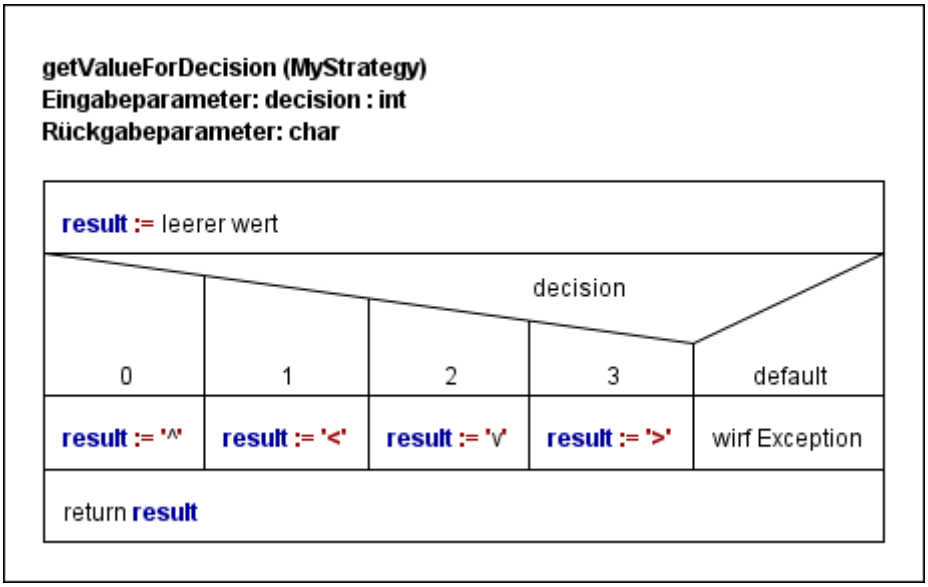
Während der Schleife werden die versiegelten Parzellen zum Routenplan hinzugefügt und wenn ein Schritt rückgängig gemacht wird, auch wieder entfernt. Dies stellt sicher, dass bei einer Lösung immer ein Routenplan existiert. Ist die Lösung unvollständig werden natürlich auch alle Parzellen wieder entfernt. Da aber die Methode *addToRouteIfNeeded(point)* entscheidet, wann eine neue beste Route erzeugt wurde und diese dann zwischenspeichert, kann später im Programmverlauf die beste Route einfach ausgelesen werden.

3.2.5 Präzisierung –getNextDecision (MyStrategy)



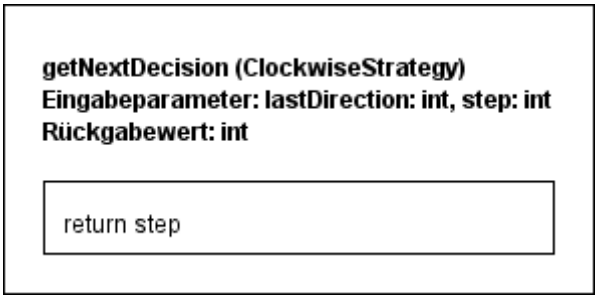
Diese Methode sorgt dafür, dass im ersten Schritt im die letzte Richtung beibehalten wird. In weiteren Schritten werden dann die anderen Richtungen gewählt, wobei zu berücksichtigen ist, dass nur die ganzzahligen Werte 0 bis 3 gültig sind.

3.2.6 Präzisierung –getValueForDecision (MyStrategy)



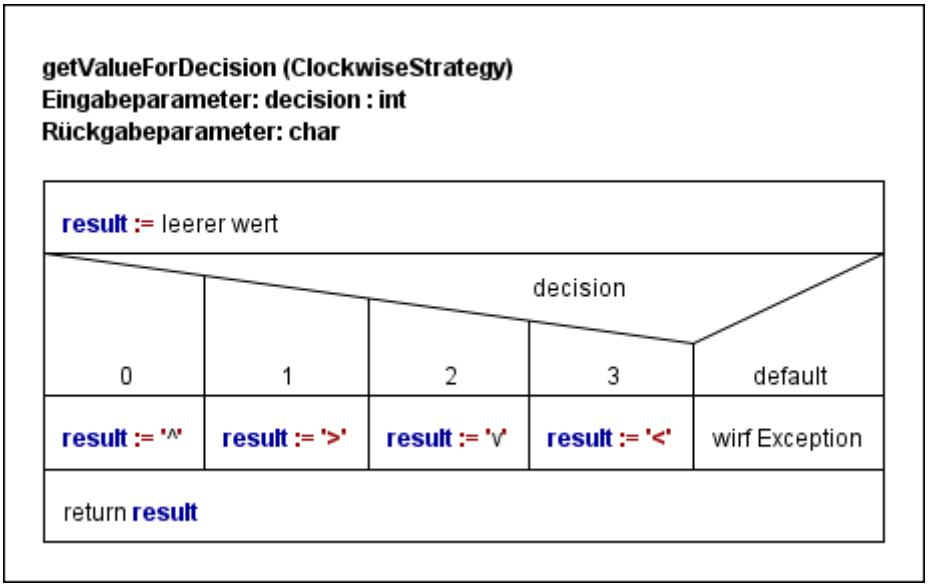
Die Richtungen werden gegen den Uhrzeigersinn ausgewertet. Wenn eine Richtung übergeben wird, die nicht zwischen 0 und 3 liegt, wird eine *Exception* geworfen.

3.2.7 Präzisierung –getNextDecision (ClockwiseStrategy)



Die Uhrzeigerstrategie probiert an jeder Parzelle immer neu die Richtungen von 0 bis 3 aus. Daher muss nur der entsprechende Schritt als nächste Richtung zurückgegeben werden.

