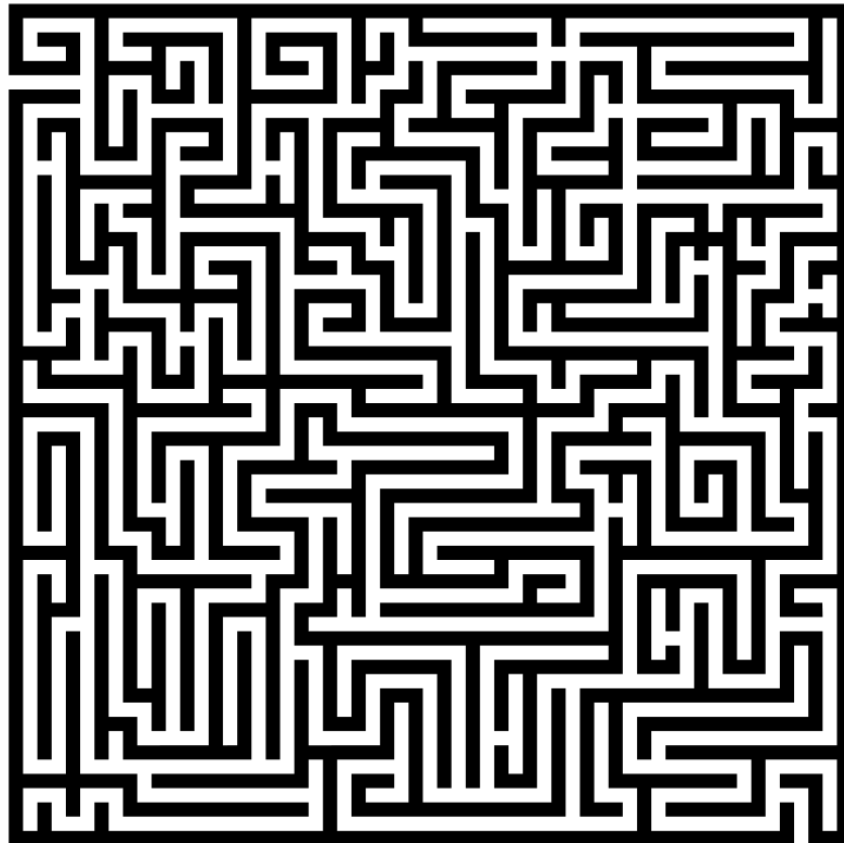


Labyrinth

Wo ist der Ausweg?



Eine Semesterarbeit an der ZHAW School of Engineering

Benjamin Bütikofer
buetikbe@students.zhaw.ch

und

Dominic Schlegel
schledom@students.zhaw.ch

Mai 2013

Inhaltsverzeichnis

<u>EINLEITUNG</u>	3
AUFGABENSTELLUNG	3
ZIELE	3
<u>LABYRINTH ALLGEMEIN</u>	4
DEFINITION	4
ARTEN VON LABYRINTHEN	4
GESCHICHTE IN EUROPA	5
<u>THEORIE ALGORITHMEN</u>	6
DEPTH FIRST SEARCH	6
PRIM ALGORITHMUS	8
RIGHT-HAND	9
A-STAR	11
<u>UMSETZUNG</u>	14
PROGRAMMIERSPRACHE	14
UMFELD	14
PATTERN	14
TESTING	15
ARCHITEKTURDIAGRAMME	16
<u>PLANUNG</u>	18
1. ITERATION (06.03.2013 – 27.03.2013)	18
2. ITERATION (03.04.2013 – 24.04.2013)	19
3. ITERATION (28.04.2013 – 25.05.2013)	20
<u>SCHLUSSWORT UND FAZIT</u>	21
ERWEITERUNGSMÖGLICHKEITEN	21
FAZIT	21
SCHLUSSWORT	21
<u>PROJEKT LINKS</u>	22
<u>LITERATURVERZEICHNIS</u>	22
<u>ABBILDUNGSVERZEICHNIS</u>	23

Einleitung

Wer als Kind schon einmal in einem Malbuch den Ausweg aus einem Labyrinth gesucht hat, weiss, dass dies nicht immer ganz einfach ist. Oft kommt man in eine Sackgasse und muss zurück zur letzten Kreuzung, um dort dann einen anderen Weg einzuschlagen. Je mehr Sackgassen ein solches Labyrinth hat, desto schwieriger wird es, aus dem Labyrinth zu finden. Nach mehr als fünf gemerkten Sackgassen, weiss man plötzlich nicht mehr, wo man schon einmal war und wo noch nicht.

