Diese Kopfleiste bitte unbedingt ausfüllen!

Familienname, Vorname (bitte durch eine Leerspalte trennen)

FABEMOEA MARKUS

Fach	Berufsnummer	IHK-Nummer	Prüflingsnummer
30	6511	142	18592
Sp. 1-2	Sp. 3-6	Sp. 7-14	

Termin: Montag, 12. Mai 2014

IHK

Abschlussprüfung Sommer 2014

6511

4 Entwicklung eines Softwaresystems

Schriftliche Aufgabenstellungen

Mathematisch-technischer Softwareentwickler
Mathematisch-technische Softwareentwicklerin

Bearbeitungsbogen

3 Phasen, davon

- 7 Stunden schriftliche Aufgabe
- 4 Tage Realisierung des Konzepts
- 30 Minuten Fachgespräch

100 Punkte

Vorbemerkung

Dieser Aufgabensatz besteht aus einem Aufgabenbogen und einem Bearbeitungsbogen.

Die vorliegende bundeseinheitliche Prüfungsaufgabe wird in drei Phasen bearbeitet.

Phase I:
- Schriftliche Klausur unter Aufsicht des Prüfungsausschusses der IHK
- Sie soll in der Regel montags stattfinden, Dauer 7 Stunden
- Es sind keine Hilfsmittel zugelassen.
- Die Ergebnisse sind handschriftlich auf Papier zu erstellen.
- Das Original wird dem Prüfungsausschuss übergeben, eine Kopie behält der Prüfling zur weiteren Bearbeitung.

Phase II:
- Die Bearbeitung erfolgt am betrieblichen Arbeitsplatz.
- Sie findet von dienstags bis freitags statt.
- Verwendete Hilfsmittel und Quellen sind anzugeben.
- Die Ergebnisse sind auf Papier und elektronisch lesbar nach Vorgabe des Prüfungsausschusses abzugeben.
- Hinzuzufügen ist auch eine Eigenhändigkeitserklärung.

Phase III:
- Das Fachgespräch findet zeitnah als Einzelprüfung mit dem Prüfungsausschuss statt.
- Der Prüfling soll das Prüfungsprodukt, die Aufgabenanalyse und den Lösungsentwurf in maximal 10 Minuten vorstellen und begründen. Hilfsmittel wie Flip Chart, Folien o. Ä. können verwendet werden.
- Im anschließenden etwa 20-minütigen Gespräch sind die Ergebnisse zu verteidigen.

Korrekturrand

Aufgabenanalyse

12.05.14

Auf einem Rechteckigen Feld, soll ein Roboter den Boden versiegeln. Damit die bisher Versiegelung nicht sofort wieder zerstört wird, muss eine Route über alle Felder (sofern möglich) gefunden werden, die jedes Feld einmal abläuft.

Je nach Hindernissen kann ein Feld nicht vollständig abgelaufen werden. Dann soll eine Route gefunden werden, die möglichst viele Felder versiegelt.

Die abzufahrenden Felder werden als Rechtecken in einer Liste gespeichert, die anschließend abgearbeitet wird. Der Roboter darf auf dem letzten Feld stehen bleiben.

Es gibt Hindernisse, die über mehrere Benachbarte Parzellen gehen können und die der Roboter nicht überwinden kann.

Vorgegeben ist eine Uhrzeiger-Strategie, die eine Route findet. Dabei wird auf jeder Zelle entschieden, in welche Richtung fortgefahren wird. Dabei sind natürlich einige Richtungen nicht sinnvoll oder möglich.

Das Verfahren ist ein Backtracking-Algorithmus, der zum letzten Feld zurück springt, wenn auf einer Parzelle keine Richtung mehr möglich ist.

Auf der vorherigen Parzelle wird versucht eine andere Richtung einzuschlagen. W.s.w. bis eine Lösung gefunden wurde oder keine vollständige Lösung vorhanden ist.

Korrekturrand

Meine Strategie wird ebenfalls das Backtracking-Verfahren benutzen, um eindeutig entscheiden zu können, ob es überhaupt eine gültige, vollständige Lösung gibt.

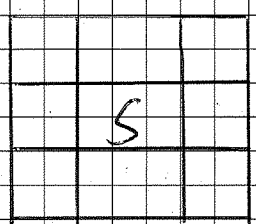
Auf jeder Parzelle wird versucht zunächst die vorherige Richtung beizubehalten. Sollte dies nicht möglich sein, wird gegen den Uhrzeigersinn eine andere Richtung probiert.