3.2.8 Präzisierung –getValueForDecision (ClockwiseStrategy)



Die Richtungen werden im Uhrzeigersinn ausgewertet. Wenn eine Richtung übergeben wird, die nicht zwischen 0 und 3 liegt, wird eine *Exception* geworfen.

4 Testdokumentation

Parallel zu der Entwicklung des Softwaresystems wurden Tests geschrieben, die die Funktionalität überprüfen. Bei jeder Änderung wurden die Tests durchgeführt, um mögliche Fehler frühzeitig aufzudecken. Die Tests sind in Form von Eingabedateien mit der Dateiendung *.in* im Ordner „*tests*“ abgelegt. Die Eingabedateien haben jeweils ein Präfix vor dem Namen, um ihre Klasse zu kennzeichnen:

- n_ bedeutet Normalfall
- s_ bedeutet Sonderfall
- e_ bedeutet Fehlerfall (engl. error case)
- b_ bedeutet Grenzfall (engl. border case)

4.1 Normalfall

Ein Normalfall liegt vor, wenn keine Grenzfälle, Sonderfälle oder Fehlerfälle auftreten. Die in der Aufgabenstellung abgedruckten Beispiele sind allesamt Normalfälle und wurden erfolgreich durchgeführt.

Das erste Beispiel mit der Eingabedatei *n_ihk_example1.in* wird korrekt ausgeführt und beide Strategien finden einen vollständigen Lösungsweg. Da keine Hindernisse vorkommen, kann jede Parzelle versiegelt werden. Hier wird gut sichtbar, wie die von mir entwickelte Strategie so lange die gleiche Richtung beibehält bis es nicht mehr anders geht: Der Lösungsweg führt wie eine Spirale in die Mitte.

Die Hindernisse im zweiten Beispiel mit der Eingabedatei n*_ihk_example2.in* werden problemlos umgangen, sodass ebenfalls ein vollständiger Lösungsweg gefunden wird. Die Hindernisse sind wie eine Mauer aufgebaut, die an einer Stelle überwunden werden kann. Weil „dahinter“ noch zwei Spalten frei sind, ist es keine „Sackgasse“.

Im dritten Beispiel mit der Eingabedatei *n_ihk_example3.in* existieren nur zwei Routen, welche die maximale Anzahl an Parzellen versiegeln. Dies liegt daran, dass zwischen dem Startpunkt und den Hindernissen nur eine freie Parzelle ist und somit „Sackgassen“ entstehen, aus denen es keinen Rückweg gibt. Hier finden beide Strategien dieselbe Lösung.

Das Beispiel mit der Eingabedatei *n_own_example3x3.in* wird in dem Abschnitt 4.5 als einfaches Beispiel detailliert aufgeführt. Die beiden Strategien kommen zu unterschiedlichen Lösungen, die beide vollständig sind.

Ein weiteres von mir entwickeltes Beispiel ist eine Art „Labyrinth“, in dem der Weg zwischen mehreren Hindernissen entlangführt. Die zugehörige Eingabedatei heißt *n_own_example_labyrinth.in*. In dem Beispiel sieht man, wie die erzeugte Route möglichst viele Felder versiegelt, aber eben nicht alle.

4.2 Sonderfall

Das Worst-Case-Szenario kann nicht als Beispiel ausgeführt werden, da es wahrscheinlich nicht in absehbarer Zeit terminiert. Dies hängt natürlich auch bis zu einem gewissen Grad von der Rechenleistung des Computers ab.

Um ein Beispiel zu geben, dass dieses Problem darstellt, habe ich den Aufbau auf ein 6x6 Feld übertragen. Diese Aufgabe zu lösen dauert in meiner Entwicklungsumgebung etwa 10-15 Sekunden.

Eingabe (*s_worstcase6x6.in*):

```
; Abmessungen der Flaeche
6 6
; Startparzelle
1 1
; Eckepunkte
6 5 6 5
5 6 5 6
```

Ausgabe:

```
Startpunkt (1,1)
  1 2 3 4 5 6
1 S
2
3
4
5           H
6         H

Uhrzeiger-Strategie:
  1 2 3 4 5 6
1 > > > > > v
2 v < < v < v
3 > v ^ v ^ v
4 v < ^ < ^ <
5 v > v > Z H
6 > ^ > ^ H

Routenplan:
1,1 / 1,6 / 4,6 / 4,5 / 2,5 / 2,4 / 4,4 / 4,3 / 2,3 / 2,1 / 3,1 / 3,2 / 4,2
/ 4,1 / 6,1 / 6,2 / 5,2 / 5,3 / 6,3 / 6,4 / 5,4 / 5,5

Meine Strategie:
  1 2 3 4 5 6
1 v > > > > v
2 v ^ v < v <
3 v ^ v ^ > v
4 v ^ v ^ v <
5 v ^ < ^ Z H
6 > > > ^ H

Routenplan:
1,1 / 6,1 / 6,4 / 2,4 / 2,3 / 5,3 / 5,2 / 1,2 / 1,6 / 2,6 / 2,5 / 3,5 / 3,6
/ 4,6 / 4,5 / 5,5

zu versiegelnde Parzellen: 34
Hindernisparzellen: 2
versiegelte Parzellen: 33
nicht versiegelte Parzellen: 1
```

Die nächste größere Variante des Worst-Case-Szenarios ist ein 6x7 Feld. Dieses Problem zu lösen dauert in meiner Entwicklungsumgebung schon etwa 15 Minuten. Die Eingabedatei sieht folgendermaßen aus:

```
; Abmessungen der Flaeche
6 7
; Startpunkt
1 1
; Eckpunkte
5 7 5 7
6 6 6 6
```

Der zweite Sonderfall tritt auf, wenn sich zwei oder mehr Hindernisse überlagern. Dies wird in dem Test mit der Eingabedatei s_*blocks_overlaying.in* durchgeführt und stellt keinerlei Probleme dar.

