IHK Abschlussprüfung Sommer 2014

Entwicklung eines Softwaresystems

Dokumentation

Student:
Fabian Braun
Prüflingsnummer 101 20510

Ausbildender Betrieb:
Atos Worldline GmbH
Pascalstr. 19
52076 Aachen

Inhaltsverzeichnis

1 Dokumentinformation

Dieses Dokument enthält die in der Aufgabenstellung geforderten Prüfungsproduktteile Verbale Beschreibung des realisierten Verfahrens, Programmsystem, Entwicklerdokumentation, Benutzeranleitung, Testdokumentation, Zusammenfassung und Ausblick.

2 Einleitung

Das Programm 20510_solver wurde im Rahmen der IHK Prüfung „Entwicklung eines Softwaresystems" entwickelt.
Es dient dazu einen sogenannten „Hamiltonweg" durch einen Graphen zu finden. Die praktische Anwendung, für die dieses theoretische Problem gelöst werden soll, besteht in der iterativen Versiegelung einer Parkettfläche mit Hindernissen. Ein Roboter soll eine Parkettfläche versiegeln, wobei er an einem definierten Punkt startet und die Fläche nach und nach überquert. Dabei darf er nicht über bereits versiegelte Flächen fahren. Für die genauere Beschreibung des Problems siehe Kapitel Aufgabenanalyse.

2.1 Eigenhändigkeitserklärung

Ich erkläre hiermit 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.

Fabian Braun
Aachen, den 16.05.2014

3 Benutzeranleitung

3.1 Systemanforderungen

Das Programm 20510_solver.jar wurde unter Windows 7, 64-Bit und Java(TM) SE Runtime Environment 1.7.0 Update 51 getestet. Es ist jedoch grundsätzlich plattformunabhängig. Die beigefügten Batch-Dateien sind nur auf Windows-Betriebssystemen ausführbar.

3.2 Installation

Das gelieferte ZIP-Archiv muss entpackt werden. Für den entstehenden Ordner, sowie alle Unterordner müssen Lese- und Schreibrechte gegeben sein. Auf dem System muss ein Java Runtime Environment ab Version 7 Update 51 installiert sein.

3.2.1 Konfiguration

Es ist möglich den Detailgrad der Ausgabe des Programms zu konfigurieren. Hierzu dient die Datei configuration.properties, die auf der Ordnerebene der 20510_solver.jar-Datei liegen muss.
Dort kann über den Wert der property „debug", das detaillierte Logging eingeschaltet werden (durch Setzen des Werts auf „true").
Außerdem kann die Reihenfolge konfiguriert werden, in der die verschiedenen Lösungsstrategien auf das Problem angewendet werden. Hierzu dient die property „switchAlgorithms", die die Default-Reihenfolge umkehrt, wenn ihr Wert auf „true" gesetzt wird.

3.3 Programmaufruf

Zum komfortablen Programmaufruf wurden batch-Dateien erstellt, die im Unterordner ./test liegen.
Ansonsten wird das Programm wie folgt von einer Windows Shell aufgerufen:

```
$ java –jar 20510_solver.jar [.]endingInputFiles pathToFileOrFolder
[[pathToFileOrFolder2] [pathToFileOrFolder3] ...]
```

Mit dem ersten Übergabeparameter wird die Endung der Eingabedateien definiert. Der zweite bis n-te Parameter dient zur Spezifizierung von Ordnern, in denen Eingabedateien (mit der entsprechenden Endung) gesucht werden sollen. Es ist auch möglich hier explizit weitere Dateien anzugeben, die als Eingabedateien betrachtet werden.
Bsp.:

```
$ java -jar 20510_solver.jar in ..\test C:\data\specialtest.special
```

(Hier wird ein Ordner, und eine zusätzliche Datei angegeben)

Die Ausgabedateien werden jeweils in den Ordner geschrieben, in dem die Eingabedatei liegt. Der Name der Eingabedatei wird übernommen und um die Endung „.out" ergänzt.

Anmerkung:
Die Ausgabedateien werden jedes Mal geschrieben, wenn eine bessere Problemlösung ermittelt werden konnte. Dies kann zu folgendem (erwünschten) Effekt führen:
Wenn das System von außen abgebrochen wird (Steuerung + C auf der Windows Shell), kann es sein, dass erst zu einem Algorithmus eine Lösung ermittelt werden konnte, und dass die Ausgabedatei dementsprechend nur eine Lösung enthält.

3.4 Ausführung der Testfälle

Die Testfall-Eingabedateien befinden sich im Ordner ./test und sind unterteilt in IHK-Testfälle (./test/ihk), eigene OK-Testfälle (./test/ok), sowie eigene Not-OK-Testfälle (./test/nok).
In diese Ordner werden ebenfalls die Ausgabedateien geschrieben. Die Ausgabedateien haben die Endung .out und können mit herkömmlichen Texteditoren gelesen werden (z.B. Windows 7 Editor, Notepad++).

3.4.1 Vorbereitete Batch-Dateien:

- run_all.bat - führt alle Testfälle in den Ordnern ok, ok/performance, nok und ihk aus
- run_ok.bat - führt alle Testfälle im Ordner ok und ok/performance aus
- run_nok.bat - führt alle Testfälle im Ordner nok aus
- run_ihk.bat - führt alle Testfälle im Ordner ihk aus
- run_custom.bat - erlaubt die Eingabe eines gewünschten Ordners sowie der Dateiendung der auszuführenden Testfälle

3.4.2 Performance-kritische Testfälle:

Im Ordner ./test/ok befinden sich zwei weitere Ordner, die Testfälle enthalten, deren Laufzeit länger als 10 Sekunden beträgt. Werden diese Testfälle ausgeführt, so sollte vorher das detaillierte Logging in der Konfiguration deaktiviert werden.

3.4.2.1 Ordner performance

Die Testfälle im Unterordner „performance" dauern zwar spürbar länger als die automatisierten Testfälle, terminieren jedoch auch in insgesamt weniger als einer Minute. In diesem Ordner befinden sich Ausgabe-Dateien, die zusätzlich mit der Endung „.backup" versehen wurden, um dem Überschreiben vorzubeugen.

3.4.2.2 Ordner performance-critical

Im Gegensatz dazu terminieren die Testfälle im Ordner „performance_critical" nicht in absehbarer Zeit, selbst wenn sie einzeln ausgeführt werden. Für diese Testfälle wurde

keine Batch-Dateien erstellt, da sie besser einzeln und manuell ausgeführt werden sollten, statt automatisiert. Unter Umständen muss das System bei der Bearbeitung von außen unterbrochen werden, da in naher Zukunft kein Ende abzusehen ist. Um Näherungslösungen für jeden Algorithmus zu erhalten kann die Ausführungsreihenfolge der Algorithmen konfiguriert werden. In diesem Ordner befinden sich Ausgabe-Dateien, die zusätzlich mit der Endung „.<algorithm>_first" versehen wurden, um dem Überschreiben vorzubeugen. Diese enthalten jeweils eine Näherungslösung zu einem Algorithmus, die nach einer Minute vorlag.

3.5 Mögliche Fehlermeldungen

Fehlermeldung	**Ursache**
File does not contain at least 2 lines (not counting comments): …	Die Eingabedatei enthält weniger als zwei nicht-Kommentar-Zeilen. Es sind jedoch mindestens zwei notwendig (Eine Zeile für die Parkettmaße und eine Zeile für die Startposition).
Input File cannot be read: …	Es fehlen Leseberechtigungen für die angegebene Datei / den angegebenen Ordner.
Input File cannot be found: …	Die angegebene Eingabedatei konnte nicht gefunden werden.
Invalid format for area measures: '…' format must be 'number number' and number in [1..10]	Die Parkettmaße haben nicht das korrekte Format, oder die angegebenen Zahlen liegen nicht im Intervall [1..10]
Invalid format for obstacle coordinates: '…' format must be 'number number number number' and number in [1..10]	Ein Hindernis wurde nicht korrekt formatiert angegeben, oder die angegebenen Zahlen liegen nicht im Intervall [1..10]
Invalid format for start coordinates: '…' format must be 'number number' and number in [1..10]	Der Startpunkt wurde nicht korrekt formatiert angegeben, oder die angegebenen Zahlen liegen nicht im Intervall [1..10]
obstacle coordinate not in area: y=…	Die y-Koordinate eines Hindernis liegt nicht im Bereich der Parkettfläche
obstacle coordinate not in area: x=…	Die x-Koordinate eines Hindernis liegt nicht im Bereich der Parkettfläche
start coordinate not in area: y=…	Die y-Koordinate der Startparzelle liegt nicht im Bereich der Parkettfläche
start coordinate not in area: x=…	Die x-Koordinate der Startparzelle liegt nicht im Bereich der Parkettfläche
cannot place start on obstacle: y=… x=…	Die Position (y,x) des Startfelds und eines Hindernis stimmen überein, was nicht erlaubt ist.
cannot place obstacle on start: y=… x=…	Die Position (y,x) des Startfelds und eines Hindernis stimmen überein, was nicht erlaubt ist.
Folgende Fehlermeldungen werden nicht in eine Ausgabedatei geschrieben, sondern ausschließlich auf die Konsole	
File or dir could not be found: …	Beim Programmaufruf wurde ein Pfad übergeben, der auf dem System nicht existiert.

config file … cannot be read. Default value '…' assumed for property '…'	Die Konfigurationsdatei für das Programm konnte nicht gefunden werden, es wurde der Defaultwert angewendet. Die Datei sollte im selben Ordner liegen wie die ausführbare .jar-Datei und den Namen „configuration.properties“ haben.
cannot write into output file … file will not be created	Es bestehen keine Schreibrechte für die Ausgabedatei. Diese wird daher nicht erstellt.
Please execute the program via 'java -jar 20510_solver.jar [.]endingInputFiles pathToFileOrFolder [[pathToFileOrFolder2] [pathToFileOrFolder3] ...]'	Das Programm wurde nicht mit den benötigten Parametern gestartet. Siehe auch Programmaufruf

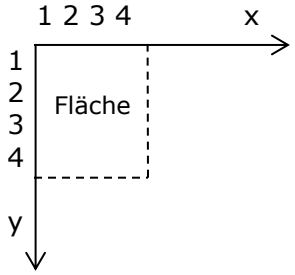
4 Theoretischer Hintergrund

4.1 Aufgabenanalyse

(IHK, Sommer 2014)
Es ist ein Softwaresystem zu entwickeln, das einen möglichen Weg über eine rechteckige Fläche mit Hindernissen ermittelt.
Die Fläche ist in quadratische Parzellen gleicher Größe unterteilt (40 cm x 40 cm) und die Hindernisse entsprechen immer einer Menge von Parzellen, die nicht überquert werden dürfen.
Der zu ermittelnde Weg besitzt einen definierten Startpunkt (bzw. Startparzelle), wohingegen der Endpunkt frei gewählt werden kann. Es gilt die Einschränkung, dass bereits durchquerte Parzellen nicht erneut betreten werden dürfen, bzw. dass kein Wegabschnitt mehrfach durchlaufen wird.
Das Überspringen von Parzellen ist nicht erlaubt und es gelten ausschließlich die Bewegungsrichtungen:

- Im Plan eine Parzelle nach oben.
- Im Plan eine Parzelle nach rechts.
- Im Plan eine Parzelle nach unten.
- Im Plan eine Parzelle nach links.

In der Aufgabenstellung wird dieses „Weg-Finde-Problem" auf einen Roboter bezogen, der eine Parkettfläche nach und nach versiegeln soll. Jede Parzelle ist eindeutig durch ihre x- und y-Koordinate in der Parkettfläche definiert. Der Weg des Roboters über die Fläche wird in einem Routenplan festgehalten. Darin wird die Anfangs- bzw. Zielparzelle gespeichert, sowie alle Parzellen auf dem Weg, an denen eine Richtungsänderung stattgefunden hat.

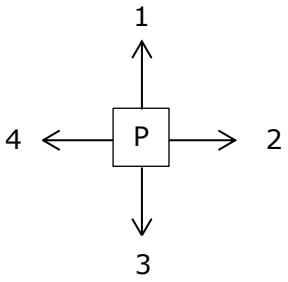


Inhaltliche Restriktionen:

- Die Fläche ist weder in x- noch in y-Richtung länger als 10 Parzellen.
- Aufgrund der Hindernisse ist es möglich, dass kein Weg existiert auf dem alle Parzellen erreicht werden können. In diesem Fall soll das System den Weg ausgeben, bei dem die meisten Parzellen versiegelt wurden.

Wegfindestrategien:
Das Softwaresystem soll zwei verschiedene Wegfindestrategien anwenden.

Die 1. Strategie ist durch die Aufgabenstellung vorgegeben und heißt Uhrzeiger-Strategie. In der Uhrzeiger-Strategie wird die als nächste zu befahrende Parzelle nach dem folgenden Diagramm ermittelt:



Von der Position P aus wird zuerst versucht nach oben zu gehen. Ist dies nicht möglich (aufgrund eines Hindernisses oder des Randes), so wird versucht nacheinander in die Richtungen rechts, unten und links zu gehen.

Auf diese Art und Weise kann es dazu kommen, dass ein ungünstiger Weg gewählt wird, der in einer Sackgasse endet. Falls dies geschieht und noch nicht alle Parzellen erreicht wurden, soll das System zur letzten Wegalternative zurücklaufen und diese ausprobieren, sodass ein möglicher Weg auch immer gefunden wird. Dieses Verfahren entspricht der Tiefensuche in einem Graphen.

Die 2. Strategie ist im Rahmen der Aufgabenstellung zu entwickeln und wird im Kapitel Verfahrensbeschreibung erläutert.

4.1.1 Eingabe:

Die zu bearbeitende Fläche wird dem Softwaresystem durch eine Eingabedatei übermittelt.

Die Eingabedatei hat dabei folgendes Format:

1. Zeile: `Ganzzahl aus [1..10] + „ “ + Ganzzahl aus [1..10]`
   - ➔ Abmessung des Parketts
2. Zeile: `Ganzzahl aus [1..10] + „ “ + Ganzzahl aus [1..10]`
   - ➔ Startposition des Roboters
   - ➔ Hier muss überprüft werden, ob die Startposition innerhalb der Abmessungen aus Zeile 2 liegt

3-n.Zeile: `Ganzzahl aus [1..10] + „ “ + Ganzzahl aus [1..10] + „ “ + Ganzzahl aus [1..10] + „ “ + Ganzzahl aus [1..10]`
- ➔ Angabe der Hindernisbereiche durch die Koordinaten y1, x1, y2, x2
- ➔ Auch hier muss geprüft werden, dass alle Koordinaten innerhalb der Fläche liegen
- ➔ Zusätzlich darf kein Hindernis auf dem Startfeld platziert werden.

Leerzeichen und Tabulatoren am Anfang und Ende einer Zeile werden grundsätzlich ignoriert.
Die obige Indizierung der Zeilen bezieht sich auf eine Datei ohne Kommentarzeilen. Alle Zeilen, die mit einem Semikolon beginnen werden ignoriert. Dies gilt ebenfalls für Leerzeilen.
Anmerkung:
Es kann vorkommen, dass sich Hindernisse vollständig oder teilweise überschneiden. Dies ist explizit nicht verboten.

4.1.2 Ausgabe:

Die Ausgabe der ermittelten Lösung soll in eine Datei erfolgen.

Darin wird zunächst der Ausgangszustand dokumentiert.
In der ersten Zeile wird die Startposition des Roboters ausgegeben:
`„Startposition (y1, x1)“`
Anschließend wird der Anfangszustand der Fläche dokumentiert. Dazu wird ein Koordinatensystem ausgegeben, in dem das Startfeld mit „S“ und die Hindernis-Felder

mit „H“ repräsentiert werden. Alle anderen Parzellen werden durch Leerzeichen repräsentiert. Die Indizierung der Spalten und Zeilen startet bei 1.
Anschließend folgen die Lösungen, die die Uhrzeiger-Strategie bzw. die eigene Strategie ermittelt haben:
Zunächst wird der Name der entsprechenden Strategie ausgeben. Anschließend die Fläche und der Weg, den der Roboter darauf zurückgelegt hat. Hierbei gelten folgende Zeichen:

„ “	Parzelle wurde nicht befahren
„H“	Hindernis in dieser Parzelle
„Z“	Position des Roboters nach Zurücklegen des Weges
„>“	Von dieser Position hat der Roboter sich nach rechts bewegt
„V“	Von dieser Position hat der Roboter sich nach unten bewegt
„<“	Von dieser Position hat der Roboter sich nach links bewegt
„^“	Von dieser Position hat der Roboter sich nach oben bewegt

Nach der Ausgabe des Koordinatensystems soll der Routenplan wie folgt ausgegeben werden:
Eine Zeile: „Routenplan:“
Nächste Zeile: y1,x1 / y2,x2 / y3,x3 / y4,x4 / …
An dieser Stelle ist die Strategie-spezifische Ausgabe beendet und es werden abschließend verschiedene Metriken ausgegeben:

1. „Zu versiegelnde Parzellen: p1“
2. „Hindernisparzellen: h1“
3. „Versiegelte Parzellen: p2“
4. „Nicht versiegelte Parzellen: p3“

Anmerkung:
Falls die Parkettfläche lediglich aus einer Parzelle besteht, so wird diese in der Ausgabe mit „S“ repräsentiert.

4.1.3 Teilaufgaben

Insgesamt soll das Softwaresystem folgende Teilaufgaben lösen:

1. Validierung der Eingabedatei
2. Überführung der Eingabedatei in eine geeignete Datenstruktur
3. Ermittlung der Lösung unter Nutzung der Uhrzeiger-Strategie
4. Ermittlung der Lösung unter Nutzung der eigenen Strategie
5. Ausgabe der ermittelten Lösungen

4.2 Verfahrensbeschreibung

4.2.1 Zweite Strategie

Die selbst zu entwickelnde Strategie wird nachfolgend „Greedy-Strategie" genannt und funktioniert wie folgt:

Der Roboter prüft zunächst, welche umliegenden Parzellen er anfahren kann und für jede von ihnen, wie viele Parzellen er von dort aus momentan anfahren könnte. Anschließend wählt er den direkten Nachbarn, der selbst am wenigsten freie Nachbarn (Freiheitsgrade) hat.

Beispiel: Roboter an Position P:

	1	2	3	4	5
1					
2			N_1	N_{21}	
3		N_4	P	N_2	H
4			N_3	N_{22}	
5					

Hier kann der Roboter alle Nachbarn N_1, N_2, N_3, N_4 befahren. Es wird jedoch der Nachbar N_2 ausgewählt, da er die geringste Anzahl an befahrbaren direkter Nachbarn hat (nämlich 2): N_{21} und N_{22}

Diese Strategie führt dazu, dass sich der Roboter an vorhandenen Rändern und Hindernissen entlang bewegt, anstatt in freie Flächen hineinzudrängen und so gegebenenfalls leere „Inseln" zu produzieren:

Beispiel für leere Insel mit Uhrzeiger-Strategie:
(nur bis zum Erreichen der 1. Sackgasse)

	1	2	3	4
1			>	V
2			^	V
3			^	V
4			^	Z

= Leere Insel

Bei der Nutzung der Greedy-Strategie kann es vorkommen, dass mehrere Nachbarn dieselbe Anzahl Freiheitsgrade besitzen. In diesem Fall wird unter diesen Nachbarn nach der Uhrzeiger-Strategie ausgewählt (Fallback-Strategie).