Nicht erst seit der Einführung von Navigationsgeräten geht es der Menschheit darum, den schnellsten, effizientesten oder kürzesten Weg zu einem Ziel zu finden.

Bei solch einem Problem kann ein Computerprogramm natürlich Abhilfe schaffen. In diesem Software-Projekt wollen wir uns deswegen vertieft mit Fragen rund um ein Labyrinth und dessen Algorithmen befassen.

Aufgabenstellung

Die Aufgabe dieses Projektes besteht darin, eine Applikation zu erstellen, welche ein Labyrinth generiert und anschliessend die „beste“ Lösung findet. Dabei ist zu beachten, dass das generierte Labyrinth einen Eingang und einen Ausgang hat. Zudem sollte es mehrere mögliche Lösungswege geben.

Ziele

- Generieren eines Labyrinths
 - Mit dem Depth Search First Algorithmus
 - ~~Mit dem Prim Algorithmus~~
- Den Weg aus einem Labyrinth finden
 - Mit dem Right-Hand Algorithmus
 - Mit dem A* Algorithmus
- Verstehen, nachvollziehen und anwenden der Algorithmen, welche für das dynamische Generieren oder das Lösen eines Labyrinths verantwortlich sind

Labyrinth Allgemein

Wir alle wissen was ein Labyrinth ist und wie es aussehen kann. Einfache Gangsysteme können wir auch auf Anhieb und ohne viel Denkarbeit lösen. Doch welche anderen Arten gibt es denn genau und wie definiert und beschreibt man ein Labyrinth am besten?

Definition

Ein Labyrinth ist ein System, das aus Linien oder Wegen besteht und zahlreiche Kreuzungen und Sackgassen vorweisen kann, welche es schwierig machen einen Weg zu finden.

Arten von Labyrinth

Es gibt verschiedene Arten von Labyrinth. Anhand der Linienführung des Musters kann man zwei Kategorien definieren:

Verzweigungsfreies Labyrinth

Ein Labyrinth im ursprünglichen Sinne besteht aus einem verzweigungsfreien Weg. Dieser kann jedoch auf eine Art und Weise verschachtelt sein, so dass der Weg ins Ziel nicht auf Anhieb ersichtlich ist. Folgt man ihm aber, so stellt man fest, dass es keine Sackgassen gibt und man daher zwangsweise früher oder später zum Ziel oder dem Ausgang gelangt.



Abbildung 1: Kretisches Labyrinth (<http://de.wikipedia.org>)

Man denkt, dass die Urform des Labyrinths von den Kreten stammt. Der kretische König soll ein Gebäude erbaut haben lassen, welches als Gefängnis diente und ein verzweigtes Gangsystem hatte. Dies Gangsystem sollte die Insassen verwirren und eine Flucht deutlich erschweren. Diese These konnte nie bestätigt werden, da bisher niemand das Gebäude gefunden hat.

Irrgarten Labyrinth

Die kompliziertere Art eines Labyrinths nennt man auch Irrgarten. Solch ein System hat, im Gegensatz zum verzweigungsfreien Labyrinth, Sackgassen und oft deutlich mehr Kreuzungen. Bei dieser Art ist es viel schwieriger den Ausweg zu finden, da man sich beim Lösen merken muss, an welcher Verzweigung man schon einmal war, und wo die Sackgassen sind.