Der dritte Sonderfall besteht, wenn statt einem Leerzeichen als Trennzeichen Tabulatoren oder andere sogenannte *Whitespaces* (Leerräume) verwendet werden. Da aber vor dem Parsen einer Zeile zuerst die Methode *trim()* aufgerufen wird und anschließend nach einem oder mehreren *Whitespaces* der String aufgeteilt wird, treten hier keine Probleme auf. Das Umwandeln in eine Ganzzahl funktioniert fehlerfrei und das Programm wird korrekt ausgeführt. Der entsprechende Test wird mit der Eingabedatei s_*whitespaces_divider.in* durchgeführt.

4.3 Grenzfall

Die festgelegten Grenzfälle einer minimalen und maximalen Eingabedatei sind ebenfalls in den vordefinierten Tests enthalten.

Minimalfall

Eingabe (*b_minimal.in)*:

```
; Abmessungen der Flaeche
1 1
; Startparzelle
1 1
; Eckepunkte
```

Ausgabe:

```
Startpunkt (1,1)
  1
1 S

Uhrzeiger-Strategie:
  1
1 Z

Routenplan:
1,1

Meine Strategie:
  1
1 Z
```

```
Routenplan:
1,1

zu versiegelnde Parzellen: 1
Hindernisparzellen: 0
versiegelte Parzellen: 1
nicht versiegelte Parzellen: 0
```

Maximalbeispiel

Eingabe (*b_maximal.in*):

```
; Abmessungen der Flaeche
10 10
; Startparzelle
1 1
; Eckpunkte
```

Ausgabe:

```
Startpunkt (1,1)
  1 2 3 4 5 6 7 8 9 10
1 S
2
3
4
5
6
7
8
9
10

Uhrzeiger-Strategie:
  1 2 3 4 5 6 7 8 9 10
1 > > > > > > > > > v
2 Z v < v < v < v < v
3 ^ v ^ v ^ v ^ v ^ v
4 ^ v ^ v ^ v ^ v ^ v
5 ^ v ^ v ^ v ^ v ^ v
6 ^ v ^ v ^ v ^ v ^ v
7 ^ v ^ v ^ v ^ v ^ v
8 ^ v ^ v ^ v ^ v ^ v
9 ^ v ^ v ^ v ^ v ^ v
10 ^ < ^ < ^ < ^ < ^ <

Routenplan:
1,1 / 1,10 / 10,10 / 10,9 / 2,9 / 2,8 / 10,8 / 10,7 / 2,7 / 2,6 / 10,6 /
10,5 / 2,5 / 2,4 / 10,4 / 10,3 / 2,3 / 2,2 / 10,2 / 10,1 / 2,1

Meine Strategie:
  1 2 3 4 5 6 7 8 9 10
1 v v < < < < < < < <
2 v v v < < < < < < ^
3 v v v v < < < < ^ ^
```

```
4 v v v v v < < ^ ^ ^
5 v v v v v Z ^ ^ ^ ^
6 v v v v > ^ ^ ^ ^ ^
7 v v v > > > ^ ^ ^ ^
8 v v > > > > > ^ ^ ^
9 v > > > > > > > ^ ^
10 > > > > > > > > > ^

Routenplan:
1,1 / 10,1 / 10,10 / 1,10 / 1,2 / 9,2 / 9,9 / 2,9 / 2,3 / 8,3 / 8,8 / 3,8 /
3,4 / 7,4 / 7,7 / 4,7 / 4,5 / 6,5 / 6,6 / 5,6

zu versiegelnde Parzellen: 100
Hindernisparzellen: 0
versiegelte Parzellen: 100
nicht versiegelte Parzellen: 0
```

4.4 Fehlerfall

Es sind neun Tests für Fehlerfälle enthalten. In allen Fällen wird, falls nötig, eine Fehleranalyse oder eine Warnung an die Ausgabe angehängt. Danach wird das Programm korrekt beendet.

Beschreibung	***Eingabedatei***	**Erwartete Ausgabe**
Die Startzeile fehlt	*e_missing_start.in*	Fehler: Es muss ein Startpunkt definiert sein!
Weniger als 1 Parzelle in einer Richtung gegeben	*e_zero_dimensions.in*	Fehler: Die Abmessung der Fläche muss zwischen 1 und 10 liegen!
Mehr als 10 Parzellen in einer Richtung gegeben	*e_large_dimensions.in*	Fehler: Die Abmessung der Fläche muss zwischen 1 und 10 liegen!
Ungültige Trennzeichen	*e_invalid_divider.in*	Fehler: Es müssen genau zwei Werte als Abmessungen der Fläche gegeben sein!
Fließkommazahlen	*e_float_values.in*	Fehler: Die Abmessung der Fläche muss ganzzahlig sein!
Mehrere Abmessungen in einer Zeile	*e_multiple_dimensions.in*	Fehler: Es müssen genau zwei Werte als Abmessungen der Fläche gegeben sein!
Startpunkt außerhalb des Feldes	*e_invalid_start.in*	Fehler: Der Startpunkt muss in der Fläche liegen!
Startpunkt ist ein Hindernis	*e_start_is_blocked.in*	Fehler: Der Block darf den Startpunkt nicht überschneiden!