Obiges Insel-Beispiel mit Greedy-Strategie:

	1	2	3	4
1	>	V	V	<
2	^	Z	V	^
3	^	V	<	^
4	^	<	>	^

Hier würde gleich der 1. Versuch zu einem optimalen Weg führen.

Auch diese Strategie zur Auswahl des nächsten Feldes findet nicht immer im 1. Versuch einen optimalen Weg. Daher wird diese Art der Priorisierung ebenfalls mit einer Tiefensuche gekoppelt, die wie in der Aufgabenstellung beschrieben Alternativen sucht, bis ein optimaler Weg gefunden wird.

4.2.2 Logische Datenstruktur

Die Fläche der Parzellen wird intern als Graph abgespeichert, wobei jeder Knoten einer Parzelle entspricht. Die Knoten zeigen jeweils auf ihre direkten Nachbarn. Insbesondere diese Datenstruktur wird bei der Tiefensuche ausgenutzt.
Alle Parzellen werden jedoch auch in einem zwei-dimensionalen Feld gespeichert, um unter anderem eine effizientere Ein- und Ausgabe zu ermöglichen. Dieses Feld wird nach dem objektorientierten Prinzip der Datenkapselung in eine eigene Klasse ausgelagert.
Zur Speicherung des aktuell durchlaufenen Pfads wird zudem eine Liste von Parzellen bereitgestellt. Die Lösungen zu einem Algorithmus werden ebenfalls als Liste von Parzellen vorgehalten, wobei diese Liste in einer Schlüssel-Wertepaar-Tabelle zum entsprechenden Algorithmus hinterlegt wird.
Um die verschiedenen Weg-Finde-Algorithmen umzusetzen, wird das Strategie-Pattern eingesetzt.
Es wird demnach zwei Klassen geben, die eine gemeinsame Schnittstelle ausprägen. Die gemeinsame Schnittstelle enthält eine Funktion, die zu einer gegebenen Zelle die nächste Zelle ermittelt. Dieser Funktion kann außerdem eine Liste von Richtungen übergeben werden, die bei der Auswahl der nächsten Zelle ignoriert wird.
Die Tiefensuche, die jeweils einen dieser Algorithmen einsetzt, wird so realisiert, dass der Algorithmus generisch ausgetauscht werden kann.

4.2.3 Hauptalgorithmus

Der Hauptalgorithmus (Tiefensuche) nutzt Backtracking um einen optimalen Weg zu ermitteln:
Hier wird ein Beispiel mit der Greedy-Strategie erläutert:

	1	2	3	4
1	S			
2			H	H

Prüfe zunächst die Freiheitsgrade der direkten Nachbarn (2, 1) und (1, 2). Die Freiheitsgrade sind jeweils 2, also wird der Nachbar mit der Uhrzeiger-Strategie als Fallback bestimmt: rechts kommt vor unten -> nächste Position (1, 2)

	1	2	3	4
1	>	P		
2			H	H

Freiheitsgrade der nächsten Nachbarn:
(1, 3) = 1
(2, 2) = 1
⇨ Auswahl des rechten Nachbarn (1, 3)

	1	2	3	4
1	>	>	P	
2			H	H

Es gibt nur noch einen möglichen Nachbarn: (1, 4)

	1	2	3	4
1	>	>	>	P
2			H	H

Sackgasse erreicht
⇨ Speichere diesen Weg als momentan beste Lösung
Mache alle Schritte rückgängig bis ein Alternativweg möglich ist:

	1	2	3	4
1	>	P		
2			H	H

Wähle nun die einzige Alternative: (2, 2)

	1	2	3	4
1	>	V		
2		P	H	H

Nächster Schritt: (2, 1)

	1	2	3	4
1	>	V		
2	P	<	H	H

Diese Lösung ist nicht besser als die bereits gefundene, also speichere sie nicht.

Mache alle Schritte rückgängig bis ein Alternativweg möglich ist:

	1	2	3	4
1	P			
2			H	H

Wähle nun die einzige Alternative: (2, 1)

	1	2	3	4
1	V			
2	P		H	H

	1	2	3	4
1	V	>	>	P
2	>	^	H	H

Diese Lösung ist besser als die bisherige und zudem optimal, da alle Parzellen erreicht wurden.
Speichere diese Lösung als die beste ab und brich den Algorithmus ab, da keine bessere Lösung mehr gefunden werden kann.
Anmerkung:
Der Algorithmus wird rekursiv implementiert. Auf jeder Rekursionstiefe wird die Strategie zur Bestimmung des nächsten Knotens viermal aufgerufen, wobei nach jedem Mal ein weiterer Rekursionsaufruf folgt.
Bei der iterativen Bestimmung des nächsten Knotens werden jedes Mal die vorher ermittelten Knoten als zu ignorierende mitgegeben.
Dementsprechend wird zunächst der erste Knoten ermittelt und anschließend der am zweithöchsten priorisierte, indem der erste als ignorierter Knoten übergeben wird.
Um Speicherplatz zu sparen, wird die gesamte Zeit mit einem Pfad-Objekt (Liste) gearbeitet. Dies hat allerdings zur Folge, dass Änderungen am Pfad nach einem rekursiven Aufruf wieder rückgängig gemacht werden müssen.
Damit eine gefundene Lösung nicht im Anschluss verändert wird, muss der Pfad, der eine neue Lösung repräsentiert kopiert und in einem anderen Objekt gespeichert werden.

4.2.4 Eingabevalidierung und Fehlerbehandlung:

Das korrekte Format der Eingabe wird mit Hilfe von regulären Ausdrücken validiert.
Außerdem werden Bereichsprüfungen eingesetzt um beispielsweise zu erkennen, wenn ein Hindernis auf dem Startfeld platziert werden soll.
Die Fehlerbehandlung wird Ausnahme-basiert umgesetzt.
Tritt ein Fehler auf, so wird die entsprechende Fehlermeldung auf der Konsole ausgegeben. Zusätzlich wird die Meldung in die Ausgabedatei geschrieben, falls möglich (es ist beispielsweise nicht möglich, wenn kein Dateipfad bekannt ist).
Um Ausgaben auf die Konsole zu kapseln wird ein Singleton verwendet, das Funktionen bereitstellt, um auf die Konsole zu schreiben.

4.2.5 Module

Insgesamt lässt sich das Softwaresystem in folgende Module unterteilen:

start	Enthält globalen Startpunkt des Systems
io	Ein- und Ausgabe, Konsolenausgabe
control	Steuert den Ablauf, führt die Algorithmen durch
model	Enthält Klassen für die Datenhaltung
error	Enthält Ausnahme-Klassen

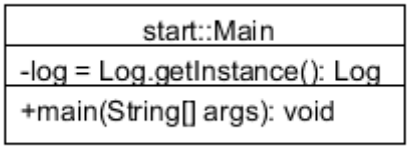
4.3 Programmkonzeption

Das Softwaresystem wird in Java entwickelt und die nachfolgenden Diagramme beziehen sich auf diese Programmiersprache.

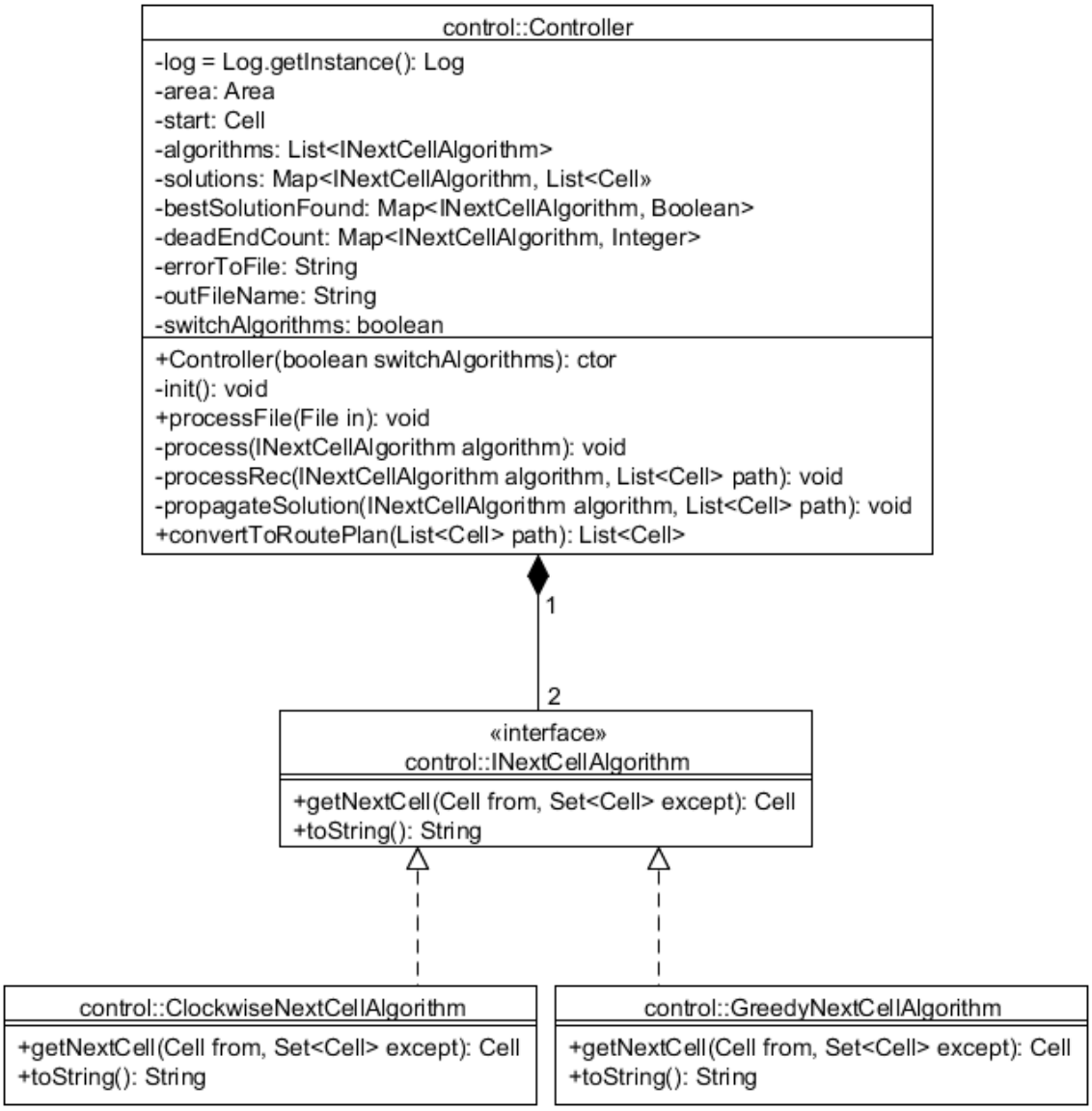
4.3.1 Datenstrukturen

4.3.1.1 Klassendiagramme unterteilt nach Modulen / packages

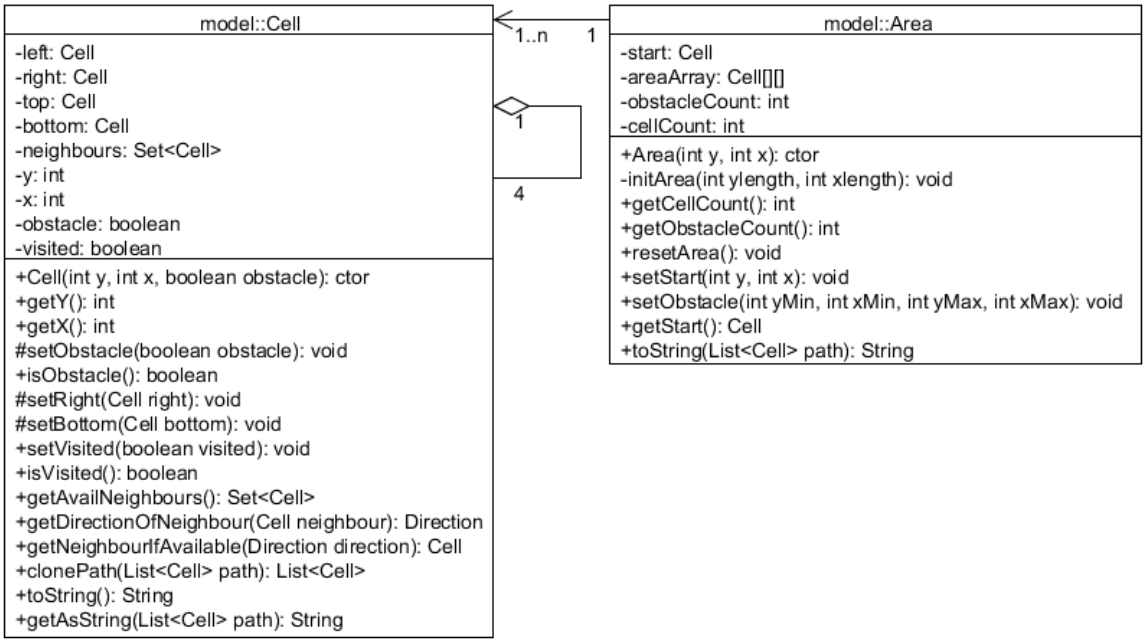
Modul start



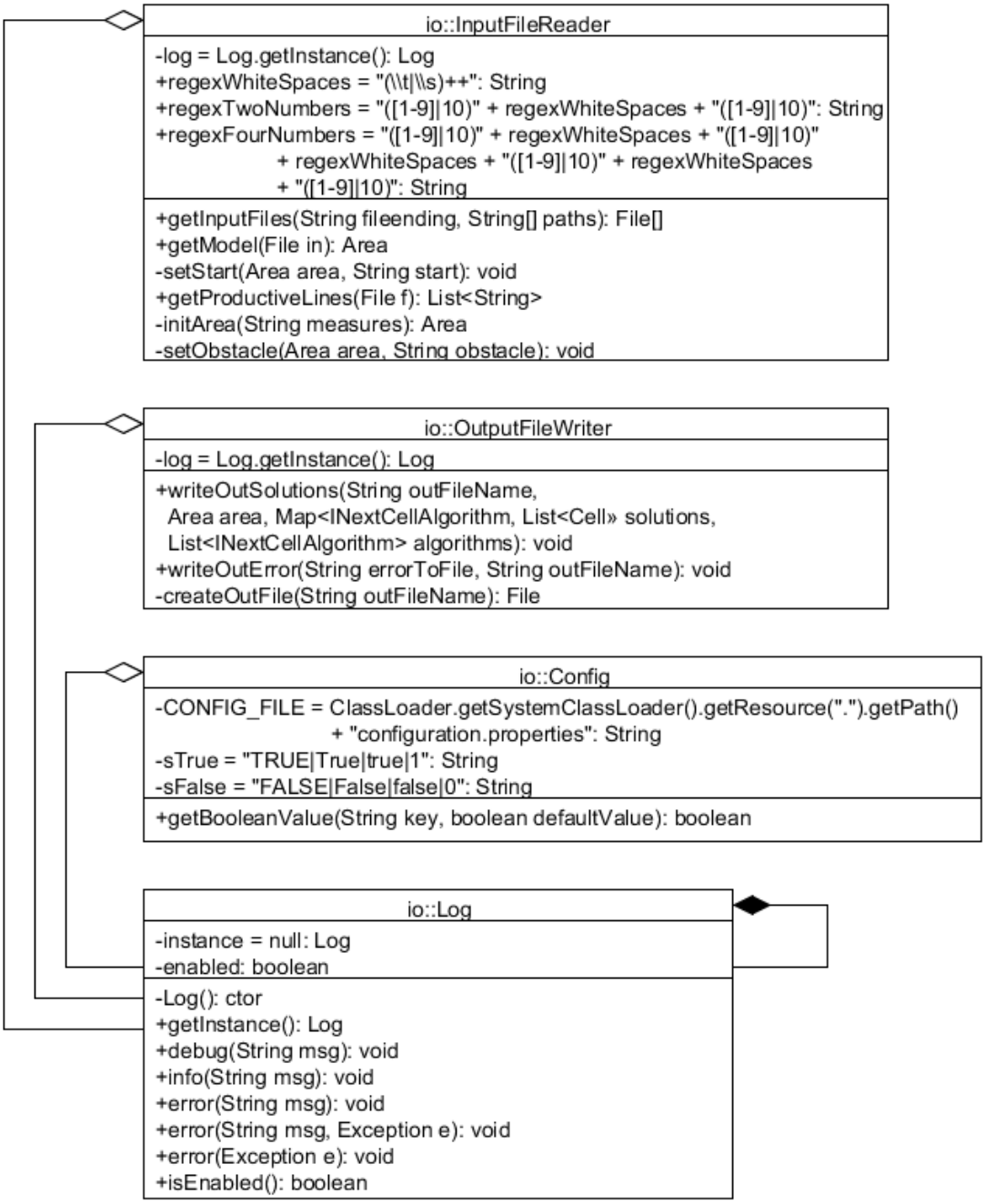
Modul control



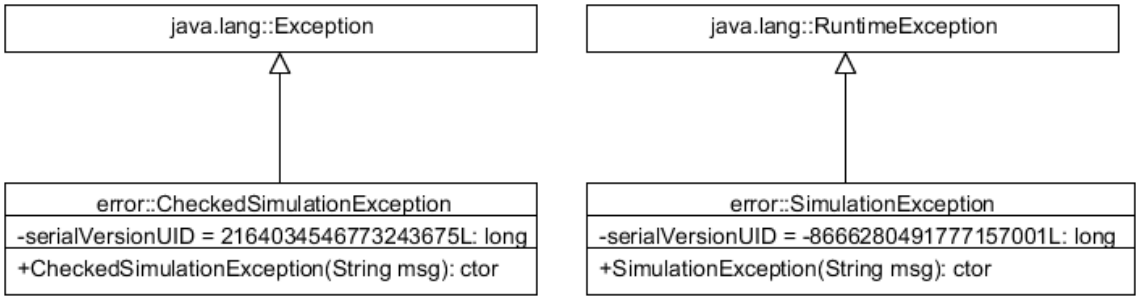
Modul model


model::Cell
-left: Cell
-right: Cell
-top: Cell
-bottom: Cell
-neighbours: Set<Cell>
-y: int
-x: int
-obstacle: boolean
-visited: boolean
+Cell(int y, int x, boolean obstacle): ctor
+getY(): int
+getX(): int
#setObstacle(boolean obstacle): void
+isObstacle(): boolean
#setRight(Cell right): void
#setBottom(Cell bottom): void
+setVisited(boolean visited): void
+isVisited(): boolean
+getAvailNeighbours(): Set<Cell>
+getDirectionOfNeighbour(Cell neighbour): Direction
+getNeighbourIfAvailable(Direction direction): Cell
+clonePath(List<Cell> path): List<Cell>
+toString(): String
+getAsString(List<Cell> path): String
1..n
1
1
4
model::Area
-start: Cell
-areaArray: Cell[][]
-obstacleCount: int
-cellCount: int
+Area(int y, int x): ctor
-initArea(int ylength, int xlength): void
+getCellCount(): int
+getObstacleCount(): int
+resetArea(): void
+setStart(int y, int x): void
+setObstacle(int yMin, int xMin, int yMax, int xMax): void
+getStart(): Cell
+toString(List<Cell> path): String