Falls z. B. die letzte Richtung „nach links" war, wird danach „nach unten", dann „nach rechts" und dann „nach oben" probiert. Beim Start wird mit „nach oben" begonnen.

Davon erhoffe ich, dass möglichst selten die Richtung geändert werden muss und so ein relativ kurzer Routenplan erstellt wird.

Beispiel:

; Abmessung der Fläche
3 3
; Startparzelle
2 2
; Eckparzellen der Hindernisse



Uhrzeiger-Strategie

↑	↑	↓
↑	↑	↓
↑	←	←

Meine Strategie

↓	←	↑
↓	↑	↑
→	→	↑

Korrekturrand

-3- Wenn mein Algorithmus ~~ob~~ nicht weiter in eine Richtung gehen kann, muss entschieden werden, ob alle Felder (außer den Hindernissen) abgearbeitet wurden. Falls ja terminiert er, sonst muss ein Schritt zurückgegangen werden.
Dies wird solange wiederholt, bis eine gültige Lösung erreicht wurde oder alle Felder in alle Richtungen abgelaufen worden sind.

Im Worst-Case hat dieser Algorithmus daher eine Laufzeit in der Klasse $O(4^k)$, wobei k die Anzahl aller ~~Felder~~ Parzellen ist.
Bei 5x5 wäre der Worstcase also 4^{25}.

Grenzfälle

Als Grenzfall gibt es zwei Szenarien:

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

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

Es wäre theoretisch möglich, dass die eine Fläche komplett voll mit Hindernissen steht, sodass nur der Startpunkt frei ist. Dies kann allerdings weder

auf den minimal Fall zurückgeführt werden.

Eingabe

Datenformat

Die Parzellen können in einem ~~Feld~~ zweidimensionalen Array abgespeichert werden. Als Typ bietet sich char an, weil hier einzelne Zeichen eingetragen werden (z.B. ^, <, v, >, S, Z, H).
Alternativ könnte man auch einen eigenen Typen (eine Klasse oder ein enum) verwenden.
Das Array kann nach dem einlesen der ersten Zeile allokiert werden, weil die Größe dann fest ist und sich nicht mehr ändert. Daher muss hier auch nicht unbedingt mit einer Array-Liste o.ä. gearbeitet werden.
Die Koordinaten hingegen sollte als List implementiert werden, weil anfangs nicht die Anzahl der Schritte bekannt ist.

Die beiden Hauptvariablen sollten zusammen als Klasse gekapselt werden, um einen einfachen und sauberen Zugriff zu ermöglichen.

Eingabe

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

Die Startpositionen und Hindernisse können dann
erst jetzt in das Datenobjekt aufgenommen werden.

Fehlerfälle

Hier wird zwischen technischen, semantischen und
syntaktischen Fehlern unterschieden.

Technisch:

- Keine Eingabe-Datei vorhanden
- keine Zugriffsrechte

Syntaktisch:

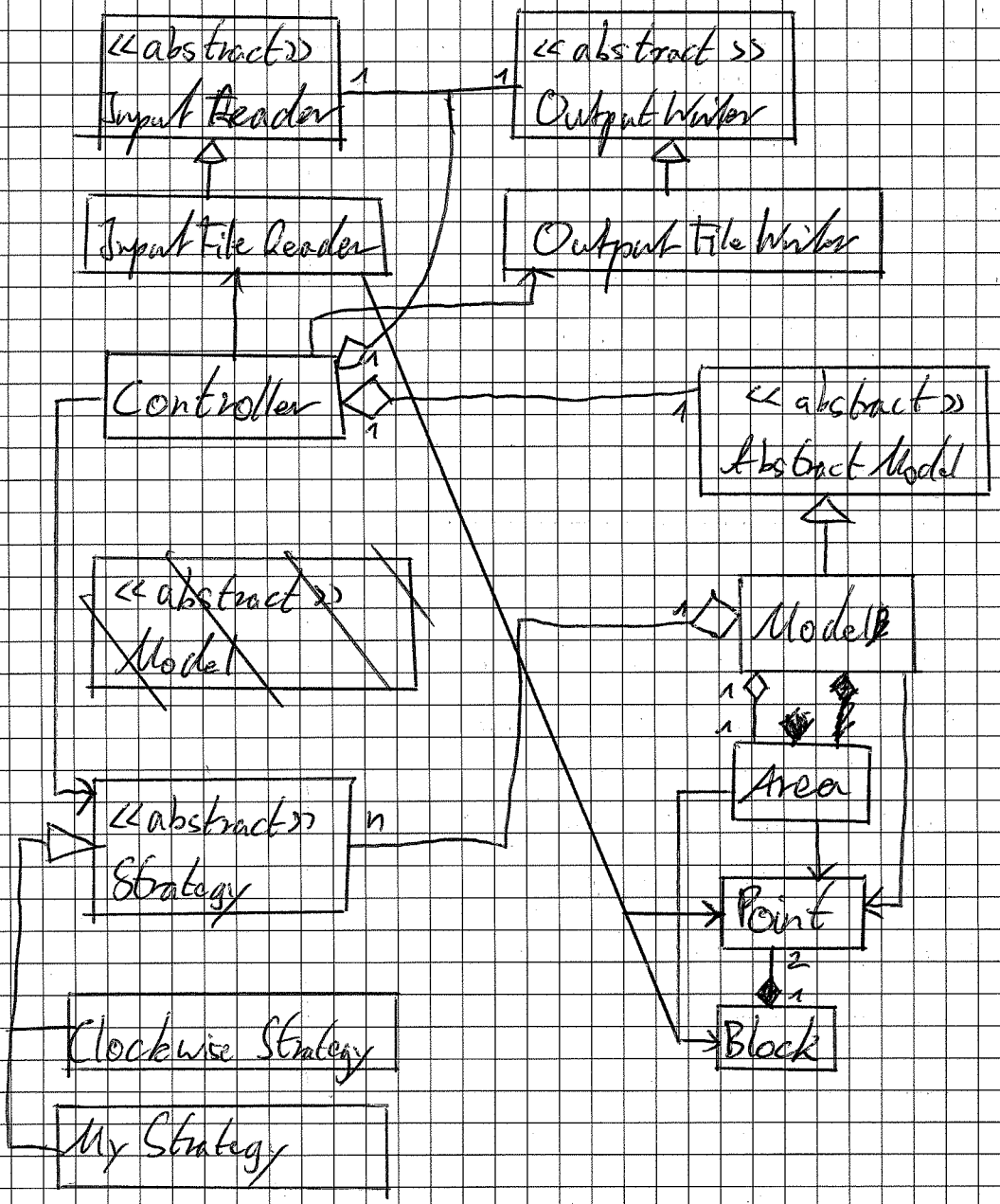
- Es fehlt die Start-~~oder~~zeile oder die Anzahl
der Felder
- Es werden keine Leerzeichen als Trenner benutzt
- Es werden Fließkommazahlen eingegeben
- ~~Es sind mehrere Hindernisse oder Startpunkte~~
~~gegeben~~
-

Semantisch

- Es sind mehrere Hindernisse oder Startpunkte
gegeben
- Der Startpunkt liegt ~~auf~~ außerhalb des
Feldes
- Die Hindernisse liegen außerhalb des Feldes
- Es sind 0 oder mehr als 10 Felder in
jeder Richtung gegeben
- mehrere Blöcke über einander.