Hindernisse nicht ausreichend definiert	*e_undefined_block.in*	Normale Ausführen ohne Hindernis mit Warnung: Das Hindernis wurde ignoriert, da es unzureichend definiert ist.

4.5 Ausführliches Beispiel

Um den Algorithmus schrittweise zu erläutern, verwende ich das Beispiel aus der Klausur am Montag, in dem ein 3x3 Feld ohne Hindernisse gegeben ist. Dies ist ein Normalfall, der problemlos durchgeführt wird.

Als erstes werden die Abmessungen und die Startparzelle eingelesen und in das Model geschrieben. Die beiden Strategien werden mit der Startfläche initialisiert und ebenfalls ins Model geschrieben. Danach beginnt der Backtracking-Algorithmus zuerst mit der Uhrzeiger-Strategie und danach mit meiner Strategie. Dies ist in den folgenden Abschnitten detailliert beschrieben. Abschließend werden die Startfläche und die Lösungen der Strategien in die Ausgabe geschrieben.

Eingabe:

```
; Abmessung der Fläche
3 3
; Startparzelle
2 2
; Eckparzelle der Hindernisse
```

Zum besseren Verständnis füge ich immer dem jeweils aktuellen Feld die Übergabeparameter hinzu. Vor den Variablennamen sind mehrere Unterstriche („_“) um die Rekursionstiefe zu symbolisieren.

4.5.1 ClockwiseStrategy

Erster Schritt:

```
  1 2 3
1
2   S
3
```

```
_point = 2,2
_lastDecision = 0
```

```
Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten
durchgeführt:
```

```
_step = 0
_decision = 0
_value = ^
```

Zweiter Schritt:

```
  1 2 3
1
2   ^
3
```

```
__point = 1,2
__lastDecision = 0
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
__step = 0
__decision = 0
__value = ^
```

Dritter Schritt:

```
  1 2 3
1   ^
2   ^
3
```

```
___point = 0,2
___lastDecision = 0
```

Weil die Parzelle 0,2 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
__step = 1
__decision = 1
__value = >
```

Vierter Schritt:

```
  1 2 3
1   >
2   ^
3
```

```
___point = 1,3
___lastDecision = 1
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
___step = 0
___decision = 0
___value = ^
```

Fünfter Schritt:

```
  1 2 3
1   > ^
2   ^
3
```

____point = 0,3
____lastDecision = 0

Weil die Parzelle 0,3 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

___step = 1
___decision = 1
___value = >

Sechster Schritt:

```
  1 2 3
1   > >
2   ^
3
```

____point = 1,4
____lastDecision = 1

Weil die Parzelle 1,4 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

___step = 2
___decision = 2
___value = v

Siebter Schritt:

```
  1 2 3
1   > v
2   ^
3
```

____point = 2,3
____lastDecision = 2

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

____step = 0
____decision = 0
____value = ^

Achter Schritt:

```
  1 2 3
1   > v
2   ^ ^
3
```

_____point = 1,3
_____lastDecision = 0

Weil die Parzelle 1,3 schon versiegelt ist, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

____step = 1
____decision = 1
____value = >

Neunter Schritt:

```
  1 2 3
1   > v
2   ^ >
3
```

_____point = 2,4
_____lastDecision = 1

Weil die Parzelle 2,4 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

____step = 2
____decision = 2
____value = v

Zehnter Schritt:

```
  1 2 3
1   > v
2   ^ v
3
```

_____point = 3,3
_____lastDecision = 2

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

_____step = 0
_____decision = 0
_____value = ^

Elfter Schritt:

```
  1 2 3
1   > v
2   ^ v
3     ^
```

_____point = 2,3
_____lastDecision = 0

Weil die Parzelle 2,3 schon versiegelt ist, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
_____step = 1
_____decision = 1
_____value = >
```

Zwölfter Schritt:

```
  1 2 3
1   > v
2   ^ v
3     >

______point = 3,4
______lastDecision = 1
```

Weil die Parzelle 3,4 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
_____step = 2
_____decision = 2
_____value = v
```

13. Schritt:

```
  1 2 3
1   > v
2   ^ v
3     v

______point = 4,3
______lastDecision = 2
```

Weil die Parzelle 4,3 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
_____step = 3
_____decision = 3
_____value = <
```

14. Schritt:

```
  1 2 3
1   > v
2   ^ v
3     <

______point = 3,2
______lastDecision = 3
```

Weil die Parzelle valide ist, wird der nächste Schritt durchgeführt. Der Algorithmus arbeitet diese Schritte immer weiter ab, bis zum vorletzten Feld:

15. Schritt:

```
  1 2 3
1   > v
```

```
2 ^ ^ v
3 ^ < <

_________point = 1,1
_________lastDecision = 0
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
_________step = 0
_________decision = 0
_________value = ^
```

16. Schritt

```
  1 2 3
1 ^ > v
2 ^ ^ v
3 ^ < <

__________point = 0,1
__________lastDecision = 0
```

Der Algorithmus prüft nun noch alle vier Richtungen, bevor klar ist, dass das Feld bereits vollständig versiegelt wurde. Dann wird an die letzte Stelle ein ‚Z' für fertig geschrieben und alle Rekursionsschritte springen zum Ende.

```
  1 2 3
1 Z > v
2 ^ ^ v
3 ^ < <
```

Der Routenplan ist hier nicht in jedem Schritt abgedruckt, da dieser immer nur die aktuellen Punkte hinzufügt bzw. beim Rückschritt entfernt. Vollständig sieht dieser so aus:
[2,2, 1,2, 1,3, 2,3, 3,3, 3,2, 3,1, 2,1, 1,1]