Modul io

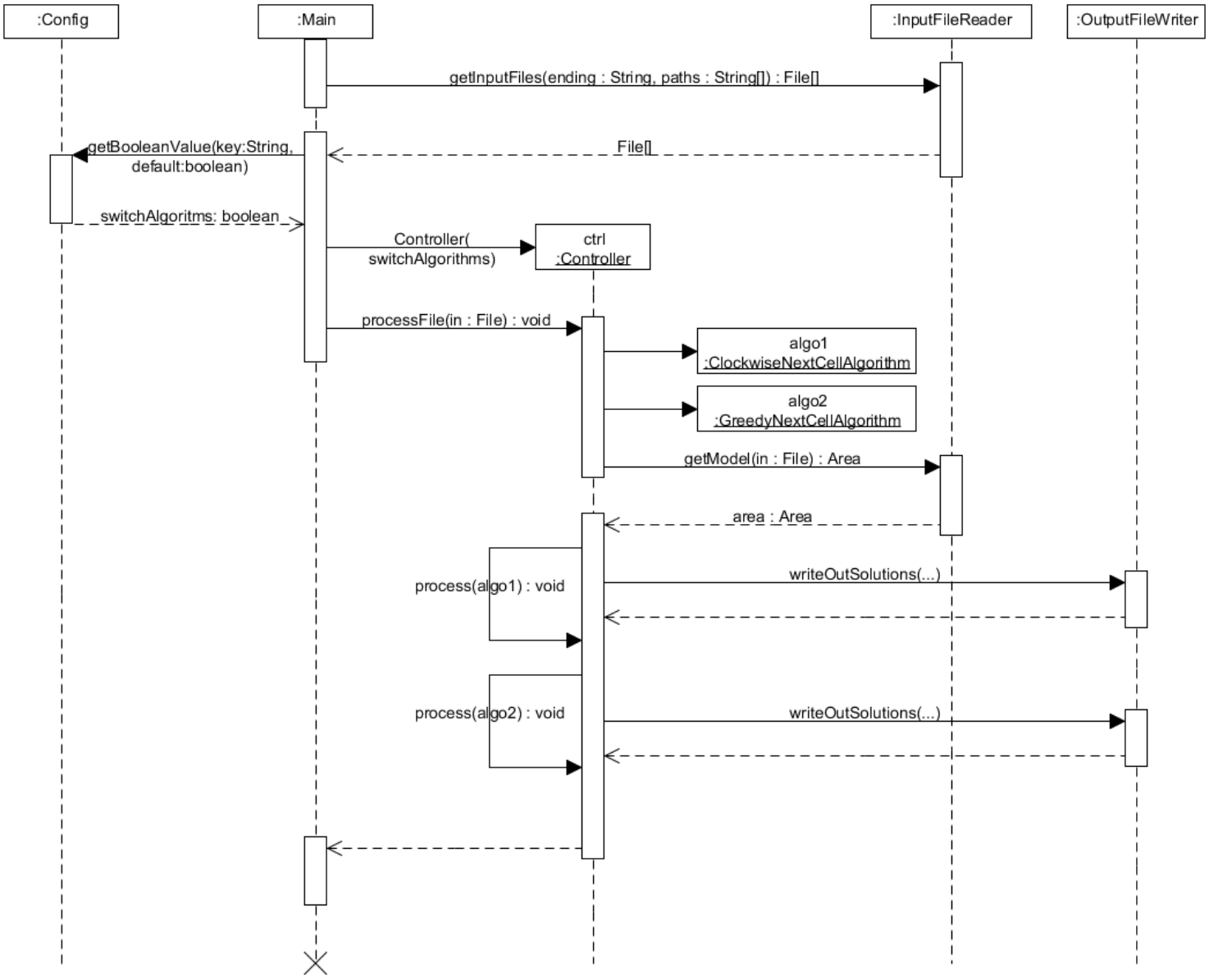

io::InputFileReader
-log = Log.getInstance(): Log
+regexWhiteSpaces = "(\\t|\\s)++": String
+regexTwoNumbers = "([1-9]|10)" + regexWhiteSpaces + "([1-9]|10)": String
+regexFourNumbers = "([1-9]|10)" + regexWhiteSpaces + "([1-9]|10)"
+ regexWhiteSpaces + "([1-9]|10)" + regexWhiteSpaces
+ "([1-9]|10)": String
+getInputFiles(String fileending, String[] paths): File[]
+getModel(File in): Area
-setStart(Area area, String start): void
+getProductiveLines(File f): List<String>
-initArea(String measures): Area
-setObstacle(Area area, String obstacle): void
io::OutputFileWriter
-log = Log.getInstance(): Log
+writeOutSolutions(String outFileName,
Area area, Map<INextCellAlgorithm, List<Cell» solutions,
List<INextCellAlgorithm> algorithms): void
+writeOutError(String errorToFile, String outFileName): void
-createOutFile(String outFileName): File
io::Config
-CONFIG_FILE = ClassLoader.getSystemClassLoader().getResource(".").getPath()
+ "configuration.properties": String
-sTrue = "TRUE|True|true|1": String
-sFalse = "FALSE|False|false|0": String
+getBooleanValue(String key, boolean defaultValue): boolean
io::Log
-instance = null: Log
-enabled: boolean
-Log(): ctor
+getInstance(): Log
+debug(String msg): void
+info(String msg): void
+error(String msg): void
+error(String msg, Exception e): void
+error(Exception e): void
+isEnabled(): boolean

Modul error


java.lang::Exception
java.lang::RuntimeException
error::CheckedSimulationException
-serialVersionUID = 2164034546773243675L: long
+CheckedSimulationException(String msg): ctor
error::SimulationException
-serialVersionUID = -8666280491777157001L: long
+SimulationException(String msg): ctor

4.3.2 Algorithmen und Abläufe

4.3.2.1 Sequenzdiagramm für Grobablauf

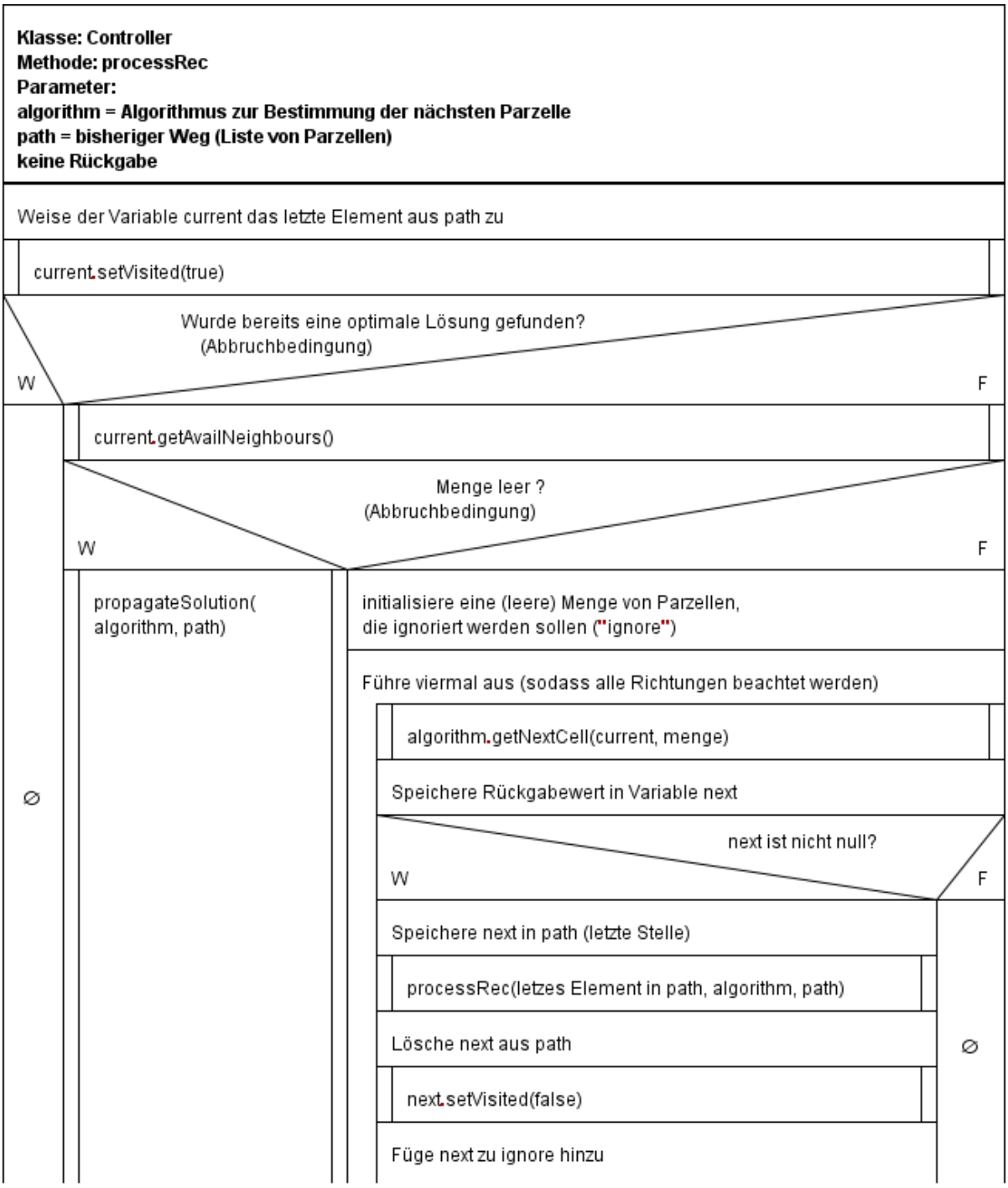


4.3.2.2 Formale Algorithmenbeschreibung

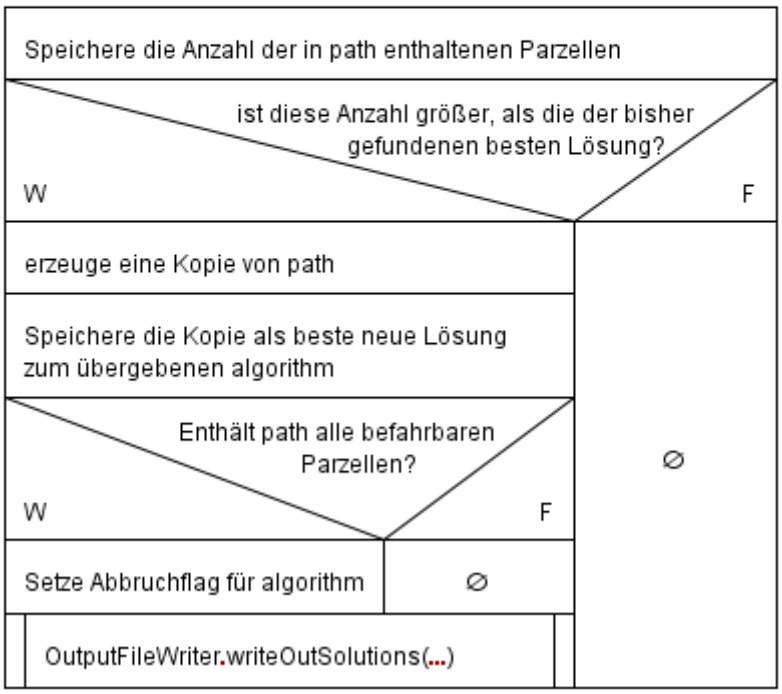
Dieses Kapitel enthält Nassi-Shneiderman-Diagramme zu den zentralen Methoden / Algorithmen.

Hauptalgorithmus (Tiefensuche)

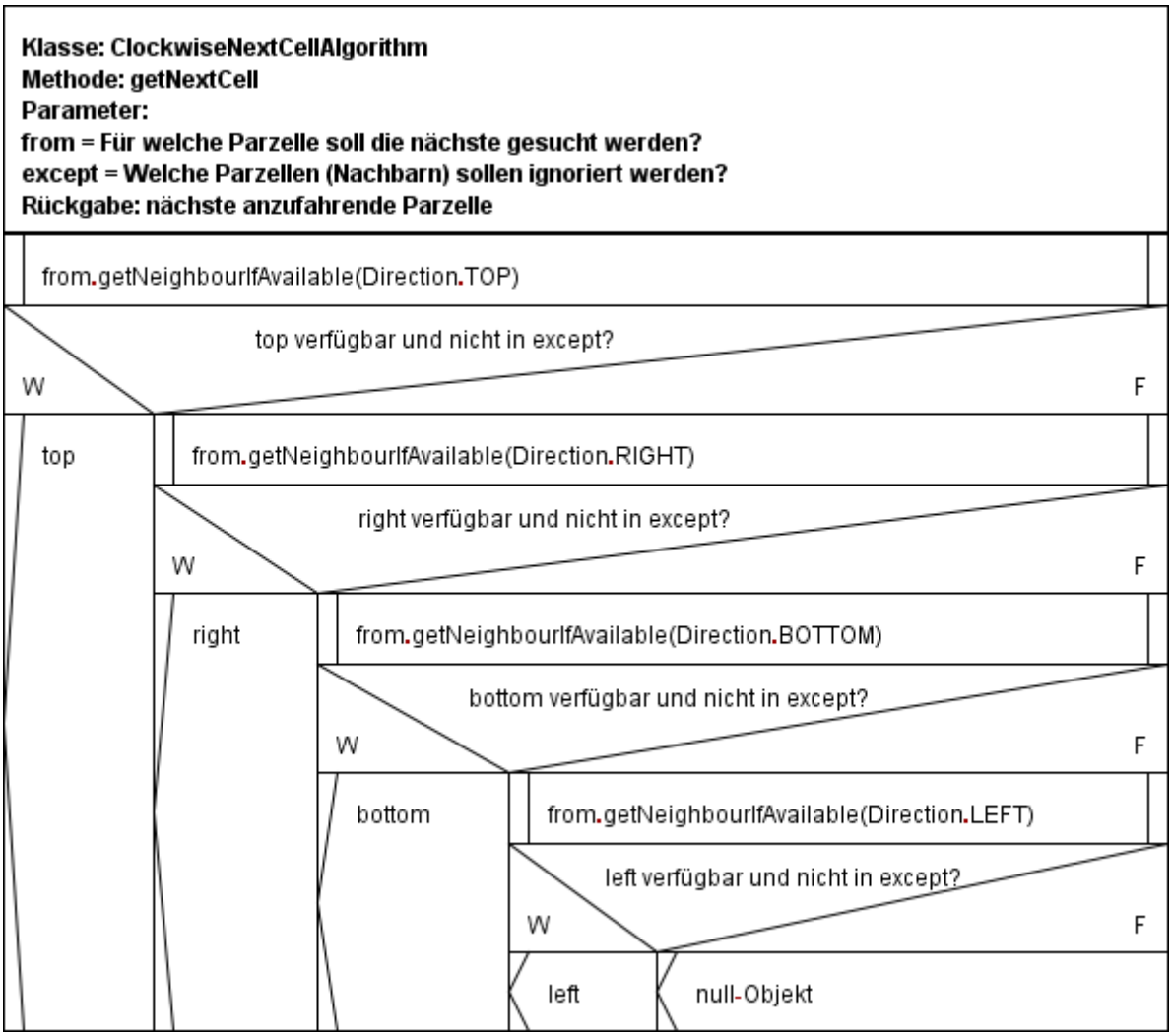
Klasse: Controller **Methode: process** **Parameter:** **algorithm = Algorithmus zur Berechnung der nächsten Parzelle** **keine Rückgabe**
Setze die Parkettfläche zurück (besucht-Eigenschaft)
Initialisiere eine Liste path zur Speicherung von Zellen und füge start hinzu
processRec(algorithm, path)


Klasse: Controller
Methode: processRec
Parameter:
algorithm = Algorithmus zur Bestimmung der nächsten Parzelle
path = bisheriger Weg (Liste von Parzellen)
keine Rückgabe
Weise der Variable current das letzte Element aus path zu
current.setVisited(true)
Wurde bereits eine optimale Lösung gefunden?
(Abbruchbedingung)
W
F
current.getAvailNeighbours()
Menge leer ?
(Abbruchbedingung)
W
F
propagateSolution(
algorithm, path)
initialisiere eine (leere) Menge von Parzellen,
die ignoriert werden sollen ("ignore")
Führe viermal aus (sodass alle Richtungen beachtet werden)
algorithm.getNextCell(current, menge)
Speichere Rückgabewert in Variable next
next ist nicht null?
W
F
Speichere next in path (letzte Stelle)
processRec(letzes Element in path, algorithm, path)
Lösche next aus path
next.setVisited(false)
Füge next zu ignore hinzu
∅
∅

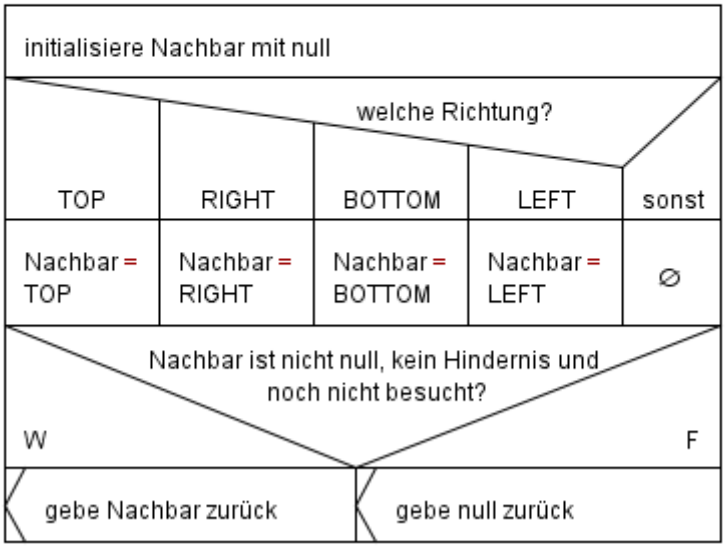
Klasse: Controller
Methode: propagateSolution
Parameter:
algorithm = Algorithmus zur Berechnung der nächsten Parzelle
path = ermittelter Weg über die Parkettfläche
keine Rückgabe
Anmerkung: logging code wird in diesem Diagramm nicht aufgeführt