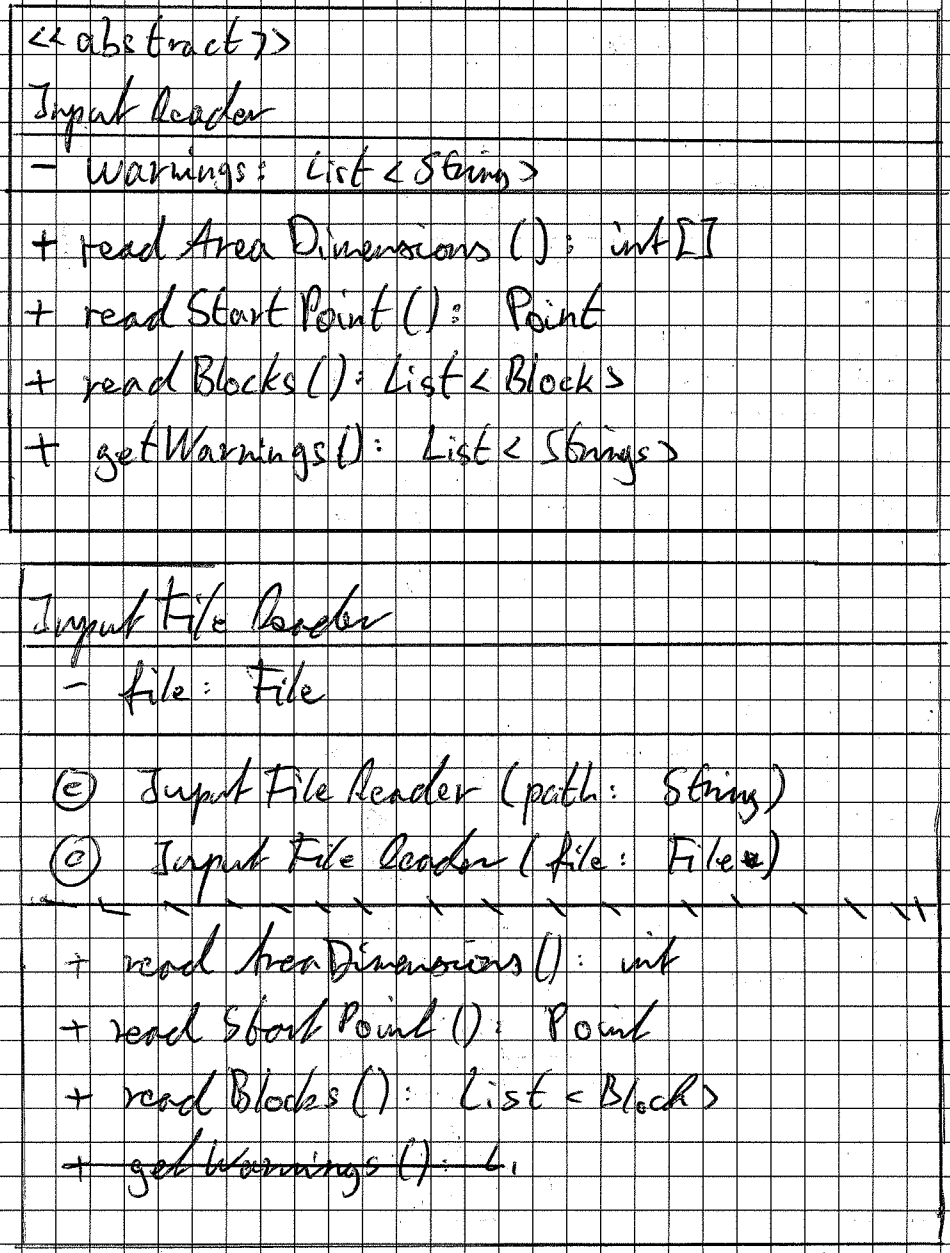
UML

Gesamtübersicht (ohne Variablen und Methoden)