Wenn man die nicht benötigten Werte herauskürzt erhält man den Routenplan der Ausgabe:
2,2 / 1,2 / 1,3 / 3,3 / 3,1 / 1,1

4.5.2 MyStrategy

Erster Schritt:

```
  1 2 3
1
2   S
3

_point = 2,2
_lastDecision = 0
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
_step = 0
_decision = 0
_value = ^
```

Zweiter Schritt:

```
  1 2 3
1
2   ^
3

__point = 1,2
__lastDecision = 0
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
__step = 0
__decision = 0
__value = ^
```

Dritter Schritt:

```
  1 2 3
1   ^
2   ^
3

___point = 0,2
___lastDecision = 0
```

Weil die Parzelle 0,2 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
__step = 1
__decision = 1
__value = <
```

Vierter Schritt:

```
  1 2 3
1   <
2   ^
3

___point = 1,1
___lastDecision = 1
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
___step = 0
___decision = 1
___value = <
```

Fünfter Schritt:

```
  1 2 3
1 < <
2   ^
3

____point = 1,0
____lastDecision = 1
```

Weil die Parzelle 1,0 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
___step = 1
___decision = 2
___value = v
```

Sechster Schritt:

```
  1 2 3
1 v <
2   ^
3

____point = 2,1
____lastDecision = 2
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
____step = 0
____decision = 2
____value = v
```

Siebter Schritt:

```
  1 2 3
1 v <
2 v ^
3

_____point = 3,1
_____lastDecision = 2
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
_____step = 0
_____decision = 2
_____value = v
```

Achter Schritt:

```
  1 2 3
1 v <
2 v ^
3 v

______point = 4,1
______lastDecision = 2
```

Weil die Parzelle 4,1 außerhalb der Grenzen liegt, wird wieder ein Schritt zurückgesprungen und step erhöht sich um 1:

```
_____step = 1
_____decision = 3
_____value = >
```

Neunter Schritt

```
  1 2 3
1 v <
2 v ^
3 >

______point = 3,2
______lastDecision = 3
```

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

```
______step = 0
______decision = 3
______value = >
```

Zehnter Schritt:

```
  1 2 3
1 v <
2 v ^
3 > >

_______point = 3,3
_______lastDecision = 3
```

Weil die Parzelle valide ist, wird der nächste Schritt durchgeführt. Der Algorithmus arbeitet diese Schritte immer weiter ab, bis zum vorletzten Feld.

Elfter Schritt:

```
  1 2 3
1 v <
2 v ^ ^
3 > > ^
```

_____point = 1,3
_____lastDecision = 0

Weil die Parzelle valide ist, wird der nächste Schritt mit folgenden Werten durchgeführt:

_____step = 0
_____decision = 0
_____value = ^

Zwölfter Schritt:

```
  1 2 3
1 v < ^
2 v ^ ^
3 > > ^
```

_____point = 0,3
_____lastDecision = 0

Der Algorithmus prüft nun noch alle vier Richtungen, bevor klar ist, dass das Feld bereits vollständig versiegelt wurde. Dann wird an die letzte Stelle ein ‚Z' für fertig geschrieben und alle Rekursionsschritte springen zum Ende.

```
  1 2 3
1 v < Z
2 v ^ ^
3 > > ^
```

Der Routenplan ist hier nicht in jedem Schritt abgedruckt, da dieser immer nur die aktuellen Punkte hinzufügt bzw. beim Rückschritt entfernt. Vollständig sieht dieser so aus:
[2,2, 1,2, 1,1, 2,1, 3,1, 3,2, 3,3, 2,3, 1,3]

Wenn man die nicht benötigten Werte herauskürzt, erhält man den Routenplan der Ausgabe:
2,2 / 1,2 / 1,3 / 3,3 / 3,1 / 1,1

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

In dieser Arbeit habe ich ein Softwaresystem in mehreren Schritten umgesetzt. Beginnend in der Klausur am Montag wurden alle Anforderungen geklärt und der Gesamtaufbau entwickelt. Die weiteren Tage habe ich das System implementiert, getestet und dokumentiert. Das Ergebnis besteht aus dem abgegebene, lauffähige Programm mit Tests, der Entwicklerdokumentation und diesem Dokument.

Die Firma „Plan & Eben“ benötigt einen Routenplan für ihre Roboter, die Parkettflächen versiegeln können. Dafür muss eine unterbrechungsfreie Route über alle Parzellen (eine Parzelle ist 40cm x 40cm-Fläche als Teil der Gesamtfläche) gefunden werden, die möglichst viele Parzellen jeweils genau einmal abfährt. Durch die bereits in der Aufgabe vorgegebene Strategie und meine selbstständig entwickelte Strategie stehen der Firma zwei gleichwertige Routen zur Verfügung.

Mithilfe der funktionalen Trennung kann der bestehende Code leicht erweitert und wiederverwendet werden. Soll eine andere Eingabe verwendet werden (beispielweise über die Konsole) kann einfach von der Schnittstelle geerbt werden. Genauso ließe sich auch die Ausgabe leicht umlenken. Über das *Strategy-Pattern* können weitere Strategien ohne Änderungen des bestehenden Codes hinzugefügt werden. Nur wenn der Lösungsalgorithmus (Backtracking) geändert werden soll, kann dieser leider nicht auf dem gleichen Weg ersetzt werden. Es ist aber denkbar die Klasse *Controller* abzuleiten und die Methoden zu überschreiben.