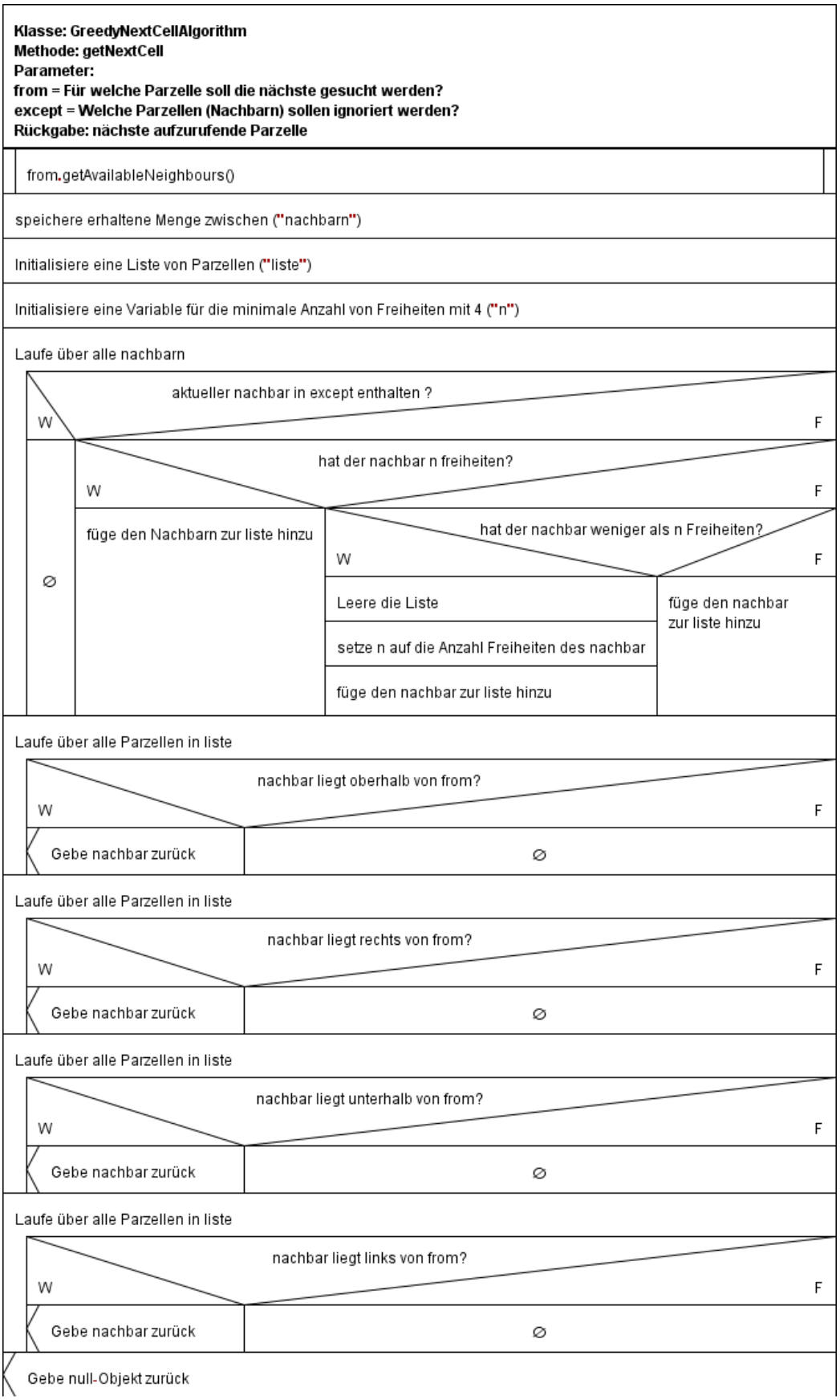
Uhrzeiger-Strategie



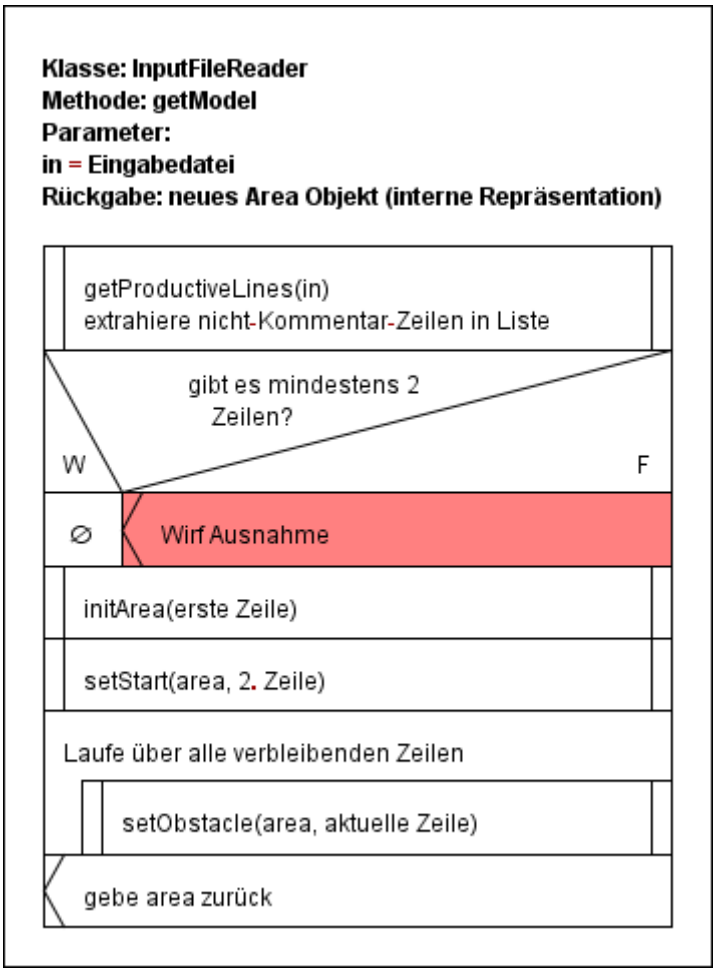
Klasse: Cell
Methode: getNeighbourIfAvailable
Parameter:
direction = welcher Nachbar soll zurückgegeben werden
Rückgabe: Nachbar in angegebener Richtung oder null



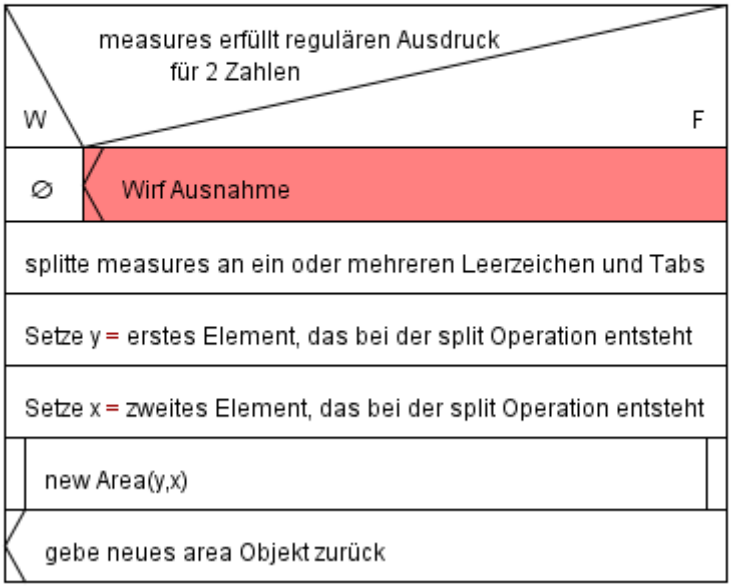
Greedy-Strategie



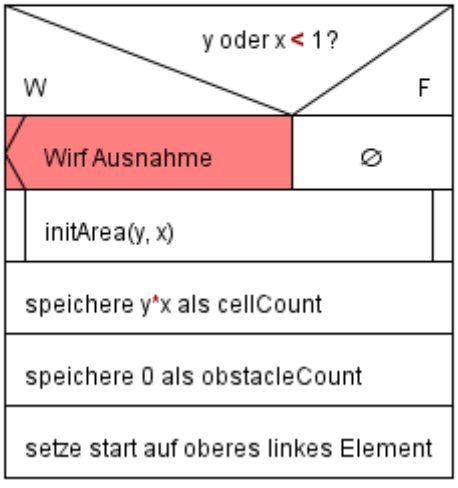
Überführung in interne Datenstruktur

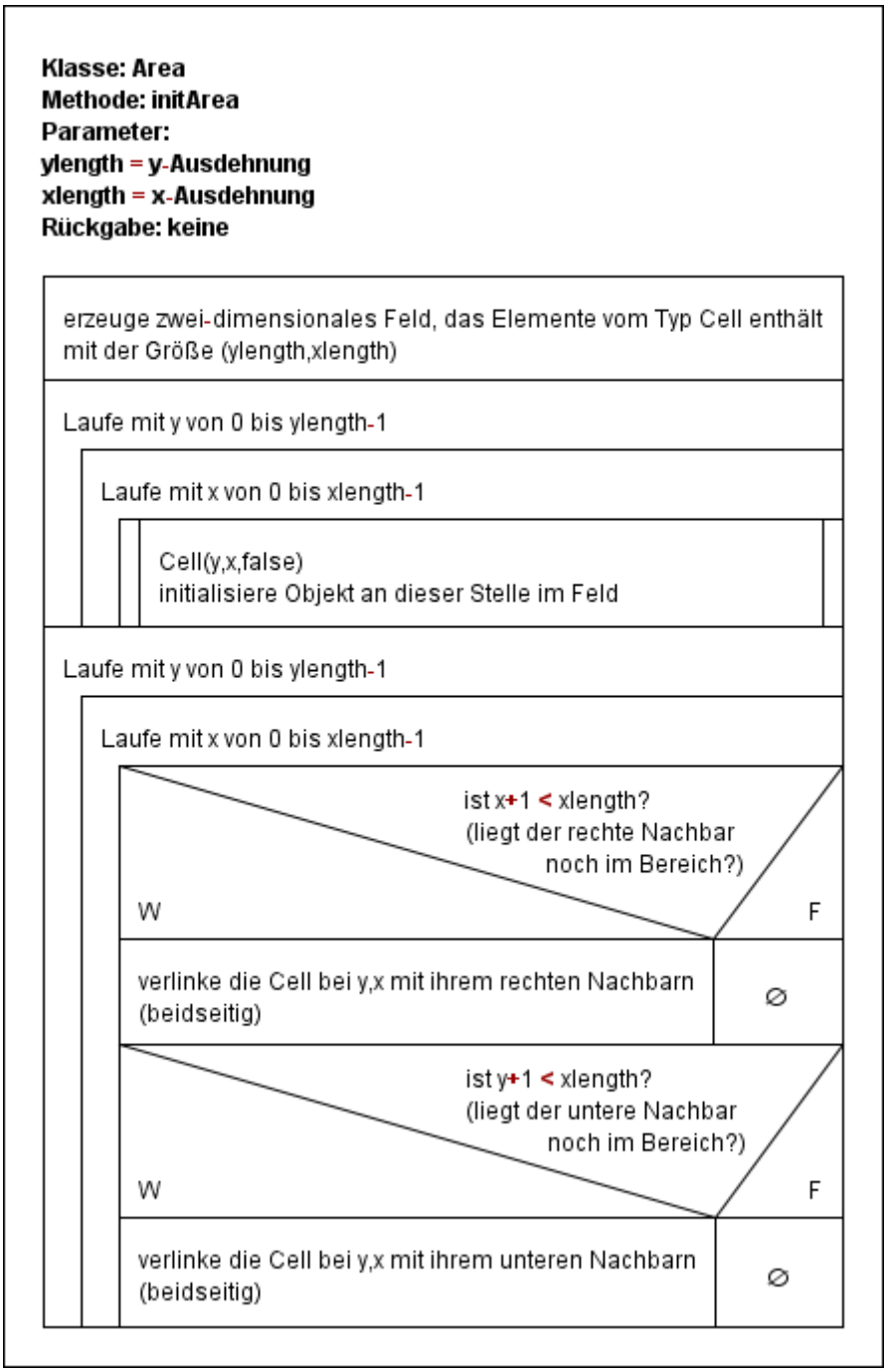


Klasse: InputFileReader
Methode: initArea
Parameter:
measures = Abmessungen als Zeichenkette
Rückgabe: area Objekt (interne Repräsentation)



Klasse: Area
Methode: Area
Parameter:
y = y-Ausdehnung
x = x-Ausdehnung
Rückgabe: Konstruktur




Klasse: Area
Methode: initArea
Parameter:
ylength = y-Ausdehnung
xlength = x-Ausdehnung
Rückgabe: keine
erzeuge zwei-dimensionales Feld, das Elemente vom Typ Cell enthält mit der Größe (ylength,xlength)
Laufe mit y von 0 bis ylength-1
Laufe mit x von 0 bis xlength-1
Cell(y,x,false)
initialisiere Objekt an dieser Stelle im Feld
Laufe mit y von 0 bis ylength-1
Laufe mit x von 0 bis xlength-1
ist x+1 < xlength?
(liegt der rechte Nachbar noch im Bereich?)
W
F
verlinke die Cell bei y,x mit ihrem rechten Nachbarn (beidseitig)
∅
ist y+1 < xlength?
(liegt der untere Nachbar noch im Bereich?)
W
F
verlinke die Cell bei y,x mit ihrem unteren Nachbarn (beidseitig)
∅

4.4 Erweiterungen, Modifikationen

Dieses Kapitel enthält Erläuterungen zu den Erweiterungen und Modifikationen, die während des praktischen Prüfungsteils gegenüber dem Konzept vom 1. Prüfungstag vorgenommen wurden.

4.4.1 Verbesserung des Datenmodells

Klasse model::Area

Diese Klasse wurde hinzugefügt, um die Parkettfläche nicht nur als verketteten Graphen (wie im Konzept) sondern auch als zwei-dimensionales Feld vorzuhalten. Außerdem konnte so die Initialisierung der internen Datenstruktur besser gekapselt werden. Verschiedene setter-Methoden aus der Klasse model::Cell konnten auf Sichbarkeit „protected" reduziert werden, sodass diese nur noch aus dem gleichen package (model) aufgerufen werden können. Dies verhindert unerwünschte Manipulationen der Datenstruktur von außen.
Die Klasse Area besitzt zudem eine Methode, die eine String-Repräsentation der Parkettfläche und eines darauf zurückgelegten Weges generiert. Diese Methode nutzt ebenfalls die interne Speicherung als zwei-dimensionales Feld.

Klasse io::OutputFileWriter

Da zur Ausgabe einer Lösung oder eines Fehles in die Ausgabedatei mehrere komplexe Methoden implementiert wurden, macht es Sinn, diese in eine eigene Klasse auszulagern.

Klasse io::Config

Diese Klasse wurde hinzugefügt, um Funktionen zum Lesen einer Konfigurationsdatei bereitzustellen. Diese Konfiguration bietet die Möglichkeit, den Detailgrad der Konsolenausgabe einzustellen, sowie die Ausführungsreihenfolge der Strategien zu modifizieren (Dies wurde notwendig, um Performance-kritische Testfälle auszuführen).

Enum model::Direction

Dieser Datentyp dient zur Repräsentation der Richtungen oben, unten, links, rechts. Bereits im ursprünglichen Konzept enthalten (Methode model::Cell::getDirectionOfNeighbour) wurde er dort nicht explizit im Klassendiagramm aufgeführt.

Änderungen an Klasse control::Controller

- Zur Speicherung Strategie-spezifischer Daten (zum Beispiel der momentan besten Lösung) wurden Schlüssel-Wertepaar-Tabellen eingeführt, die zu der entsprechenden Strategie (Schlüssel) den angegebenen Wert enthalten. Dies garantiert ein Finden des entsprechenden Werts in O(1), was die Performance begünstigt, da die Werte insbesondere während der Tiefensuche (und daher sehr oft) benötigt werden.
- Die Prüfung, ob ein gefundener Weg, besser ist als alle vorherigen, und das Abspeichern desselben wurde in eine eigene Methode ausgelagert (propagateSolution). Diese Methode wurde außerdem dahingehend verbessert, dass bei jedem Fund einer besseren Lösung die Ausgabedatei neu geschrieben wird. Dies führt dazu, dass selbst beim Programmabbruch von außen bereits eine Ausgabedatei mit der bis dato besten Lösung existiert.
- Es wurde eine Methode convertToRoutePlan hinzugefügt, die eine Liste von benachbarten Parzellen in den geforderten Routenplan überführt.

Änderungen an Klasse model::Cell

- Die Nachbar-Parzellen werden nicht nur als einzelne Referenzen vorgehalten, sondern auch in einer Menge. Dies erlaubt die effizientere Implementierung der Methode model::Cell::getAvailNeighbours
- Die Methode getTopIfAvail, getRightIfAvail, getBottomIfAvail, getLeftIfAvail wurde zu einer Methode zusammengefasst, die die Richtung (model::Direction) als Übergabeparameter erwartet.
- Die Methode clone wurde nicht benötigt. Stattdessen wurde eine statische Methode clonePath eingeführt, die dazu dient, eine Liste von Parzellen zu klonen. Das elementweise kopieren der Parzellen war an keine Stelle nötig.

4.4.2 Steigerung der Algorithmen-Effizienz

Die drei Algorithmen, die im Konzept als Nassi-Shneiderman-Diagramme definiert wurden, sind grundsätzlich beibehalten worden. Folgende marginale Änderungen wurden vorgenommen:

- Die Anzahl der bereits ausprobierten Sackgassen wird geloggt, wenn eine neue beste Lösung gefunden wird. Diese zusätzliche Ausgabe wurde implementiert, um die Effizienz von Greedy- und Uhrzeiger-Strategie besser vergleichen zu können.
- Die Greedy-Strategie nutzt intern keine Sortierung nach Freiheitsgraden und Richtungen, da das Sortieren nach Richtung sehr aufwändig ist. Stattdessen werden zunächst alle Nachbarn mit minimalem Freiheitsgrad identifiziert. Dann wird für alle diese Nachbarn iterativ geprüft, ob sie oberhalb (, rechts, unterhalb, links) von der aktuellen Parzelle liegen. Wenn dies so ist, wird der entsprechende Nachbar sofort zurückgegeben. Das Verhalten des Algorithmus, wie in der Verfahrensbeschreibung definiert, ändert sich hierdurch nicht. Siehe auch zugehöriges Nassi-Shneiderman-Diagramm.

4.5 Testdokumentation

Mit Hilfe von Tests wird sichergestellt, dass ein Softwaresystem korrekt funktioniert und den Anforderungen genügt. Das entwickelte System wurde testgetrieben entwickelt, d.h. auf Basis der Anforderungen (Aufgabenstellung) werden Testfälle definiert, die nach der Implementierung mit Erfolg durchgeführt werden müssen. Hierbei handelt es sich um Blackbox-Tests, die auf definierte Eingabedaten dazu passende, definierte Ausgabedaten erwarten. Dabei wird der Code des Systems außer Acht gelassen.* Die Testfälle sollen ein möglichst breites Spektrum von verschiedenen Eingabeklassen umfassen. Es wird unterschieden zwischen OK-Testfällen und Not-OK-Testfällen, wobei die OK-Testfälle in Normal- und Randfälle unterteilt werden können.
Im nächsten Kapitel sind die Testfälle beschrieben, die für dieses System erstellt und getestet wurden.
*: Dies gilt nicht für die Tests, die im Hinblick auf Performance erstellt wurden. Diese wurden speziell auf die intern verwendeten Strategien, sowie die Tiefensuche angepasst.

4.5.1 Testfälle

Eine Anleitung zur Ausführung der Testfälle findet sich im Kapitel Ausführung der Testfälle.

Nachfolgend wird beschrieben, welche Testfälle erstellt und durchgeführt wurden und welchen Zweck sie erfüllen. Außerdem wird jeweils das Ergebnis diskutiert.