Korrekturrand

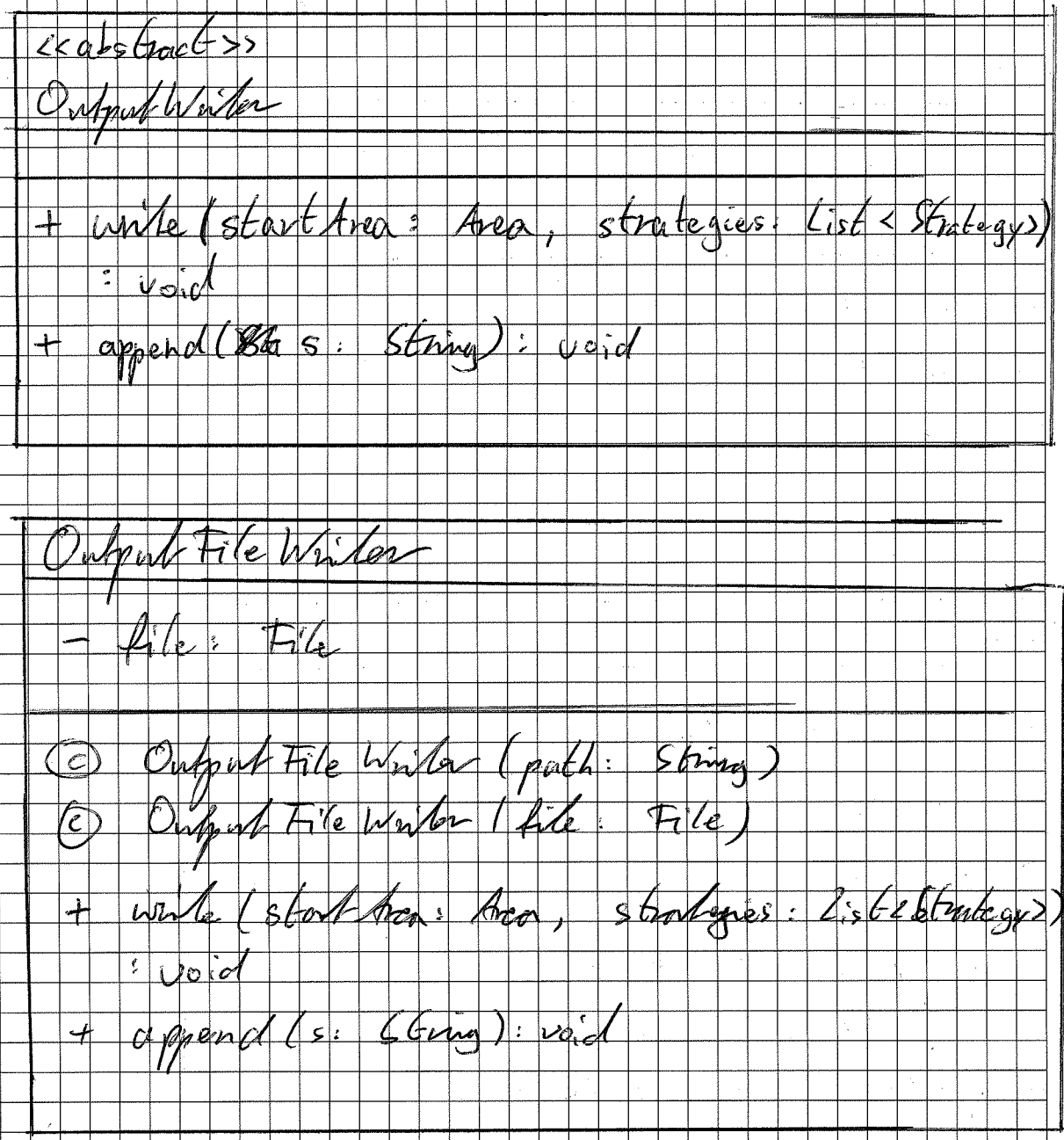
UML - View



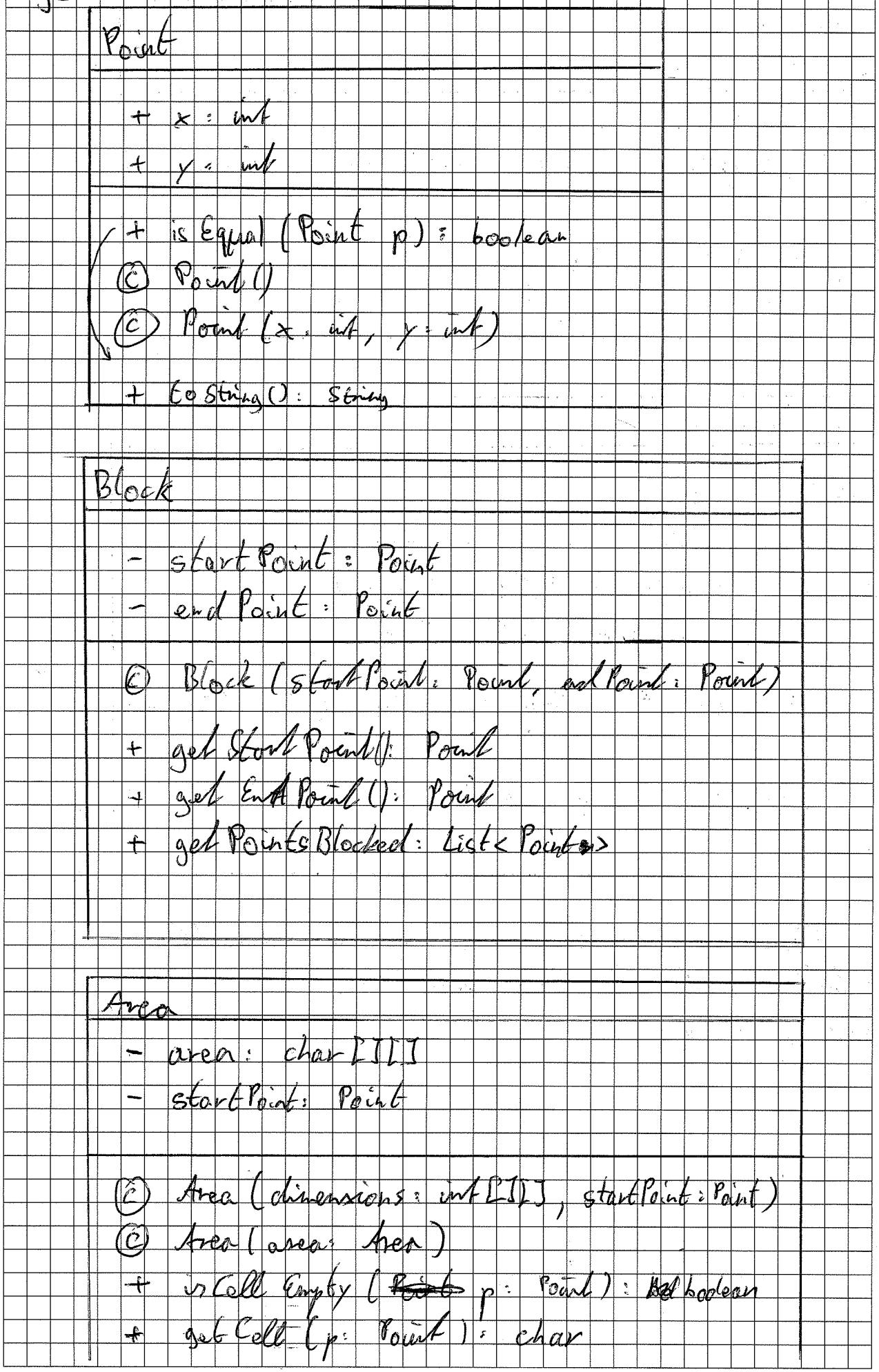
Die Klassen zum Einlesen. Neben der unbedingt benötigten Daten, kann geprüft werden, ob Fehler auftraten.

Der InputFileReader spezialisiert sich auf Dateien und ist eine konkrete Ableitung der abstrakten Klasse.

Korrekturrand



Die Klassen zur Ausgabe, die Ergebnisse darstellen. OutputFileWriter ist eine Spezialisierung zur Ausgabe in Dateien



Korrekturrand

+ setCell(p: Point, value: char): void
+ clearCell(p: Point): void
+ getStartPoint(): Point
+ numberOfWorkedCells(): int
+ numberOfNonWorkedCells(): int
+ numberOfBlockedCells(): int
+ setBlocks(blocks: List<Block>): void
+ toString(): String

Die Klasse Area stellt eine Kapselung der zu bearbeitenden (versiegelnden) Fläche dar. Sie hält die Daten zur Fläche auf der ein Algorithmus ausgeführt werden kann.

<<abstract>>
Strategy

- ~~name~~ - pointsInBestRoute: int
- area: Area
- route: List<Point>
- bestRoute: List<Point>

(c) Strategy(area: Area)
+ getName(): ~~String~~ String
+ getNextDirection(lastDirection: int, step: int): int
+ getArea(): Area
+ getRoute(): List<Point>

+ getFirstDecision(): int

~~+ getV~~

+ getValueForDecision(decision: int): char

+ addToRouteIfNeeded(lastDecision: int, pastePoint): void
+ removeFromRouteIfNeeded(point: Point): void

Korrekturrand

ClockwiseStrategy

+ getName(): String
+ getNextDecision(lastDecision: int, step: int): int

ⓒ ClockwiseStrategy(area: Area)

MyStrategy

ⓒ MyStrategy(area: Area)
+ getName(): String
+ getNextDecision(lastDecision: int, step: int): int
+ getValueForDecision(decision: int): char

Die Strategien implementieren die konkrete Entscheidung in welche Richtung gelaufen wird. Dabei hat jede Strategie ihre Area und Route, der dies in Abhängigkeit der Strategie verändert wird.

<<abstract>>
AbstractModel

+ getStartArea(): Area
+ setStartArea(area: Area): void
+ addStrategy(strategy: Strategy): void
+ getStrategies(): List<Strategy>
+ getValueForDecision(decision: int): char