Ich habe die Klassen *InputReader* und *OutputWriter* absichtlich als abstrakte Klassen deklariert, da dies meiner Meinung nach korrekter ist als ein Interface zu verwenden. Ein Interface sollte für Eigenschaften verwendet werden (ein Objekt ist z.B. *drawable*), während eine abstrakte Klasse einen Zweck erfüllt (z.B. das Einlesen). Falls es Probleme mit Mehrfachvererbung geben sollte, kann die neue Klasse dies über Objekte umgehen.

Der Backtrackingalgorithmus ist bei wachsender Größe der zu versiegelnden Fläche ineffizient und stellt daher eine Schwäche des Programms dar. Wie bereits beschrieben, kann der Worst-Case nicht vollständig durchgeführt werden. Hier kann allerdings durch weitere Maßnahmen bzw. Optimierungen die Laufzeit verbessert werden.

5.2 Ausblick

Das Problem der langen Laufzeit bekommt der Algorithmus, wenn es eine große Anzahl an möglichen Lösungen gibt und nur wenige oder gar keine vollständig sind. Eine denkbare Optimierung ist, dass vor dem Backtracking eine Analyse des Spielfeldes gemacht wird. Dadurch könnte z.B. herausgefunden werden, dass eine Parzelle niemals zu erreichen ist und deshalb in der Route nicht versiegelt werden muss. Der Algorithmus könnte dann früher abbrechen, nämlich wenn alle Parzellen, die auch wirklich erreichbar sind, versiegelt wurden.

Mögliche Erweiterungen für das Gesamtsystem wären:

- Ein- und Ausgabe über eine Benutzeroberfläche (GUI) zu implementieren. In Java stehen beispielsweise die *Swing-API* oder die *SWT-Bibliothek* zur Verfügung.

- Die Eingabe könnte in dafür vorgesehene Textfelder erfolgen. So kann schon an einer früheren Stelle die Eingabe validiert werden und der Benutzer würde direkt eine Fehlermeldung erhalten. Alternativ könnte man z.B. auch per *Drag&Drop* Hindernisse „auf das Feld ziehen“.
- Manuelle Eingabe der Anzahl nicht zu erreichender Parzellen, um den Algorithmus zu beschleunigen.
- Anzeigen eines Fortschrittsbalkens, der das Verhältnis der bereits geprüften Lösungen zu allen möglichen Lösungen angibt. Falls vorzeitig eine vollständige Lösung gefunden wird, ist der Benutzer eher erfreut, dass er nicht länger warten muss.

- Offensichtlich werden die beiden Strategien ja nacheinander ausgeführt und benötigen jeweils einen großen Teil der Laufzeit. Hier wäre es sinnvoll die zwei Strategien gleichzeitig in verschiedenen Threads zu starten, da sie komplett unabhängig von einander laufen. Dies würde auf Mehrkernprozessoren die Laufzeit für größere Probleme in etwa halbieren, da das Einlesen und Ausgeben nur wenige Millisekunden benötigt.

- Mit den genannten Optimierungen könnten auch größere Felder zugelassen werden. Alternativ kann ein großes Feld auch in mehrere kleine zerteilt werden.

- Es könnten auch unregelmäßige Felder zugelassen werden. Alternativ könnte man auch mit Hindernissen unregelmäßige Felder nachstellen.

- Weitere Algorithmen könnten hinzugefügt werden, die nicht immer die bestmögliche Lösung, aber eine gute Näherung in einem Bruchteil der Zeit, finden (eventuell *Greedy-Algorithmen*).

- Es könnte erlaubt werden, dass eine Route nicht zusammenhängen sein muss. Dann müsste der Algorithmus so erweitert werden, dass alle Parzellen mit möglichst wenig „Umsetzen des Roboters“ versiegelt werden.

6 Änderungen

6.1 Abweichungen

- Die Sichtbarkeit der Liste *warnings* der Klasse *InputReader* von *private* auf *protected* gesetzt, damit Unterklassen die Liste bearbeiten können.

- In der Klausur am Montag habe ich für eine Richtungsentscheidung manchmal den Begriff „direction“ und manchmal „decision“ verwendet. Ich habe dies zu „decision“ vereinheitlicht.

- Dem Konstruktor der Klasse *Area* habe ich im UML-Diagramm den Parameter *dimensions* der Typ *int[][]* übergeben. Dies war ein Schreibfehler und wurde zu *int[]* geändert.

- Die Methode *getWarnings(): List<String>* in der Klasse *InputReader* wurde in getWarning(): String geändert, die nur noch eine zusammengefasste Warnung ausgibt.

- Die Logik zur Erstellung der Routenliste wurde geändert. In der Klausur ging ich davon aus, dass ich eine Parzelle nur einfüge, wenn sich die Richtung geändert hat. Da es aber bei Rückschritten des Algorithmus schwierig ist, die Liste konsistent zu halten, habe ich mich entschlossen, jede abgelaufene Parzelle zu speichern bzw. immer die letzte zu entfernen. Für die Ausgabe werden aus der Liste nicht benötigte Parzellen entfernt. Daher benötigt die Methode addPointToRoute in der Klasse *Strategy* nicht mehr den Parameter *lastDecision: int*.

- Meine Strategie (Klasse *MyStrategy*) implementiert die Methode *getNextDecision(lastDecision: int, step: int)* nun so, wie ich es in der Aufgabenanalyse beschrieben habe. Die angegebene Umsetzung im Nassi-Shneidermann-Diagramm war nicht korrekt und wurde auch dort korrigiert.