4.5.1.1 IHK-Testfälle

IHK-Bespiel 1

Das erste IHK-Beispiel ist ein einfacher Basistestfall: Die Parkettmaße sind 5x5 und es gibt keine Hindernisse:

```
  1 2 3 4 5
1 S
2
3
4
5
```

Für dieses Beispiel laufen Uhrzeiger- und Greedy-Strategie zunächst identisch ab, was daran liegt, dass der Startpunkt am oberen Parkettrand liegt. Durch diesen Umstand wählt auch die Uhrzeiger-Strategie weitere Randzellen als nächste Zellen aus.
Bis zu folgendem Zustand produzieren die Strategien das gleiche Ergebnis:

```
  1 2 3 4 5
1 > > > > V
2     V < V
3     V ^ V
4     V ^ V
5   Z < ^ <
```

Anschließend wählt die Uhrzeiger-Strategie gemäß ihrer Priorisierung den Weg nach oben aus. Die Greedy-Strategie jedoch trifft eine andere Wahl: Es wird erkannt, dass die benachbarte Parzelle zur linken ihrerseits nur noch einen freien Nachbarn hat (die darüber liegende Parzelle). Die von der Uhrzeiger-Strategie ausgewählte hingegen hat noch zwei freie Nachbarn. Daher entscheidet sich die Greedy-Strategie an dieser Stelle anders als die Uhrzeiger-Strategie, nämlich für die linke Nachbar-Parzelle.
Dies findet auch im Anschluss Anwendung, sodass die Greedy-Strategie die verbleibende Fläche von unten nach oben füllt, wohingegen die Uhrzeiger-Strategie die Fläche von rechts nach links füllt.
Insgesamt ergeben sich folgende Wege:
Uhrzeiger-Strategie:

```
   1 2 3 4 5
 1 > > > > V
 2 V < V < V
 3 V ^ V ^ V
 4 V ^ V ^ V
 5 Z ^ < ^ <
```

Greedy-Strategie:

```
   1 2 3 4 5
 1 > > > > V
 2 V < V < V
 3 Z ^ V ^ V
 4 > ^ V ^ V
 5 ^ < < ^ <
```

Beide Algorithmen finden diese Lösung im ersten Versuch, bzw. ohne dass die Tiefensuche „rückwärts" nach Alternativwegen durchsucht werden muss. Der Hauptalgorithmus erkennt jeweils, dass eine optimale Lösung gefunden wurde und bricht die Tiefensuche daraufhin ab. Der Testfall dient demnach zur Prüfung der grundlegenden Funktionalität der Weg-Finde-Strategien.

Fabian Braun, Prüflingsnummer 101 20510, Dokumentation

Eingabedatei:

```
; Abmessungen der Flaeche
5 5
; Startparzelle
1 1
; Eckparzellen der Hindernisse, ein Hindernis pro Zeile
; (in diesem Fall ohne Hindernisse)
```

Ausgabedatei:

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

Uhrzeiger-Strategie:
   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

Greedy-Strategie:
   1 2 3 4 5
 1 > > > > V
 2 V < V < V
 3 Z ^ V ^ V
 4 > ^ V ^ V
 5 ^ < < ^ <

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

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

IHK-Beispiel 2

Das zweite IHK-Beispiel ist komplexer als das Erste, da es Hindernissparzellen enthält:

```
 1 2 3 4 5 6
 1
 2       H
 3       H
 4       H
 5 S     H
```

Das Finden eines optimalen Weges ist allerdings auch hier möglich.
In diesem Beispiel kommt die Tiefensuche zur Anwendung, da beide Strategien hier den optimalen Weg nicht im ersten Versuch finden. Zustand beim Erreichen der ersten Sackgasse:
Uhrzeiger-Strategie:

```
   1 2 3 4 5 6
 1 > > > > > V
 2 ^     H Z V
 3 ^     H ^ V
 4 ^     H ^ V
 5 ^     H ^ <
```

Greedy-Strategie:

```
   1 2 3 4 5 6
 1 > > > > > V
 2 ^     H V <
 3 ^     H > V
 4 ^     H Z V
 5 ^     H ^ <
```

Dieses Beispiel demonstriert demnach, ob die Tiefensuche korrekt implementiert ist, da in diesem Fall eine optimale Lösung bei Anwendung beider Strategien gefunden werden muss. Dies ist der Fall wie die finalen Ausgaben des Tests zeigen:
Uhrzeiger-Strategie:

```
   1 2 3 4 5 6
 1 > V > > > V
 2 ^ V ^ H Z V
 3 ^ V ^ H ^ V
 4 ^ V ^ H ^ V
 5 ^ > ^ H ^ <
```

Greedy-Strategie:

```
   1 2 3 4 5 6
 1 > V > > > V
 2 ^ V ^ H V <
 3 ^ V ^ H > V
 4 ^ V ^ H Z V
 5 ^ > ^ H ^ <
```

Dieser Testfall zeigt jedoch auch exemplarisch, dass die Greedy-Strategie schneller einen optimalen Weg findet, als die Uhrzeiger-Strategie (unter der Voraussetzung, dass ein solcher existiert):

Für dieses Beispiel läuft die Tiefensuche mit Greedy-Strategie in 88 Sackgassen, wohingegen die Uhrzeiger-Strategie 112 Sackgassen untersucht, bis jeweils ein optimaler Weg gefunden wird.

Eingabedatei:

```
; Abmessungen der Flaeche
5 6
; Startparzelle
5 1
; Eckparzellen der Hindernisse, ein Hindernis pro Zeile
2 4 5 4
```

Ausgabedatei:

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

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

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

Greedy-Strategie:
   1 2 3 4 5 6
 1 > V > > > V
 2 ^ V ^ H V <
 3 ^ V ^ H > V
 4 ^ V ^ H Z V
 5 ^ > ^ H ^ <

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

zu versiegelnde Parzellen:   26
Hindernisparzellen:          4
versiegelte Parzellen:       26
nicht versiegelte Parzellen: 0
```

IHK-Beispiel 3

Das dritte IHK-Beispiel unterscheidet sich von den vorherigen insbesondere durch die Eigenschaft, dass für diesen Fall kein Weg existiert, der alle Parzellen einschließt. Der Testfall demonstriert demnach die Verhaltensweisen der Algorithmen, wenn sie mit dieser Schwierigkeit konfrontiert werden. Beide Algorithmen durchlaufen den Graphen per Backtracking und brechen erst nach dem Finden einer optimalen Lösung oder nach dem Durchlaufen aller Möglichkeiten ab. Dies hat zur Folge, dass beide Algorithmen immer alle möglichen Wege untersuchen, wenn es keinen optimalen Weg gibt. Dies geschieht auch bei diesem Beispiel und führt abschließend zu folgendem (in diesem Beispiel für beide Strategien zufällig identischen) Weg:

```
   1 2 3 4 5
 1 > V
 2 ^ V H H H
 3   > > > Z
```

Alle IHK-Beispiele dienen zudem dazu, die korrekte Ausgabe zu prüfen. Diese beinhaltet außer der visuellen Darstellung der Wege auch: Die Parkettfläche im Rohzustand, die Routenlisten der gefundenen Wege, die Lösungsmetriken (Anzahl zu versiegelnde Parzellen, Hindernisparzellen, versiegelte Parzellen und nicht versiegelte Parzellen).

Fabian Braun, Prüflingsnummer 101 20510, Dokumentation

Eingabedatei:

```
;  Abmessungen der Fläche
  3 5
;  Startparzelle
  2 1
;  Eckparzellen der Hindernisse, ein Hindernis pro Zeile
  2 3   2 5
```

Ausgabedatei:

```
Startposition (2,1)
   1 2 3 4 5
 1
 2 S   H H H
 3

Uhrzeiger-Strategie:
   1 2 3 4 5
 1 > V
 2 ^ V H H H
 3   > > > Z

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

Greedy-Strategie:
   1 2 3 4 5
 1 > V
 2 ^ V H H H
 3   > > > Z

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

zu versiegelnde Parzellen:   12
Hindernisparzellen:          3
versiegelte Parzellen:       8
nicht versiegelte Parzellen: 4
```

4.5.1.2 Eigene OK-Testfälle

Die nachfolgenden Testfälle wurden sämtlich mit Erfolg durchgeführt und sind im Prüfprodukt mit zugehöriger Ein- und Ausgabedatei enthalten.

Leerzeichen

Dateiname: ok_format_whitespaces.in
Beschreibung: Die Eingabe enthält Leerzeichen und Tabulatoren am Anfang und Ende jeder Zeile, sowie zwischen den Werte-Tupeln in den Datenzeilen.
Zweck: Es ist erlaubt in jeder Zeile an beliebigen Stellen Leerzeichen und Tabulatoren einzufügen (außer zwischen den Ziffern 1 und 0 der Zahl 10). Dieser Test stellt sicher, dass das Lesen und Parsen der Datei auch in diesem Fall erfolgreich ist.

Eingabedatei:

```
; Leerzeichen
  ; Parkettmaße:
       4     4
            ; Startparzelle

      4       3
; Eckparzellen der Hindernisse, ein Hindernis pro Zeile
      2      2     2     4
```

Ausgabedatei:

```
Startposition (4,3)
   1 2 3 4
 1
 2   H H H
 3
 4     S

Uhrzeiger-Strategie:
   1 2 3 4
 1 > > > Z
 2 ^ H H H
 3 ^ V < <
 4 ^ < > ^

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

Greedy-Strategie:
   1 2 3 4
 1 > > > Z
 2 ^ H H H
 3 ^ V < <
 4 ^ < > ^

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

zu versiegelnde Parzellen:   13
Hindernisparzellen:          3
versiegelte Parzellen:       13
nicht versiegelte Parzellen: 0
```

Maximale Feldgröße ohne Hindernisse

Dateiname: ok_maximalsize_no_obstacles.in
Beschreibung: Die Parkettfläche ist maximal groß. Auf ihr befinden sich keine Hindernisse. Der Startpunkt liegt in der unteren rechten Ecke des Parketts.
Zweck: Dieser Test stellt sicher, dass das Parkett die maximale Größe annehmen kann (Randfall). Außerdem wird geprüft, ob nach Finden der ersten optimalen Lösung die Suche terminiert wird.

Eingabedatei:

```
; Maximale Feldgröße ohne Hindernisse
10 10
; Startparzelle
10 10
; keine Hindernisse
```

Ausgabedatei:

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

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

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

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

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

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

Maximale Feldgröße, nur ein Weg

Dateiname: ok_maximalsize_one_way.in
Beschreibung: Auf der maximalgroßen Parkettfläche gibt es genau einen möglichen Weg, der durch Hindernisse erzwungen wird.
Zweck: Dieser Test stellt sicher, dass die Algorithmen keine Hindernisse als nächste Parzelle auswählen und einen fest durch Hindernisse vorgegebenen Weg finden.

Eingabedatei:

```
; Maximale Feldgröße, nur ein Weg
10 10
; Startparzelle
1 1
; Hindernisse:
2 1 2 9
4 2 4 10
6 1 6 9
8 2 8 10
10 1 10 9
```

Ausgabedatei:

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

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

Routenplan:
1,1 / 1,10 / 3,10 / 3,1 / 5,1 / 5,10 / 7,10 / 7,1 / 9,1 / 9,10 / 10,10

Greedy-Strategie:
   1 2 3 4 5 6 7 8 9 10
 1 > > > > > > > > > V
 2 H H H H H H H H H V
 3 V < < < < < < < < <
 4 V H H H H H H H H H
 5 > > > > > > > > > V
 6 H H H H H H H H H V
 7 V < < < < < < < < <
 8 V H H H H H H H H H
 9 > > > > > > > > > V
10 H H H H H H H H H Z

Routenplan:
1,1 / 1,10 / 3,10 / 3,1 / 5,1 / 5,10 / 7,10 / 7,1 / 9,1 / 9,10 / 10,10

zu versiegelnde Parzellen:   55
Hindernisparzellen:          45
versiegelte Parzellen:       55
nicht versiegelte Parzellen: 0
```

Unterschied Greedy-Strategie und Uhrzeiger-Strategie

Dateiname: ok_mediumsize_no_obstacles.in
Beschreibung: Die Parkettfläche ist leer und misst 4x4. Der Startpunkt liegt am unteren Rand, jedoch nicht in einer Ecke. Dies ist das Beispiel, das in der Konzeptphase am 1. Prüfungstag erläutert wurde.
Zweck: Das Beispiel dient zur Demonstration des Unterschieds zwischen Greedy- und Uhrzeiger-Strategie. Siehe auch Kapitel Zweite Strategie. Dieser Test zeigt, dass die Greedy-Strategie wie geplant umgesetzt wurde.

Eingabedatei:

```
; Unterschied Greedy-Strategie und Uhrzeiger-Strategie (aus Konzept 1. Tag)
4 4
; Startparzelle
4 3
; keine Hindernisse
```

Ausgabedatei:

```
Startposition (4,3)
   1 2 3 4
 1
 2
 3
 4     S

Uhrzeiger-Strategie:
   1 2 3 4
 1 > > > V
 2 ^ V < V
 3 ^ V ^ V
 4 ^ < ^ Z

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

Greedy-Strategie:
   1 2 3 4
 1 > V V <
 2 ^ Z V ^
 3 ^ V < ^
 4 ^ < > ^

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

zu versiegelnde Parzellen:   16
Hindernisparzellen:          0
versiegelte Parzellen:       16
nicht versiegelte Parzellen: 0
```

Mittlere Feldgröße, zwei Sackgassen

Dateiname: ok_mediumsize_obstacles_impossible.in
Beschreibung: Die Parkettfläche misst 4x4. Zwei Hindernisse wurden so platziert, dass es von vornherein zwei Sackgassen gibt, die nicht beide erreicht werden können. Daher gibt es keinen optimalen Weg.
Zweck: Dieser Test prüft, ob die Tiefensuche auch dann terminiert, wenn kein optimaler Weg existiert.

Eingabedatei:

```
; Mittlere Feldgröße, zwei Sackgassen
4 4
; Startparzelle
4 3
; Eckparzellen der Hindernisse, ein Hindernis pro Zeile
1 1 1 1
3 1 3 1
```

Ausgabedatei:

```
Startposition (4,3)
   1 2 3 4
 1 H
 2
 3 H
 4     S

Uhrzeiger-Strategie:
   1 2 3 4
 1 H V < <
 2   > V ^
 3 H V < ^
 4 Z < > ^

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

Greedy-Strategie:
   1 2 3 4
 1 H V < <
 2   > V ^
 3 H V < ^
 4 Z < > ^

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

zu versiegelnde Parzellen:   14
Hindernisparzellen:          2
versiegelte Parzellen:       13
nicht versiegelte Parzellen: 1
```

Mittlere Feldgröße, Hindernisse, optimaler Weg existiert

Dateiname: ok_mediumsize_obstacles_optimal.in
Beschreibung: Die Parkettfläche misst 6x4. Es sind Hindernisse vorhanden, aber es existieren dennoch mehrere optimale Wege.
Zweck: Dieser Test prüft, ob die Algorithmen nach dem Finden einer optimalen Lösung terminieren. Außerdem wird sichergestellt, dass die nicht quadratische Fläche kein unerwartetes Ergebnis oder einen Fehler hervorruft.

Eingabedatei:

```
; Mittlere Feldgröße, Hindernisse, optimaler Weg existiert
6 4
; Startparzelle
5 4
; Eckparzellen der Hindernisse, ein Hindernis pro Zeile
5 2 5 3
```

Ausgabedatei:

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

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

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

Greedy-Strategie:
   1 2 3 4
 1 > > > V
 2 ^ > V V
 3 ^ ^ Z V
 4 ^ ^ < <
 5 ^ H H V
 6 ^ < < <

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

zu versiegelnde Parzellen:   22
Hindernisparzellen:          2
versiegelte Parzellen:       22
nicht versiegelte Parzellen: 0
```

Minimale Feldgröße

Dateiname: ok_minimalsize.in
Beschreibung: Die Parkettfläche misst 1x1.
Zweck: Dieser Randfall prüft, ob die minimal erlaubte Fläche zu einem Fehler führt.

Eingabedatei:

```
; Minimale Feldgröße
1 1
; Startparzelle
1 1
```

Ausgabedatei:

```
Startposition (1,1)
   1
 1 S

Uhrzeiger-Strategie:
   1
 1 S

Routenplan:
1,1

Greedy-Strategie:
   1
 1 S

Routenplan:
1,1

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

Reihenfolge Hindernisdefinition

Dateinamen: ok_obstacle_definition_max_first.in, ok_obstacle_definition_min_first.in
Beschreibung: Das erste Beispiel definiert ein Hindernis in der Reihenfolge 1. Linke, obere Ecke, 2. Rechte, untere Ecke. Das zweite Beispiel definiert das gleiche Hindernis in umgekehrter Reihenfolge. Die Ergebnisse zu beiden Beispielen müssen identisch sein.
Zweck: Es wird verifiziert, dass die Reihenfolge, in denen die Eckpunkte eines Hindernisses definiert werden, im Programm kein unterschiedliches Verhalten hervorruft.

Eingabedatei 1:

```
; Hindernis wird wie folgt definiert:
; erst untere rechte Ecke dann obere linke
; Dieser Testfall sollte das exakt identische Ergebnis liefern wie
; ok_obstacle_definition_min_first
3 3
; Startparzelle
1 1
; Eckparzellen der Hindernisse
3 3 1 2
```

Eingabedatei 2:

```
; Hindernis wird wie folgt definiert:
; erst obere linke Ecke dann untere rechte
; Dieser Testfall sollte das exakt identische Ergebnis liefern wie
; ok_obstacle_definition_max_first
3 3
; Startparzelle
1 1
; Eckparzellen der Hindernisse
1 2 3 3
```

Ausgabedatei (identisch):

```
Startposition (1,1)
   1 2 3
 1 S H H
 2   H H
 3   H H

Uhrzeiger-Strategie:
   1 2 3
 1 V H H
 2 V H H
 3 Z H H

Routenplan:
1,1 / 3,1

Greedy-Strategie:
   1 2 3
 1 V H H
 2 V H H
 3 Z H H

Routenplan:
1,1 / 3,1

zu versiegelnde Parzellen:   3
Hindernisparzellen:          6
versiegelte Parzellen:       3
nicht versiegelte Parzellen: 0
```

4.5.1.3 Eigene Performance-kritische OK-Testfälle

Da anhand der Ergebnisse dieser Testfälle verschiedene Eigenschaften und Verhaltensweisen des Softwaresystems sichtbar werden, sind in diesem Kapitel genauere Erläuterungen zu den Ergebnissen hinzugefügt worden.

Mittlere Feldgröße, Hindernisse, optimaler Weg existiert nicht

Dateiname: ok_mediumsize_obstacles_impossible.in
Beschreibung: Parkettfläche 6x6. Durch Hindernisse wird verhindert, dass eine Eckparzelle erreicht werden kann. Die bestmögliche Lösung beinhaltet jedoch alle Parzellen außer dieser.
Zweck: Die Tiefensuche muss vollständig durchlaufen werden, was bei dieser Feldgröße bereits einen nicht zu unterschätzenden Aufwand bedeutet. Die bestmögliche Lösung sollte gefunden werden. Dieser Test dient außerdem zur Prüfung, ob ein Programm die Intelligenz besitzt, isolierte Zellen zu erkennen und aus der Lösungssuche auszuschließen.
Ergebnis: Das Programm terminiert erfolgreich nach etwa 10 Sekunden. Ein bestmöglicher Weg wird nach beiden Strategien im ersten Versuch gefunden, jedoch wird die Suche nach Alternativen anschließend nicht abgebrochen, wodurch die lange Laufzeit zustande kommt.

Eingabedatei:

```
; Mittlere Feldgröße, Hindernisse, optimaler Weg existiert nicht
; Parkettmaße:
6 6
; Startparzelle
1 3
; Hindernisse
1 2 1 2
2 1 2 1
```

Ausgabedatei:

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

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

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

Greedy-Strategie:
   1 2 3 4 5 6
 1   H > > > V
 2 H V < V < V
 3 V < ^ V ^ V
 4 > V ^ V ^ V
 5 Z V ^ V ^ V
 6 ^ < ^ < ^ <

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

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

Mittlere Feldgröße, Hindernisse, optimaler Weg existiert nicht

Dateiname: ok_mediumsize_obstacles_impossible2.in
Beschreibung: Parkettfläche 6x6. Durch Hindernisse werden zwei Sackgassen erzeugt, die nicht beide im selben Weg enthalten sein können. Im Unterschied zum vorherigen Beispiel gibt es keine isolierte Parzelle, sodass es nicht trivial ist, die Unmöglichkeit eines optimalen Weges zu erkennen.
Zweck: Die Tiefensuche muss vollständig durchlaufen werden, was bei dieser Feldgröße bereits einen nicht zu unterschätzenden Aufwand bedeutet. Die bestmögliche Lösung sollte gefunden werden. Außerdem wird geprüft, ob ein Programm gegebenenfalls die Intelligenz besitzt, die Anzahl der in einem Weg erreichbaren Zellen im Vorhinein zu ermitteln und in die Lösungssuche, bzw. das Abbruchkriterium miteinzubeziehen.
Ergebnis: Das Programm terminiert erfolgreich nach etwa 10 Sekunden. Ein bestmöglicher Weg wird nach beiden Strategien gefunden. Die Greedy-Strategie findet diesen jedoch schon im 2. Versuch, wohingegen die Uhrzeiger-Strategie 36 Versuche braucht. Beide Strategien laufen anschließend jedoch weiter, ohne zu Erkennen, dass die aktuelle Lösung bereits eine bestmögliche ist.

Eingabedatei:

```
; Mittlere Feldgröße, Hindernisse, optimaler Weg existiert nicht
6 6
; Startparzelle
6 6
; Hindernisse
1 2 1 2
1 4 1 4
```

Ausgabedatei:

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

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

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

Greedy-Strategie:
   1 2 3 4 5 6
 1 Z H   H V <
 2 ^ < V < V ^
 3 > ^ V ^ V ^
 4 ^ < V ^ V ^
 5 > ^ V ^ V ^
 6 ^ < < ^ < ^

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

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

Maximale Feldgröße, Hindernisse, optimaler Weg existiert

Dateiname: ok_maximalsize_obstacles_optimal_complex.in
Beschreibung: Die Feldgröße ist maximal. Darauf befinden sich mehrere Hindernisse. Es gibt mindestens einen optimalen Weg, der nicht trivial zu finden ist.
Zweck: Es wird geprüft, wie sich das Programm bei komplexesten Problemen verhält, bei denen aber immerhin ein optimaler Weg existiert.
Ergebnis: Das Programm terminiert nicht in absehbarer Zeit. Die Lösung, die nach einer Minute (je Strategie) vorliegt ist nicht die bestmögliche. Das heißt, dass bei ausreichender Komplexität der Parkettfläche im ungünstigen Fall das Programm sehr lange braucht, um die optimale Lösung zu finden. Der Austausch der Strategie ändert daran nichts, da beide Strategien zu einem frühen Zeitpunkt „in die falsche Richtung laufen". Die Uhrzeiger-Strategie produziert innerhalb von einer Minute jedoch einen etwas besseren Weg, da sie an Position (4,2) den Weg nach oben priorisiert. Die Greedy-Strategie bevorzugt den rechten Nachbarn, da dieser weniger Freiheitsgrade hat.
Anmerkung: Die nachfolgenden Ausgabedateien enthalten jeweils nur eine Lösung, die vor dem Programmabbruch nach einer Minute ermittelt wurden. Da die Ausführung bei keiner Strategie in absehbarer Zeit terminiert, die Strategien jedoch sequentiell angewendet werden, musste zum Erhalt dieser Ausgabedateien die Reihenfolge der Strategien konfiguriert werden.

Eingabedatei:

```
; Maximale Feldgröße, Hindernisse, optimaler Weg existiert
;
; mögliche Lösung (manuell erstellt):
;    1 2 3 4 5 6 7 8 9 10
;  1 > V > > > > V V < <
;  2 ^ > ^ > > Z V V H ^
;  3 ^ < H ^ < < < V H ^
;  4 H ^ < H V < < V H ^
;  5 > V ^ V < > ^ V H ^
;  6 ^ V ^ V H ^ H V H ^
;  7 ^ > ^ > V ^ < V H ^
;  8 ^ V < < > V ^ V H ^
;  9 ^ < H ^ < < ^ < H ^
; 10 > > > > > > > > > ^
; Parkettmaße:
10 10
; Startparzelle:
10 1
; Hindernisse:
4 1 4 1
3 3 3 3
9 3 9 3
4 4 4 4
6 5 6 5
6 7 6 7
9 9 2 9
```

Ausgabedateien:

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

Uhrzeiger-Strategie:

```
   1 2 3 4 5 6 7 8 9 10
 1   > > > > > > > > V
 2   ^   V < < V < H V
 3   ^ H > V ^ V ^ H V
 4 H ^ Z H V ^ V ^ H V
 5 > ^ ^ V < ^ < ^ H V
 6 ^ > ^ V H   H ^ H V
 7 ^ ^ < > > > V ^ H V
 8 ^ > ^ V < < V ^ H V
 9 ^ ^ H > V ^ V ^ H V
10 ^ ^ < < < ^ < ^ < <
```

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

```
zu versiegelnde Parzellen:   86
Hindernisparzellen:          14
versiegelte Parzellen:       81
nicht versiegelte Parzellen: 5
```

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

Greedy-Strategie:

```
   1 2 3 4 5 6 7 8 9 10
 1       > V > > > > V
 2       ^ > ^ V < H V
 3     H ^ < V < ^ H V
 4 H > V H ^ > V ^ H V
 5 > ^ > > ^ V < ^ H V
 6 ^ V < < H V H ^ H V
 7 ^ > V ^ < > V ^ H V
 8 ^ Z > V ^ < V ^ H V
 9 ^ ^ H > V ^ V ^ H V
10 ^ ^ < < < ^ < ^ < <
```

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

```
zu versiegelnde Parzellen:   86
Hindernisparzellen:          14
versiegelte Parzellen:       78
nicht versiegelte Parzellen: 8
```

Maximale Feldgröße, Hindernisse, optimaler Weg existiert nicht

Dateiname: ok_maximalsize_obstacles_impossible.in
Beschreibung: Die Feldgröße ist maximal. Darauf befindet sich eine von Hindernissen eingeschlossene Eckparzelle.
Zweck: Dies ist ein worst-case Szenario, bei dem die Anzahl der möglichen Pfade durch den Graphen maximal wird. Das Problem wird dadurch verstärkt, dass es keinen optimalen Weg gibt und dass die Tiefensuche dementsprechend vollständig ablaufen muss.
Ergebnis: Das Programm terminiert nicht in absehbarer Zeit. Die Tiefensuche ist für diese Problemgröße kein geeignetes Mittel.

Eingabedatei:

```
; Maximale Feldgröße, Hindernisse, optimaler Weg existiert nicht
10 10
; Startparzelle
10 10
; Hindernisse
1 2 1 2
2 1 2 1
```

Ausgabedateien:

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

Uhrzeiger-Strategie:

```
   1 2 3 4 5 6 7 8 9 10
 1   H V < V < V < V <
 2 H Z V ^ V ^ V ^ V ^
 3 > ^ V ^ V ^ V ^ V ^
 4 ^ < V ^ V ^ V ^ V ^
 5 > ^ V ^ V ^ V ^ V ^
 6 ^ < V ^ V ^ V ^ V ^
 7 > ^ V ^ V ^ V ^ V ^
 8 ^ < V ^ V ^ V ^ V ^
 9 > ^ V ^ V ^ V ^ V ^
10 ^ < < ^ < ^ < ^ < ^
```

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

```
zu versiegelnde Parzellen:   98
Hindernisparzellen:          2
versiegelte Parzellen:       97
nicht versiegelte Parzellen: 1
```

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

Greedy-Strategie:

```
   1 2 3 4 5 6 7 8 9 10
 1   H V < V < V < V <
 2 H V < ^ V ^ V ^ V ^
 3 Z > V ^ V ^ V ^ V ^
 4 ^ < V ^ V ^ V ^ V ^
 5 > ^ V ^ V ^ V ^ V ^
 6 ^ < V ^ V ^ V ^ V ^
 7 > ^ V ^ V ^ V ^ V ^
 8 ^ < V ^ V ^ V ^ V ^
 9 > ^ V ^ V ^ V ^ V ^
10 ^ < < ^ < ^ < ^ < ^
```

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

```
zu versiegelnde Parzellen:   98
Hindernisparzellen:          2
versiegelte Parzellen:       97
nicht versiegelte Parzellen: 1
```

4.5.1.4 Eigene Not-OK-Testfälle

Die nachfolgenden Fälle wurden sämtlich mit Erfolg durchgeführt, sodass die jeweils aufgeführten erwarteten Ergebnisse mit den erhaltenen übereinstimmen.

Keine Parameter

Beschreibung: Programmaufruf ohne Übergabeparameter
Ergebnis: Kontrollierter Programmabbruch. Ausgabe der Fehlermeldung: „Please execute the program via 'java -jar 20510_solver.jar [.]endingInputFiles pathToFileOrFolder [[pathToFileOrFolder2] [pathToFileOrFolder3] ...]'" auf die Konsole.

Datei nicht vorhanden

Dateiname: notexisting.in
Beschreibung: Formal korrekter Übergabeparameter, der allerdings eine nicht existente Datei enthält.
Ergebnis: Kontrollierter Programmabbruch. Ausgabe der Fehlermeldung: „File or dir could not be found: notexisting.in" auf die Konsole

Leere Eingabedatei

Dateiname: nok_empty.in
Beschreibung: Aufruf mit leerer Eingabedatei
Ergebnis: Ausgabe der Fehlermeldung: „Validation error: File does not contain at least 2 lines (not counting comments): …" in die Ausgabedatei

Ausgabedatei:

```
Validation error: File does not contain at least 2 lines (not counting
comments): D:\projects\IHK_softwaresystem_14\target\test\.\nok\nok_empty.in
```

Abmessungen < 1

Dateiname: nok_less_than_1.in
Beschreibung: Abmessungen der Parkettfläche enthalten einen Wert 0
Ergebnis: Ausgabe der Fehlermeldung: „Validation error: Invalid format for area measures: '8 0' format must be 'number number' and number in [1..10]" in die Ausgabedatei

Eingabedatei:

```
; NOK-Fall: Abmessungen < 1
8 0
; Startparzelle
8 0
; Hindernisse
```

Ausgabedatei:

```
Validation error: Invalid format for area measures: '8 0' format must be
'number number' and number in [1..10]
```

Abmessungen > 10

Dateiname: nok_more_than_10.in
Beschreibung: Abmessungen der Parkettfläche enthalten einen Wert 11
Ergebnis: Ausgabe der Fehlermeldung: „Validation error: Invalid format for area measures: '8 11' format must be 'number number' and number in [1..10]" in die Ausgabedatei

Eingabedatei:

```
; NOK-Fall: Abmessungen > 10
8 11
; Startparzelle
8 8
; Hindernisse
7 7 6 6
```

Ausgabedatei:

```
Validation error: Invalid format for area measures: '8 11' format must be
'number number' and number in [1..10]
```

Hindernis nicht im gültigen Bereich

Dateiname: nok_obstacle_out_of_range.in
Beschreibung: Das Hindernis liegt nicht in der Parkettfläche
Ergebnis: Ausgabe der Fehlermeldung: „Validation error: obstacle coordinate not in area: x=9" in die Ausgabedatei

Eingabedatei:

```
; NOK-Fall: Hindernis nicht im gültigen Bereich
8 8
; Startparzelle
8 8
; Hindernisse
7 7 7 9
```

Ausgabedatei:

```
Validation error: obstacle coordinate not in area: x=9
```

Startpunkt auf Hindernis

Dateiname: nok_start_on_obstacle.in
Beschreibung: Der Startpunkt ist identisch mit einem Hindernis
Ergebnis: Ausgabe der Fehlermeldung: „Validation error: cannot place obstacle on start: y=8 x=8" in die Ausgabedatei

Eingabedatei:

```
; NOK-Fall: Startpunkt auf Hindernis
8 8
; Startparzelle
8 8
; Hindernisse
7 7 8 8
```

Ausgabedatei:

```
Validation error: cannot place obstacle on start: y=8 x=8
```

Startpunkt nicht im gültigen Bereich

Dateiname: nok_start_out_of_range.in
Beschreibung: Der Startpunkt liegt nicht in der Parkettfläche
Ergebnis: Ausgabe der Fehlermeldung: „Validation error: start coordinate not in area: x=9" in die Ausgabedatei

Eingabedatei:

```
; NOK-Fall: Startpunkt nicht im gültigen Bereich
8 8
; Startparzelle
8 9
; Hindernisse
7 7 6 6
```

Ausgabedatei:

```
Validation error: start coordinate not in area: x=9
```

Fehlerhafte Eingabedatei führt nicht zu Programmabbruch

Beschreibung: Aufruf des Programms mit Übergabe verschiedener Eingabedateien, von denen mindestens eine Fehler hervorruft.
Zweck: Dieser Test stellt sicher, dass das Programm im Falle einer fehlerhaften Eingabedatei nicht abbricht, sondern mit den anderen Dateien fortfährt.
Ergebnis: Das Programm zeigt das erwartete Verhalten beim Ausführen der Testfälle im Ordner „nok".

4.6 Zusammenfassung

Das Problem der Aufgabenstellung entspricht der Suche nach einem Hamiltonweg in einem Graphen. Dies ist ein Weg, der jeden Knoten (hier Parzelle) genau einmal enthält. Außerdem liegt der Spezialfall vor, dass ein Startknoten festgelegt ist. Dieses Graphenproblem ist NP-schwer, d.h. es ist nur in exponentieller Laufzeit lösbar (Problemgröße n steht bei der Laufzeitberechnung im Exponenten), ein Lösungskandidat lässt sich hingegen in polynomieller Laufzeit überprüfen. (Wilke, 2007)
Dies hat zur Folge, dass ein zu entwickelnder Lösungsalgorithmus im ungünstigen Fall immer diese Laufzeit benötigen wird. Was hingegen optimiert werden kann, ist die Reihenfolge in der die möglichen Wege durch den Graphen ausprobiert werden.
Das entwickelte Programm enthält zwei Strategien, die diese Reihenfolge determinieren. Die Uhrzeiger-Strategie stellt eine sehr einfache Methode dar, den nächsten Knoten zu bestimmen und kann sehr effizient programmiert werden, sodass jeder Entscheidungsschritt nur wenig Zeit in Anspruch nimmt. Allerdings ist die Uhrzeiger-Strategie in der Wahl des nächsten Knotens nicht sehr geschickt, was dazu führt, dass die bestmögliche Lösung tendenziell später gefunden wird.
Die Greedy-Strategie ist aufwändiger in der Entscheidungsfindung, da für jeden Nachbarn wiederum jeder Nachbar identifiziert und gezählt werden muss. Allerdings trifft die Greedy-Strategie tendenziell bessere Entscheidungen. Sie entspricht der grundlegenden Strategie, mit der auch das sogenannte Springerproblem gelöst werden

kann, nämlich immer den Nachbarknoten zu wählen, von dem es am wenigsten Möglichkeiten gibt, weiter zu laufen.
Dieser Unterschied findet sich auch in den ausgeführten Testfällen wieder, bei denen die Greedy-Strategie, insbesondere für große n, schneller eine gute Lösung findet, als die Uhrzeiger-Strategie. Das Maß für diese Geschwindigkeit ist die Anzahl der Sackgassen, die untersucht werden, bis die bestmögliche Lösung gefunden wurde:

Testfall	Anzahl Parzellen	Anzahl Hindernis-se	Uhrzeiger-Strategie Anzahl Sackgassen bis optimaler Weg	Greedy-Strategie Anzahl Sackgassen bis optimaler Weg
ihk1.in	25	0	1	1
ihk2.in	26	4	112	88
ihk3.in	12	3	2	3
ok_format_whitespaces.in	13	3	5	1
ok_maximalsize_no_obstacles.in	100	0	1	1
ok_mediumsize_no_obstacles.in	16	0	28	1
ok_mediumsize_obstacles_impossible.in	14	2	40	2
ok_mediumsize_obstacles_optimal.in	22	2	1	1
performance/ok_mediumsize_obstacles_impossible.in	34	2	1	1
performance/ok_mediumsize_obstacles_impossible2.in	34	2	36	2
ok_maximalsize_obstacles_impossible.in	98	2	287	1

Die Fälle, in denen die Greedy-Strategie schneller ist, als die Uhrzeiger-Strategie sind hier grün markiert, der Umkehrfall rot.

4.7 Ausblick

Das Softwaresystem bietet an den nachfolgenden Punkten Verbesserungspotential:

4.7.1 Verbesserung des Hauptalgorithmus

Es ist möglich, die Abbruchbedingung der Tiefensuche zu optimieren: Im aktuellen Zustand bricht die Tiefensuche ab, wenn die Anzahl der Knoten in der Lösung mit der Anzahl der freien Parzellen im Parkett übereinstimmt. Gibt es jedoch Parzellen, die durch Hindernisse isoliert sind, sodass sie vom Startpunkt unerreichbar sind, könnten diese von der Anzahl der freien Parzellen abgezogen werden. Die Anzahl der nicht erreichbaren Parzellen könnte zu Beginn per Breitensuche ermittelt werden.

4.7.2 Parallelisierung

Um die Bedienbarkeit des Programms zu verbessern, könnte die Ausführung der beiden Strategien parallelisiert werden. Dies würde dazu führen, dass für beide Strategien schnell eine erste Lösung vorliegt und auch in der Ausgabedatei niedergelegt wird. Dabei müsste allerdings sichergestellt werden, dass das Schreiben in die Ausgabedatei immer seriell abläuft.

Außerdem wäre es möglich, eine Anzahl von Kindbäumen aus der Tiefensuche parallel untersuchen zu lassen. D.h. dass man beispielsweise die vom Startpunkt aus direkt erreichbaren Parzellen als neue Startpunkte und den eigentlichen Startpunkt als Hindernis definiert. Anschließend könnte man die Tiefensuche parallel von den neuen Startpunkten aus beginnen lassen. Dieses Vorgehen führt auf Multi-Kern-Systemen zu einem schnelleren Finden der Lösung. Allerdings kann auch mit dieser Vorgehensweise die Problemklasse (NP) nicht reduziert werden, sodass das System für große n dennoch nicht in absehbarer Zeit terminieren wird.

4.7.3 Probleme mit n > 100

Das Softwaresystem ist auf Probleme beschränkt, die höchstens 100 (10x10) Parzellen enthalten. In der Praxis gilt diese Beschränkung allerdings nicht. Wenn das Softwaresystem auch für beliebig große Parkettflächen eingesetzt werden soll, dann muss von einer rekursiven Implementierung abgesehen werden, da es sonst zu einem Stack-Überlauf kommt: Jeder Rekursionsaufruf erzeugt einen Eintrag auf dem Call Stack, der allerdings eine feste Größe hat.
Die Anzahl der rekursiven Aufrufe entspricht der maximalen Baumtiefe der Rekursion, bzw. hier der maximalen Weg-Länge (=Weglänge der bestmöglichen Lösung).