- In der Methode backtrack(strategy: Strategy, point: Point, lastDecision: int) muss die Boolean-Variable „success“ direkt am Anfang initialisiert werden, statt in der for-Schleife wie es im Nassi-Shneidermann-Diagramm dargestellt wurde.

- In der Methode *backtrack(strategy: Strategy, point: Point, lastDecision: int)* muss die Boolean-Variable *success* auf *true* gesetzt werden, wenn sie es vorher nicht war und nun alle Parzellen versiegelt sind. Dies wurde im Nassi-Shneidermann-Diagramm am Ende ebenfalls falsch dargestellt und ebenfalls auch dort korrigiert.

6.2 Ergänzungen

- Ich habe das Worst-Case-Szenario und zwei weitere Sonderfälle definiert, die durch ungewöhnliche Eingaben oder besondere Zustände entstehen. Der Algorithmus wird jedoch immer normal ausgeführt.

- Methoden zu der Klasse *Area* hinzugefügt
  - *+ existsCell(p: Point): boolean*
  - *+ containsEndPoint(): boolean*

    - *+ numberOfCells(): int*
    - *– numberOfCellsWithValues(possibleValues: char[]): int*

- Methode zu der Klasse *Strategy* hinzugefügt
    - *+ getNextPointWithDecision(currentPoint: Point, decision: int): Point*

- Methode zu der Klasse *OutputWriter* hinzugefügt
    - *# formatRoute(route: List<Point>): String*

- Methode zu der Klasse *OutputFileWriter* hinzugefügt
    - *– setFile(file: File): void*

- Methode zu der Klasse *Controller* hinzugefügt
    - *– isValidPoint(point: Point, area Area): boolean*

7 Benutzeranleitung

7.1 Verzeichnisstruktur der gepackten Datei

In der gepackten Datei sind folgende Verzeichnisse enthalten:

Wurzelverzeichnis

- Ausführbares Programm.
- Die Beschreibung als Master-Datei im Microsoft Word-Format und als PDF.
- „*bin*“ um die *.class-Dateien und das Manifest abzulegen.
- „*diagrams*“ enthält die verschiedenen Diagrammtypen jeweils als Orginaldateien und Bilddateien im PNG-Format.
- „*documentation*“ enthält die Entwicklerdokumentation.
- „*scripts*“ enthält Batch-Skripte für das
    - Kompilieren und erzeugen des ausführbaren Programms.
    - automatische Durchführen aller Tests.
    - Aufräumen bzw. Löschen der erzeugten Dateien.
- „*src*“ enthält den Quellcode.
- „*tests*“ enthält die Eingabe- und Ausgabedateien der Tests.

7.2 Systemvoraussetzungen

Um das Programm auszuführen zu können, muss das *Java Software-Development-Kit (SDK)* in der Version 1.7 oder höher installiert sein. Die aktuelle Java-Version kann unter folgendem Link heruntergeladen werden:
http://java.com/de/download/index.jsp

Von Oracle angegebene Systemvoraussetzungen sind:[2]

- Windows 8 Desktop
- Windows 7
- Windows Vista SP2
- Windows XP SP3 (32-Bit); Windows XP SP2 (64-Bit)
- Windows Server 2008

Ein Pentium 2 266 MHz oder schnellerer Prozessor mit einem physikalischen RAM von mindestens 128 MB wird empfohlen. Außerdem benötigen Sie mindestens 124 MB freien Speicherplatz auf dem Datenträger.

7.3 Installation

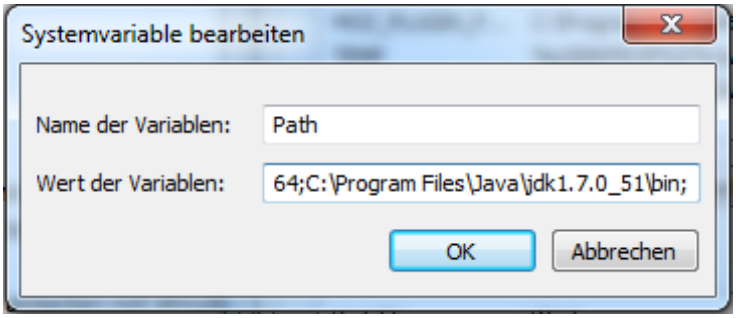
Der gesamte Inhalt der gepackten Datei kann in ein beliebiges Verzeichnis kopiert werden. Wichtig ist, dass das Zielverzeichnis beschrieben werden darf (Schreibzugriff erlaubt).

[2] Siehe http://java.com/de/download/win_sysreq-sm.jsp (Stand 16.05.2014)

Um das Programm unter Windows ausführen zu können muss in der Systemvariable „*Path*“ der Pfad in das „*bin*“-Verzeichnis der Java-SDK-Installation gesetzt sein.

Die folgende Beschreibung zum Setzen der Variable arbeitet mit Windows 7 Professional:

1. Öffnen des Startmenüs
2. Rechtsklick auf „Computer“, dann Linksklick auf „Eigenschaften“
3. Über das linke Menü „Erweiterte Systemeinstellungen“ öffnen
4. In dem neuen Fenster auf „Umgebungsvariablen“ klicken
5. Bei „Systemvariablen“ die Variable „*Path*“ auswählen und auf „Bearbeiten…“ klicken
6. Bei „Wert der Variablen“ den Pfad zur Java-Installation mit dem Unterordner „bin“ angeben.
   In meinem Beispiel wäre das:
   C:\Program Files\Java\jdk1.7.0_51\bin;
   Dabei muss auf die Semikola vor und nach dem Pfad geachtet werden, da diese als Trennzeichen dienen.
7. Mit Klick auf „OK“ bestätigen und ggf. die Konsole erneut öffnen.