5 Anhang

5.1 Literaturverzeichnis

IHK. (Sommer 2014). *Aufgabenbogen Abschlussprüfung - Entwicklung eines Softwaresystems.*

Wilke, S. (2007). *Hamilton-Pfad auf Gittergraphen.* Abgerufen am 15. 05 2014 von http://www.inf.fu-berlin.de/lehre/WS07/AlgorithmenSeminar/Hamilton-Pfad-in-Gittergraphen.pdf

5.2 Quelltext

5.2.1 Package start

Main

```
package start;

import io.Config;
import io.InputFileReader;
import io.Log;

import java.io.File;
import java.util.Arrays;

import control.Controller;

/**
 * contains global entry point of the application
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class Main {

	private static final Log log = Log.getInstance();

	/**
	 * global entry point of the application
	 *
	 * @param args
	 *            passed to the application
	 */
	public static void main(String[] args) {
		// 1. Param = fileending, other params = filepaths
		if (args.length < 2) {
			log.error("Please execute the program via 'java -jar
20510_solver.jar [.]endingInputFiles pathToFileOrFolder [[pathToFileOrFolder2]
[pathToFileOrFolder3] ...]'");
```

```
                System.exit(-1);
            }
            File[] inputfiles = InputFileReader.getInputFiles(args[0],
                    Arrays.copyOfRange(args, 1, args.length));
            boolean switchAlgorithms =
Config.getBooleanValue("switchAlgorithms",
                    false);
            Controller ctrl = new Controller(switchAlgorithms);
            for (File file : inputfiles) {
                ctrl.processFile(file);
            }
        }
}
```

5.2.2 Package control

Controller

```
package control;

import io.InputFileReader;
import io.Log;
import io.OutputFileWriter;

import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import model.Area;
import model.Cell;
import model.Direction;
import error.CheckedSimulationException;
import error.SimulationException;

/**
 * controls the logical process of the programm
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class Controller {

        private static final Log log = Log.getInstance();
        private Area area;
        private Cell start;
        private List<INextCellAlgorithm> algorithms;
        private Map<INextCellAlgorithm, List<Cell>> solutions;
        private Map<INextCellAlgorithm, Boolean> bestSolutionFound;
```

```
	private Map<INextCellAlgorithm, Integer> deadEndCount;
	private String errorToFile;
	private String outFileName;
	private boolean switchAlgorithms;

	/**
	 * @param switchAlgorithms
	 *
	 */
	public Controller(boolean switchAlgorithms) {
		this.switchAlgorithms = switchAlgorithms;
		init();
	}

	// call this method before each file processing
	private void init() {
		errorToFile = "";
		algorithms = new ArrayList<INextCellAlgorithm>();
		if (switchAlgorithms) {
			algorithms.add(new GreedyNextCellAlgorithm());
			algorithms.add(new ClockwiseNextCellAlgorithm());
		} else {
			algorithms.add(new ClockwiseNextCellAlgorithm());
			algorithms.add(new GreedyNextCellAlgorithm());
		}
		bestSolutionFound = new HashMap<INextCellAlgorithm, Boolean>();
		solutions = new HashMap<INextCellAlgorithm, List<Cell>>();
		deadEndCount = new HashMap<INextCellAlgorithm, Integer>();
	}

	/**
	 * processes an input File. The solution to the problem is written into a
	 * corresponding output file.
	 *
	 * @param in
	 *            input file which contains the problem
	 */
	public void processFile(File in) {
		init(); // reset state caused by past files
		log.info("process file: " + in.getAbsolutePath());
		outFileName = in.getAbsolutePath() + ".out";
		// get internal data representation
		try {
			area = InputFileReader.getModel(in);
			start = area.getStart();
		} catch (CheckedSimulationException e) {
			log.error("Error during validation of input file!", e);
			errorToFile = "Validation error: " + e.getMessage();
		}
		if (!errorToFile.isEmpty()) {
			// an error occurred during validation phase
			try {
				OutputFileWriter.writeOutError(errorToFile,
outFileName);
			} catch (CheckedSimulationException e) {
```

```
                        log.error(e.getMessage());
                }
        } else {
                // search solution
                for (INextCellAlgorithm algorithm : algorithms) {
                        bestSolutionFound.put(algorithm, Boolean.FALSE);
                }
                for (INextCellAlgorithm algorithm : algorithms) {
                        process(algorithm);
                }
                try {
                        // write output file
                        OutputFileWriter.writeOutSolutions(outFileName, area,
                                        solutions, algorithms);
                } catch (CheckedSimulationException e) {
                        log.error(e.getMessage());
                }
        }
}

// start of recursion
private void process(INextCellAlgorithm algorithm) {
        // do some initial stuff
        area.resetArea();
        List<Cell> path = new ArrayList<Cell>();
        path.add(start);
        log.info("start search for solution with algorithm: "
                        + algorithm.toString());
        // start recursion
        processRec(algorithm, path);
        log.debug("Solution found for algorithm: " + algorithm.toString());
        // only perform these heavy operations if logging is enabled
        if (log.isEnabled()) {
                List<Cell> solution = solutions.get(algorithm);
                String sPath = "";
                for (Cell cell : solution) {
                        sPath += " -> " + cell;
                }
                log.debug(sPath);
                log.debug(area.toString(solution));
        }
}

private void processRec(INextCellAlgorithm algorithm, List<Cell> path) {
        Cell current = path.get(path.size() - 1);
        current.setVisited(true);
        if (Boolean.TRUE.equals(bestSolutionFound.get(algorithm))) {
                return;
        } else if (current.getAvailNeighbours().isEmpty()) {
                propagateSolution(algorithm, path);
                return;
        } else {
                // only perform this heavy operation (area.toString()) if
logging is
                // enabled
```

```
                    if (log.isEnabled()) {
                          log.debug(area.toString(path));
                    }

                    Set<Cell> ignoredCells = new HashSet<Cell>();
                    for (int i = 0; i < 4; i++) {
                          Cell next = algorithm.getNextCell(current,
ignoredCells);
                          if (next != null) {
                                // add next Cell to end of path
                                path.add(next);
                                // recursion
                                processRec(algorithm, path);
                                // remove next Cell from end of path
                                path.remove(path.size() - 1);
                                // add next Cell to ignored cells
                                ignoredCells.add(next);
                                next.setVisited(false);
                          }
                    }
              }
        }

        private void propagateSolution(INextCellAlgorithm algorithm, List<Cell>
path) {
              int countVisited = path.size();
              int lastDeadEndCount = deadEndCount.get(algorithm) == null ? 0
                          : deadEndCount.get(algorithm);
              deadEndCount.put(algorithm, lastDeadEndCount + 1);
              log.debug(area.toString(path));
              log.debug("propagate Solution");
              if (solutions.get(algorithm) == null
                          || countVisited > solutions.get(algorithm).size()) {
                    log.info("save new Solution as best solution");
                    log.info("current count of reached dead ends: "
                                + (lastDeadEndCount + 1));
                    solutions.put(algorithm, Cell.clonePath(path));
                    if (countVisited == area.getCellCount() -
area.getObstacleCount()) {
                          log.debug("new solution is optimal");
                          bestSolutionFound.put(algorithm, true);
                    }
                    try {
                          OutputFileWriter.writeOutSolutions(outFileName, area,
                                      solutions, algorithms);
                    } catch (CheckedSimulationException e) {
                          log.error(e.getMessage());
                    }
              }
              log.debug("");
        }

        /**
         * converts a path ({@link List}<{@link Cell}>) into a routeplan (
         * {@link List}<{@link Cell}>). In the path all sequent nodes must be
```

```
         * neighbours. The routeplan only contains nodes, where the direction of
the
         * path changes (and first and last node).
         *
         * @param path
         * @return routeplan
         * @throws SimulationException
         *             if sequent Nodes from the path are not neighbours
         */
        public static List<Cell> convertToRoutePlan(List<Cell> path) {
              List<Cell> plan = new ArrayList<Cell>();
              if (path.size() < 1) {
                    return plan;
              }
              Cell lastCell = path.get(0);
              if (path.size() < 2) {
                    plan.add(lastCell);
                    return plan;
              }
              Direction lastDirection =
lastCell.getDirectionOfNeighbour(path.get(1));
              for (int i = 1; i < path.size(); i++) {
                    Direction currentDirection =
path.get(i).getDirectionOfNeighbour(
                                lastCell);
                    if (currentDirection != lastDirection) {
                          plan.add(lastCell);
                          lastDirection = currentDirection;
                    }
                    lastCell = path.get(i);
              }
              plan.add(lastCell);
              return plan;
        }
}
```

INextCellAlgorithm

```
package control;

import java.util.Set;

import model.Cell;

/**
 * provides methods for priorisation algorithms, which choose the next neighbour
 * according to their strategy
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public interface INextCellAlgorithm {

      /**
```

```
         * determines the next {@link Cell} on the path according to the
         * implementing algorithm. Only neighbours of the parameter from are taken
         * into account. All cells passed in the second parameter are ignored and
         * will never be returned by this method. If there is no next cell
         * remaining, the method returns <code>null</code>.
         *
         * @param from
         *            {@link Cell} from where the next cell should be determined
         * @param except
         *            {@link Set}<{@link Cell}> to ignore
         * @return next cell on the path or <code>null</code>
         */
        public Cell getNextCell(Cell from, Set<Cell> except);

        /**
         * @return the name of the implementing algorithm
         */
        public String toString();

}
```

ClockwiseNextCellAlgorithm

```
package control;

import java.util.Set;

import model.Cell;
import model.Direction;

/**
 * Implementation of {@link INextCellAlgorithm}. Chooses the next neighbour
 * {@link Cell} based on the following priorisation: top before right before
 * bottom before left
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class ClockwiseNextCellAlgorithm implements INextCellAlgorithm {

        @Override
        public Cell getNextCell(Cell from, Set<Cell> except) {
                Cell next = from.getNeighbourIfAvailable(Direction.TOP);
                if (next != null && !except.contains(next)) {
                        return next;
                }
                next = from.getNeighbourIfAvailable(Direction.RIGHT);
                if (next != null && !except.contains(next)) {
                        return next;
                }
                next = from.getNeighbourIfAvailable(Direction.BOTTOM);
                if (next != null && !except.contains(next)) {
                        return next;
                }
```

```
			next = from.getNeighbourIfAvailable(Direction.LEFT);
			if (next != null && !except.contains(next)) {
				return next;
			}
			return null;
		}

		@Override
		public String toString() {
			return "Uhrzeiger-Strategie";
		}

}
```

GreedyNextCellAlgorithm

```
package control;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

import model.Cell;
import model.Direction;

/**
 * Implementation of {@link INextCellAlgorithm}. Chooses the next neighbour
 * {@link Cell} based on the degree of freedom of the direct neighbours. The
 * neighbour with the lowest degree of freedom is chosen.
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class GreedyNextCellAlgorithm implements INextCellAlgorithm {

		@Override
		public Cell getNextCell(Cell from, Set<Cell> except) {
			Set<Cell> neighbours = from.getAvailNeighbours();
			// unignored cells of minimum freedom
			List<Cell> uineigh = new ArrayList<Cell>();
			int freedomMin = 4;
			for (Cell cell : neighbours) {
				if (!except.contains(cell)) {
					if (cell.getAvailNeighbours().size() == freedomMin) {
						uineigh.add(cell);
					} else if (cell.getAvailNeighbours().size() <
freedomMin) {
						uineigh.clear();
						freedomMin = cell.getAvailNeighbours().size();
						uineigh.add(cell);
					}
				}
			}
			// now uineigh contains only the neighbours of minimum freedom
```

```
			for (Cell cell : uineigh) {
				if (from.getDirectionOfNeighbour(cell) == Direction.TOP) {
					return cell;
				}
			}
			for (Cell cell : uineigh) {
				if (from.getDirectionOfNeighbour(cell) == Direction.RIGHT) {
					return cell;
				}
			}
			for (Cell cell : uineigh) {
				if (from.getDirectionOfNeighbour(cell) == Direction.BOTTOM) {
					return cell;
				}
			}
			for (Cell cell : uineigh) {
				if (from.getDirectionOfNeighbour(cell) == Direction.LEFT) {
					return cell;
				}
			}
			return null;
		}

		@Override
		public String toString() {
			return "Greedy-Strategie";
		}

}
```

5.2.3 Package model

Area

```
package model;

import java.util.List;

import error.CheckedSimulationException;
import error.SimulationException;

/**
 * model class that contains the parquet area in different internal
 * representations: as a two-dimensional {@link Cell} and as a linked graph. All
 * public methods refer to the top left cell with index (1,1)
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class Area {

		private Cell start;
```

```java
private Cell[][] areaArray;
private int obstacleCount;
private int cellCount;

/**
 * Constructor which creates an area of the specified size whithout any
 * obstacles and the startpoint at (1,1)
 *
 * @param y
 * @param x
 */
public Area(int y, int x) {
	if (y < 1 || x < 1) {
		throw new SimulationException("Area must not be empty!");
	}
	initArea(y, x);
	cellCount = y * x;
	obstacleCount = 0;
	// default
	start = areaArray[0][0];
}

private void initArea(int ylength, int xlength) {
	areaArray = new Cell[ylength][xlength];
	// create objects
	for (int y = 0; y < areaArray.length; y++) {
		for (int x = 0; x < areaArray[y].length; x++) {
			areaArray[y][x] = new Cell(y, x, false);
		}
	}
	// links objects
	for (int y = 0; y < areaArray.length; y++) {
		for (int x = 0; x < areaArray[y].length; x++) {
			if (x + 1 < areaArray[y].length) {
				areaArray[y][x].setRight(areaArray[y][x + 1]);
			}
			if (y + 1 < areaArray.length) {
				areaArray[y][x].setBottom(areaArray[y + 1][x]);
			}
		}
	}
}

/**
 * @return xLength*yLength
 */
public int getCellCount() {
	return cellCount;
}

/**
 * @return count of obstacles in the area
 */
public int getObstacleCount() {
	return obstacleCount;
```

```
	}

	/**
	 * sets the visited state back to the initial state. All obstacles are set
	 * to visited=true
	 */
	public void resetArea() {
		for (Cell[] ca : areaArray) {
			for (Cell c : ca) {
				c.setVisited(c.isObstacle());
			}
		}
	}

	/**
	 * sets the start cell
	 *
	 * @param y
	 *            index starts at 1
	 * @param x
	 *            index starts at 1
	 * @throws CheckedSimulationException
	 *             if y,x out of range, or if the start is placed onto an
	 *             obstacle
	 */
	public void setStart(int y, int x) throws CheckedSimulationException {
		if (y > areaArray.length) {
			throw new CheckedSimulationException(
					"start coordinate not in area: y=" + y);
		}
		if (x > areaArray[0].length) {
			throw new CheckedSimulationException(
					"start coordinate not in area: x=" + x);
		}
		if (areaArray[y - 1][x - 1].isObstacle()) {
			throw new CheckedSimulationException(
					"cannot place start on obstacle: y=" + y + " x="
+ x);
		}
		start = areaArray[y - 1][x - 1];
	}

	/**
	 * converts the defined area into obstacles. The former state of the
	 * converted cells is not taken into account.
	 *
	 * @param yMin
	 *            smallest y-Extension of the obstacle (index starts at 1)
	 * @param xMin
	 *            smallest x-Extension of the obstacle (index starts at 1)
	 * @param yMax
	 *            greatest y-Extension of the obstacle (index starts at 1)
	 * @param xMax
	 *            greatest x-Extension of the obstacle (index starts at 1)
	 * @throws CheckedSimulationException
```

```
	 */
	public void setObstacle(int yMin, int xMin, int yMax, int xMax)
			throws CheckedSimulationException {
		if (yMin > yMax || xMin > xMax || yMin < 1 || xMin < 1) {
			throw new SimulationException("coordinate out of range");
		}
		if (yMax > areaArray.length) {
			throw new CheckedSimulationException(
					"obstacle coordinate not in area: y=" + yMax);
		}
		if (xMax > areaArray[0].length) {
			throw new CheckedSimulationException(
					"obstacle coordinate not in area: x=" + xMax);
		}

		for (int y = yMin - 1; y < yMax; y++) {
			for (int x = xMin - 1; x < xMax; x++) {
				if (!areaArray[y][x].isObstacle()) {
					if (start == areaArray[y][x]) {
						throw new CheckedSimulationException(
								"cannot place obstacle on
start: y=" + (y + 1)
+ " x=" + (x +
1));
					}
					obstacleCount++;
					areaArray[y][x].setObstacle(true);
				}
			}
		}
	}

	/**
	 * @return start cell
	 */
	public Cell getStart() {
		return start;
	}

	/**
	 * generates a String representation of the area, depending on the
specified
	 * path.
	 *
	 * @param path
	 *            through the area
	 * @return String representation
	 */
	public String toString(List<Cell> path) {
		char[][] table = new char[areaArray.length][areaArray[0].length];
		// set obstacles
		for (int y = 0; y < table.length; y++) {
			for (int x = 0; x < table[y].length; x++) {
				if (areaArray[y][x].isObstacle()) {
					table[y][x] = 'H';
```

```
                        } else {
                              table[y][x] = ' ';
                        }
                  }
            }
            // traverse and write path
            table[path.get(0).getY()][path.get(0).getX()] = 'S';
            for (int i = 0; i < path.size() - 1; i++) {
                  Cell current = path.get(i);
                  Cell next = path.get(i + 1);
                  switch (current.getDirectionOfNeighbour(next)) {
                  case BOTTOM:
                        table[current.getY()][current.getX()] = 'V';
                        break;
                  case LEFT:
                        table[current.getY()][current.getX()] = '<';
                        break;
                  case RIGHT:
                        table[current.getY()][current.getX()] = '>';
                        break;
                  case TOP:
                        table[current.getY()][current.getX()] = '^';
                        break;
                  }
            }
            // write final destination
            if (table[path.get(path.size() - 1).getY()][path.get(path.size() -
1)
                        .getX()] != 'S') {
                  table[path.get(path.size() - 1).getY()][path.get(path.size() -
1)
                              .getX()] = 'Z';

            }
            // create String representation
            StringBuilder sb = new StringBuilder("  ");
            for (int x = 0; x < table[0].length; x++) {
                  sb.append(" " + (x + 1));
            }
            for (int y = 0; y < table.length; y++) {
                  sb.append(System.lineSeparator());
                  if (y < 9) {
                        sb.append(" ");
                  }
                  sb.append(y + 1);
                  for (int x = 0; x < table[y].length; x++) {
                        if (x > 9) {
                              sb.append(" ");
                        }
                        sb.append(" " + table[y][x]);
                  }
            }
            sb.append(System.lineSeparator());
            return sb.toString();
      }
```

```
}
```

Cell

```
package model;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import error.SimulationException;

/**
 * model class which represents a cell of the parquet.
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class Cell {

	private Cell left;
	private Cell right;
	private Cell top;
	private Cell bottom;
	private Set<Cell> neighbours;
	private int y;
	private int x;
	private boolean obstacle;
	private boolean visited;

	/**
	 * @return y-coordinate of the cell in the parquet (index starts by 0)
	 */
	public int getY() {
		return y;
	}

	/**
	 * @return x-coordinate of the cell in the parquet (index starts by 0)
	 */
	public int getX() {
		return x;
	}

	/**
	 * creates a new Cell object.
	 *
	 * @param y
	 *            y-coordinate of the cell in the parquet (index starts by 0)
	 * @param x
	 *            x-coordinate of the cell in the parquet (index starts by 0)
	 * @param obstacle
	 */
```

```
public Cell(int y, int x, boolean obstacle) {
	this.y = y;
	this.x = x;
	this.obstacle = obstacle;
	this.visited = obstacle;
	this.neighbours = new HashSet<Cell>();
}

/**
 * @param obstacle
 */
protected void setObstacle(boolean obstacle) {
	this.obstacle = obstacle;
}

/**
 * @return if this is an obstacle
 */
public boolean isObstacle() {
	return obstacle;
}

/**
 * links a cell as right neighbour. The this-object is also saved in the
 * right neighbour as left neighbour.
 *
 * @param right
 *            neighbour to link
 */
protected void setRight(Cell right) {
	this.right = right;
	neighbours.add(right);
	right.left = this;
	right.neighbours.add(this);
}

/**
 * links a cell as bottom neighbour. The this-object is also saved in the
 * bottom neighbour as top neighbour.
 *
 * @param bottom
 */
protected void setBottom(Cell bottom) {
	this.bottom = bottom;
	neighbours.add(bottom);
	bottom.top = this;
	bottom.neighbours.add(this);
}

/**
 * sets the internal value to the passed value. This value remains
 * unmodified until the next call of this method.
 *
 * @param visited
 */
```

```
	public void setVisited(boolean visited) {
		this.visited = visited;
	}

	/**
	 * @return if the cell has been visited
	 */
	public boolean isVisited() {
		return visited;
	}

	/**
	 *
	 * @return all neighbour cells that are neither obstacles nor visited
	 */
	public Set<Cell> getAvailNeighbours() {
		Set<Cell> availnb = new HashSet<Cell>();
		for (Cell cell : neighbours) {
			if (!cell.obstacle && !cell.visited) {
				availnb.add(cell);
			}
		}
		return availnb;
	}

	/**
	 * @param neighbour
	 * @return {@link Direction} where the passed Cell is located from the
	 *         perspective of this Cell
	 * @throws SimulationException
	 *             if the passed Cell is not a neighbour of this cell
	 */
	public Direction getDirectionOfNeighbour(Cell neighbour) {
		if (neighbour == top) {
			return Direction.TOP;
		} else if (neighbour == right) {
			return Direction.RIGHT;
		} else if (neighbour == bottom) {
			return Direction.BOTTOM;
		} else if (neighbour == left) {
			return Direction.LEFT;
		}
		throw new SimulationException(
				"illegal operation: Cell is not a neighbour");
	}

	/**
	 * checks whether the neighbour in the passed {@link Direction} exists and
	 * is neither visited nor an obstacle. If these conditions apply the
	 * neighbour is returned, otherwise null is returned.
	 *
	 * @param direction
	 * @return the neighbour if available, if not returns null
	 */
	public Cell getNeighbourIfAvailable(Direction direction) {
```

```
		Cell neighbour = null;
		switch (direction) {
		case TOP:
			neighbour = top;
			break;
		case RIGHT:
			neighbour = right;
			break;
		case BOTTOM:
			neighbour = bottom;
			break;
		case LEFT:
			neighbour = left;
			break;
		}
		if (neighbour != null && !neighbour.obstacle && !neighbour.visited)
{
			return neighbour;
		} else {
			return null;
		}
	}

	/**
	 * creates a new {@link List}<{@link Cell}> containing all Cells that were
	 * passed. The Cells themselves are not copied, only a reference to the
	 * existing objects is saved in the returned List.
	 *
	 * @param path
	 * @return new List object
	 */
	public static List<Cell> clonePath(List<Cell> path) {
		List<Cell> clone = new ArrayList<Cell>();
		clone.addAll(path);
		return clone;
	}

	/*
	 * (non-Javadoc)
	 *
	 * @see java.lang.Object#toString()
	 */
	@Override
	public String toString() {
		return "(" + (y + 1) + "," + (x + 1) + ")";
	}

	/**
	 * converts a given path into a String representation of the following
	 * format: </br> y1,x1 / y2,x2 / ... / yn,xn
	 *
	 * @param path
	 *            List of Cells
	 * @return String representation
	 */
```

```
	public static String getAsString(List<Cell> path) {
		StringBuilder sb = new StringBuilder();
		sb.append(path.get(0).toString());
		for (int i = 1; i < path.size(); i++) {
			sb.append(" / " + path.get(i).toString());
		}
		sb.append(System.lineSeparator());
		// delete braces from result
		return sb.toString().replaceAll("\\(|\\)", "");
	}
}
```

Direction

```
package model;

/**
 * enumeration which represents the directions in which {@link Cell} objects can
 * relate to each other.
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public enum Direction {

	TOP, RIGHT, BOTTOM, LEFT

}
```

5.2.4 Package io

InputFileReader

```
package io;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.FilenameFilter;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

import model.Area;
import error.CheckedSimulationException;

/**
 * class to read problems from input files.
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
```

```
public class InputFileReader {

	private static final Log log = Log.getInstance();
	/**
	 * matches greedyly all tabulator- and whitespaces
	 */
	public static final String regexWhiteSpaces = "(\\t|\\s)++";
	/**
	 * matches a String which contains two numbers. E.g. '5 5'
	 */
	public static final String regexTwoNumbers = "([1-9]|10)"
			+ regexWhiteSpaces + "([1-9]|10)";
	/**
	 * matches a String which contains four numbers. E.g. '5 5 5 5'
	 */
	public static final String regexFourNumbers = "([1-9]|10)"
			+ regexWhiteSpaces + "([1-9]|10)" + regexWhiteSpaces + "([1-
9]|10)"
			+ regexWhiteSpaces + "([1-9]|10)";

	/**
	 * returns an array of {@link File} objects matching the input parameters.
	 * The paths parameter can contain paths to files. In this case the
	 * corresponding files are added to the result array. The parameter can
also
	 * contain paths to directories. In this case the directory is scanned
	 * (non-recursively) for files with the specified ending. Matching files
are
	 * also added to the result array.
	 *
	 * @param fileending
	 *            of files which should be taken into account
	 * @param paths
	 *            to files and to directories in which files should be looked
up
	 * @return
	 */
	public static File[] getInputFiles(String fileending, String[] paths) {
		// append leading dot to fileending if not provided
		if (!fileending.startsWith(".")) {
			fileending = "." + fileending;
		}
		// for inner class FilenameFilter final keyword
		final String finalFileEnding = fileending;
		List<String> absolutepaths = new ArrayList<String>();
		for (String p : paths) {
			File f = new File(p);
			if (!f.exists()) {
				log.error("File or dir could not be found: " + p);
				continue;
			}
			if (f.isFile()) {
				absolutepaths.add(f.getAbsolutePath());
			} else {
				FilenameFilter ff = new FilenameFilter() {
```

```
                                @Override
                                public boolean accept(File dir, String name) {
                                        return name.endsWith(finalFileEnding);
                                }
                        };
                        File[] subfiles = f.listFiles(ff);
                        for (File file : subfiles) {
                                absolutepaths.add(file.getAbsolutePath());
                        }
                }
        }
        File[] files = new File[absolutepaths.size()];
        int i = 0;
        for (String p : absolutepaths) {
                log.debug("add input file " + p);
                files[i] = new File(p);
                i++;
        }
        return files;
    }

    /**
     * transforms an input file into the internal data model {@link Area}.
     * Validates the correct format of the input file's content.
     *
     * @param in
     * @return data model
     * @throws CheckedSimulationException
     *             if the input file's content is not according to the
     *             specification
     */
    public static Area getModel(File in) throws CheckedSimulationException {
        List<String> lines = getProductiveLines(in);
        if (lines.size() < 2) {
                throw new CheckedSimulationException(
                                "File does not contain at least 2 lines (not
counting comments): "
                                                + in.getAbsolutePath());
        }
        Area area = initArea(lines.get(0));
        setStart(area, lines.get(1));
        for (int i = 2; i < lines.size(); i++) {
                setObstacle(area, lines.get(i));
        }
        return area;
    }

    /**
     * sets the first cell coordinates in a given area. <br>
     * start should match format 'y x'
     *
     * @param area
     * @param start
     * @throws CheckedSimulationException
     *             if format is incorrect
```

```
	 */
	private static void setStart(Area area, String start)
			throws CheckedSimulationException {
		if (!(start.matches(InputFileReader.regexTwoNumbers))) {
			throw new CheckedSimulationException(
					"Invalid format for start coordinates: '"
							+ start
							+ "' format must be 'number number'
and number in [1..10]");
		}
		String[] snumbers = start.split(regexWhiteSpaces);
		int y;
		int x;
		y = Integer.parseInt(snumbers[0]);
		x = Integer.parseInt(snumbers[1]);
		area.setStart(y, x);
	}

	/**
	 * extracts all Lines from a file omitting leading and trailing
whitespaces.
	 * lines, which first character is a ';' are ignored as well as empty lines
	 *
	 * @param f
	 * @return textlines as {@link List}
	 * @throws CheckedSimulationException
	 */
	public static List<String> getProductiveLines(File f)
			throws CheckedSimulationException {
		List<String> lines = new ArrayList<String>();
		if (!f.canRead()) {
			throw new CheckedSimulationException("Input File cannot be
read: "
					+ f.getAbsolutePath());
		}
		try {
			Scanner sc = new Scanner(f);
			while (sc.hasNextLine()) {
				// trim whitespaces
				String line = sc.nextLine().trim();
				// ignore comment and empty lines
				if (!line.startsWith(";") && !line.isEmpty()) {
					lines.add(line);
				}
			}
			sc.close();
		} catch (FileNotFoundException e) {
			throw new CheckedSimulationException("Input File cannot be
found: "
					+ f.getAbsolutePath());
		}

		return lines;
	}
```

```
	private static Area initArea(String measures)
			throws CheckedSimulationException {
		if (!(measures.matches(regexTwoNumbers))) {
			throw new CheckedSimulationException(
					"Invalid format for area measures: '"
							+ measures
							+ "' format must be 'number number'
and number in [1..10]");
		}
		String[] numbers = measures.split(regexWhiteSpaces);
		int y = Integer.parseInt(numbers[0]);
		int x = Integer.parseInt(numbers[1]);
		Area area = new Area(y, x);
		return area;
	}

	private static void setObstacle(Area area, String obstacle)
			throws CheckedSimulationException {
		if (!(obstacle.matches(regexFourNumbers))) {
			throw new CheckedSimulationException(
					"Invalid format for obstacle coordinates: '"
							+ obstacle
							+ "' format must be 'number number
number number' and number in [1..10]");
		}
		String[] snumbers = obstacle.split(regexWhiteSpaces);
		int[] numbers = new int[4];
		for (int i = 0; i < 4; i++) {
			numbers[i] = Integer.parseInt(snumbers[i]);
		}
		int yMin = numbers[0] < numbers[2] ? numbers[0] : numbers[2];
		int yMax = numbers[0] < numbers[2] ? numbers[2] : numbers[0];
		int xMin = numbers[1] < numbers[3] ? numbers[1] : numbers[3];
		int xMax = numbers[1] < numbers[3] ? numbers[3] : numbers[1];
		area.setObstacle(yMin, xMin, yMax, xMax);
	}

}
```

OutputFileWriter

```
package io;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import model.Area;
import model.Cell;
```

```
import control.Controller;
import control.INextCellAlgorithm;
import error.CheckedSimulationException;

/**
 * class to write solutions into output files.
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class OutputFileWriter {

	private static final Log log = Log.getInstance();

	/**
	 * writes the passed solutions into the specified output file. If the
output
	 * file does not exist, it will be created. Existing output files will be
	 * overwritten.
	 *
	 * @param outFileName
	 *            name of the output file
	 * @param area
	 *            for which the solutions apply
	 * @param solutions
	 * @param algorithms
	 *            for which the solutions should be exported
	 * @throws CheckedSimulationException
	 *             if the output file cannot be written at the specified path
	 */
	public static void writeOutSolutions(String outFileName, Area area,
			Map<INextCellAlgorithm, List<Cell>> solutions,
			List<INextCellAlgorithm> algorithms)
			throws CheckedSimulationException {
		log.debug("write found solutions to output file");
		Cell start = area.getStart();
		File out = createOutFile(outFileName);
		try {
			PrintWriter pw = new PrintWriter(out);
			pw.println("Startposition " + start.toString());
			List<Cell> startOnly = new ArrayList<Cell>();
			startOnly.add(start);
			pw.println(area.toString(startOnly));
			for (INextCellAlgorithm algorithm : algorithms) {
				if (solutions.get(algorithm) != null
						&& !solutions.get(algorithm).isEmpty()) {
					pw.println(algorithm.toString() + ":");

	pw.println(area.toString(solutions.get(algorithm)));
					pw.println("Routenplan:");
					pw.println(Cell.getAsString(Controller

	.convertToRoutePlan(solutions.get(algorithm))));
				}
			}
```

```
                    pw.println("zu versiegelnde Parzellen:   "
                                + (area.getCellCount() -
area.getObstacleCount()));
                    pw.println("Hindernisparzellen:          "
                                + area.getObstacleCount());
                    // number of visited cells must be equal in all solutions
                    // so just take any solution
                    Iterator<List<Cell>> iterator = solutions.values().iterator();
                    if (iterator.hasNext()) {
                          int processedCellsCount =
solutions.values().iterator().next()
                                      .size();
                          pw.println("versiegelte Parzellen:       "
                                      + processedCellsCount);
                          pw.println("nicht versiegelte Parzellen: "
                                      + (area.getCellCount() -
area.getObstacleCount() - processedCellsCount));
                    }
                    pw.close();
              } catch (FileNotFoundException e) {
                    log.error(e);
                    throw new CheckedSimulationException(
                                "creation of output file failed");
              }

        }

        /**
         * writes the errorToFile message into the specified output file. If the
         * output file does not exist, it will be created. Existing output files
         * will be overwritten.
         *
         * @param errorToFile
         *            message to be written into the file
         * @param outFileName
         *            path to the output file
         * @throws CheckedSimulationException
         *             if the writing into the output file fails
         */
        public static void writeOutError(String errorToFile, String outFileName)
                    throws CheckedSimulationException {
              log.error("abort processing of input file");
              File out = createOutFile(outFileName);
              try {
                    PrintWriter pw = new PrintWriter(out);
                    pw.println(errorToFile);
                    pw.close();
              } catch (FileNotFoundException e) {
                    log.error(e);
                    throw new CheckedSimulationException(
                                "creation of output file failed");
              }
        }

        private static File createOutFile(String outFileName)
```

```
                throws CheckedSimulationException {
            File out = new File(outFileName);
            try {
                out.createNewFile();
                if (out.canWrite()) {
                    return out;
                } else {
                    log.error("cannot write into output file "
                                + out.getAbsolutePath()
                                + "\noutput file will not be created");
                }
            } catch (IOException e1) {
                log.error("cannot create output file " + out.getAbsolutePath()
                            + "\noutput file will not be created", e1);
            }
            throw new CheckedSimulationException("outfile creation failed");
        }

}
```

Log

```
package io;

/**
 * Singleton for all console output
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class Log {

        private static Log instance = null;
        private boolean enabled;

        /** Prevent instantiation from outside this class */
        private Log() {
            enabled = Config.getBooleanValue("debug", false);
            info("verbose logging is enabled: " + enabled);
        }

        /**
         *
         * @return the application wide unique <code>Log</code> object
         */
        public static Log getInstance() {
            synchronized (Log.class) {
                if (instance == null) {
                    instance = new Log();
                }
            }
            return instance;
        }
```

```java
/**
 * if logging is enabled msg will be printed to System.out
 *
 * @param msg
 */
public void debug(String msg) {
	if (enabled)
		System.out.println(msg);
}

/**
 * msg will be printed to System.out
 *
 * @param msg
 */
public void info(String msg) {
	System.out.println(msg);
}

/**
 * msg will be printed to System.out
 *
 * @param msg
 */
public void error(String msg) {
	System.out.println(msg);
}

/**
 * msg and e.getMessage() will be printed to System.out. If logging is
 * enabled the Stacktrace of e will be printed to System.err
 *
 * @param msg
 * @param e
 */
public void error(String msg, Exception e) {
	System.out.println(msg);
	if (enabled)
		e.printStackTrace();
	else
		System.out.println("Exception message: " + e.getMessage());
}

/**
 * e.getMessage() will be printed to System.out. If logging is enabled the
 * Stacktrace of e will be printed to System.err
 *
 * @param e
 */
public void error(Exception e) {
	if (enabled)
		e.printStackTrace();
	else
		System.out.println(e.getMessage());
}
```

```
	/**
	 * @return whether the verbose logging is enabled
	 */
	public boolean isEnabled() {
		return enabled;
	}
}
```

Config

```
package io;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

/**
 * class which provides basic methods to read from a configuration file in the
 * root directory of the classpath: "./configuration.properties"
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class Config {

	private static final String CONFIG_FILE = ClassLoader
			.getSystemClassLoader().getResource(".").getPath()
			+ "configuration.properties";
	private static final String sTrue = "TRUE|True|true|1";
	private static final String sFalse = "FALSE|False|false|0";

	/**
	 * returns the value for the given key from the property file. If value
	 * cannot be obtained from the property file, the reason is logged, and the
	 * defaultvalue is applied.
	 *
	 * @param key
	 * @param defaultValue
	 * @return
	 */
	public static boolean getBooleanValue(String key, boolean defaultValue) {
		// read configuration
		Properties prop = new Properties();
		InputStream input = null;
		boolean value = defaultValue;
		try {
			input = new FileInputStream(CONFIG_FILE);
			prop.load(input);
			String svalue = prop.getProperty(key);
			if (svalue == null) {
			} else if (svalue.matches(sTrue)) {
```

```
                        value = true;
                } else if (svalue.matches(sFalse)) {
                        value = false;
                }
        } catch (IOException ex) {
                // cannot use Log here because of deadlock possibility
                // Log calls this method in constructor
                System.err.println("config file " + CONFIG_FILE
                                + " cannot be read. Default value '" +
defaultValue
                                + "' assumed for property '" + key + "'");
        } finally {
                if (input != null) {
                        try {
                                input.close();
                        } catch (IOException e) {
                        }
                }
        }
        return value;

    }
}
```

5.2.5 Package error

CheckedSimulationException

```
package error;

/**
 * checked Exception for solver-errorcases which must be catched by calling
 * method
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class CheckedSimulationException extends Exception {

    /**
     *
     */
    private static final long serialVersionUID = 2164034546773243675L;

    /**
     * calls constructor of superclass {@link Exception}
     *
     * @param msg
     */
    public CheckedSimulationException(String msg) {
        super(msg);
    }
}
```

SimulationException

```
package error;

/**
 * unchecked Exception for solver-errorcases which may be catched by calling
 * method
 *
 * @author Fabian Braun, Prüflingsnr. 101 20510, Matr Nr. 857816
 *
 */
public class SimulationException extends RuntimeException {

	/**
	 *
	 */
	private static final long serialVersionUID = -8666280491777157001L;

	/**
	 * calls constructor of superclass {@link RuntimeException}
	 *
	 * @param msg
	 */
	public SimulationException(String msg) {
		super(msg);
	}

}
```

5.3 Entwicklerdokumentation

Die Entwicklerdokumentation liegt als javadoc vor.

Sonstige Informationen für Entwickler:

- Eingesetzte Software:
  - Java Version:
    - Java™ SE Runtime Environment 1.7.0 Update 51
    - Java™ SE Development Kit 1.7.0 Update 51
  - Entwicklungsumgebung:
    Eclipse Standard/SDK Version: Kepler Service Release 2 Build id: 20140224-0627
  - Diagramme:
    - UMLet version 12.2
    - Structorizer version 3.22
- Betriebssystem des Entwicklungs- und Testsystems:
  - Windows 7, Service Pack 1
  - 64-Bit
- Hardware:
  - Arbeitsspeicher (RAM): 8 GB
  - CPU: Intel® Core™ i5-3740; 3.20 Ghz