7.4 Ausführen der Skripte

Nachdem die Systemvariable korrekt gesetzt wurde, können die Batch-Skripte aus dem Unterordner „*scripts*“ direkt ausführt werden. Dies kann entweder über die Konsole geschehen, indem *<dateiname>.bat* aufgerufen wird oder mit einem Doppelklick im Dateisystem auf die Datei. Es werden keine Übergabeparameter erwartet oder ausgewertet.

7.4.1 clear.bat

Das Script iteriert über alle erzeugten Ausgabedateien und löscht diese.

7.4.2 compile.bat

Kompiliert das Java-Programm mithilfe des Konsolenbefehls „*javac*“ und erzeugt daraus die ausführbare Datei *ParquetRobot.jar*. Dies ist bei Änderungen am Quellcode erforderlich, um anschließend mit dem Script „run.bat“ automatisiert alle Tests auszuführen.

7.4.3 run.bat

Führt das Programm mithilfe des Konsolenbefehls „*java -jar ParquetRobot.jar*“ für alle Eingabedateien automatisiert im Verzeichnis „*test/input*“ aus, welches dann die entsprechenden Outputdateien generiert.

8 Entwicklerdokumentation

Die Entwicklerdokumentation befindet sich in Form einer vollständigen JavaDoc in dem Unterordner „documentation“. Es sind Methoden und Attribute aller Sichtbarkeitsstufen (public, protected, package, private) inkludiert. Das Programm JavaDoc erzeugt aus den entsprechenden Quellcode-Kommentaren eine vollständige Entwicklerübersicht als HTML-Dateien. Diese ist wie eine Webseite benutzbar. Durch das Öffnen der Datei „index.html“ wird die Übersicht geladen, von der man zu allen weiteren Paketen und Klassen gelangt.

JavaDoc ist ein standardisiertes Verfahren zur Dokumentation von Java-Quelltext. Vor dem zu dokumentierenden Code wird mit den Zeichen */** der Beginn eines JavaDoc-Kommentars eingeleitet und mit den Zeichen **/* beendet. Dies lehnt sich an mehrzeilige Kommentare an, die allerdings nicht von JavaDoc interpretiert werden.

Es wird für jede Klasse eine Beschreibung angegeben, die den (eindeutigen) Zweck der Klasse erklärt. Auch kann der Autor angegeben werden und bei Bedarf die Version des Programms.

Methoden werden ebenfalls mit einer Beschreibung des Zwecks versehen, doch gibt es noch weitere Schlüsselwörter: mit *@param* werden Übergabeparameter näher erklärt und mit *@throws* sollte der Entwickler *checked* und *unchecked Exceptions* dokumentieren. Ein Beispiel dafür ist das Parsen eines *Integers* mit der statischen Methode *Interger.parseInt(String s)*. Falls aus dem übergebenen String kein *Integer* geparsed werden kann, wird in dem Fall eine *FormatException* geworfen, die gefangen werden kann.

Es gibt noch einige weitere Schlüsselwörter, die aber nicht so häufig verwendet und deshalb hier nicht erklärt werden. Weitere Informationen sind bei Wikipedia unter folgendem Link zu finden: https://de.wikipedia.org/wiki/Javadoc#.C3.9Cbersicht_der_Javadoc-Tags

9 Entwicklungsumgebung

Das gesamte Softwaresystem wurde unter folgendem Computer-System entwickelt und getestet:

Programmiersprache	Java 1.7 Update 51
Rechner	Intel® Pentium® CPU P6200 2.13 GHz 2.13 GHz 4 GB Arbeitsspeicher
Betriebssystem	Windows 7 Professional 64 Bit-Betriebssystem

10 Eigenhändigkeitserklärung

Ich erkläre verbindlich, dass das vorliegende Prüfprodukt von mir selbstständig erstellt wurde. Die als Arbeitshilfe genutzten Unterlagen sind in der Arbeit vollständig aufgeführt. Ich versichere, dass der vorgelegte Ausdruck mit dem Inhalt des von mir erstellten Datenträgers identisch ist. Weder ganz noch in Teilen wurde die Arbeit bereits als Prüfungsleistung vorgelegt. Mir ist bewusst, dass jedes Zuwiderhandeln als Täuschungsversuch zu gelten hat, der die Anerkennung des Prüfprodukts als Prüfungsleistung ausschließt.

Im Rahmen des auftragsbezogenen Fachgesprächs sind die Aufgabenanalyse und der Lösungsentwurf zu begründen und das Prüfungsprodukt zu erläutern.

Ort und Datum	Unterschrift des Prüfungsteilnehmers

11 Verwendete Hilfsmittel

- Eclipse Standard/SDK (32 Bit)
  Version: Kepler Service Release 1
  Entwicklungsumgebung für Java und andere Programmiersprachen.
  http://eclipse.org/
- Notepad++
  Open-Source-Texteditor für Windows
  http://notepad-plus-plus.org
- yEd
  Plattform-unabhängiger Graph-Editor. Unter anderem Fähig UML-Diagramme zu erstellen.
  http://www.yworks.com/de/products_yed_about.html
- Structorizer
  Plattform unabhängiges Programm zur Erzeugung von Nassi-Shneidermann-Diagrammen.
  http://structorizer.fisch.lu/
- Quick Sequenz Diagram Editor
  Plattform-unabhängiges Programm zur Erzeugung von Sequenzdiagrammen.
  http://sdedit.sourceforge.net/index.html
- GIMP
  Plattform-unabhängiges Programm zur Grafikbearbeitung.
  http://www.gimp.org/