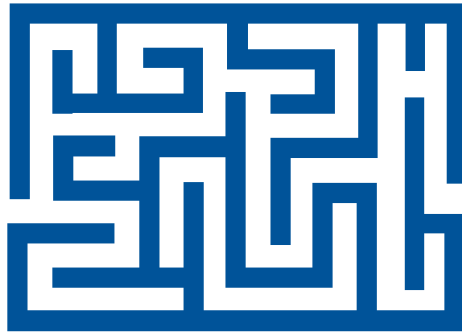


Abbildung 2: Irrgarten (<http://www.pbs.org>)

Geschichte in Europa

Das Labyrinth gehört zu den kulturellen Schätzen der Menschheit. Es ist ein Symbol, welche für die komplexen Gegensätze der Lebensordnung steht. Die Struktur, wie wir sie kennen, kommt so in der Natur nicht vor. Es ist somit eine Schöpfung des Menschen.

In Europa wurden vereinzelt im 15. Jahrhundert Labyrinth mit Verzweigungen entdeckt. Irrgärten mit Sackgassen fand man dann später im 16. Jahrhundert. Der erste bekannte Irrgarten aus Hecken, fand man um 1570 in Verona, Italien. Ab diesem Zeitpunkt nimmt die Entwicklung ihren eigenen Lauf. Bis heute werden immer wieder neue und kompliziertere Wegmuster und Labyrinth entwickelt.

Theorie Algorithmen

Depth First Search

Dieser Algorithmus wird auch Tiefensuche genannt und kann verwendet werden, um ein Labyrinth zu generieren.

Idee

Die Idee hinter dem Algorithmus ist ziemlich einfach: Man startet mit einer zufällig ausgewählten Zelle und geht so lange in eine der vier Richtungen (Nord, Süd, West oder Ost) bis es nicht mehr weiter geht. Dann geht man solange den Weg zurück, bis man wieder in eine der Richtungen gehen kann.

Algorithmus

1. Starte mit einem rechteckigen Labyrinth, welches mit Wänden gefüllt ist und eine ungerade Anzahl an Spalten und Zeilen hat.
2. Wähle zufällig eine Zelle und markiere diese als Pfad.

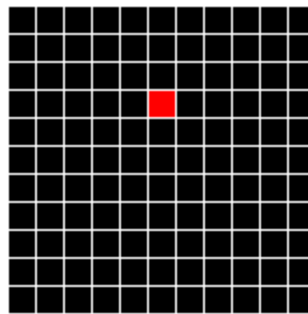


Abbildung 3: Depth First Search, Schritt 2
(<http://www.migapro.com>)

3. Wähle eine zufällige Richtung (Nord, Süd, West oder Ost) und gehe zwei Zellen in diese Richtung falls die folgenden zwei Bedingungen zutreffen:
 - a) Zwei Zellen weiter noch nicht ausserhalb des Labyrinths ist,
 - b) Die Zelle zwei Zellen weiter noch nicht als Pfad markiert wurde.

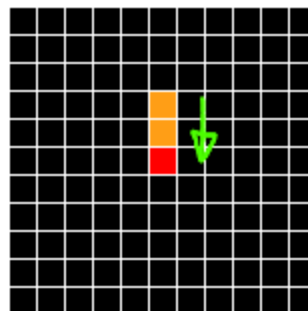


Abbildung 4: Depth First Search, Schritt 3
(<http://www.migapro.com>)

4. Wiederhole Schritt drei solange, bis die Bedingung 3b nicht mehr zutrifft, also die Zelle zwei Zellen weiter bereits ein Pfad ist.

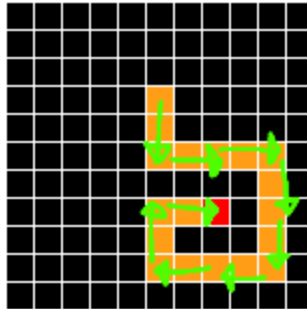


Abbildung 5: Depth First Search, Schritt 4
(<http://www.migapro.com>)

5. Gehe den Weg solange zurück, bis man wieder in eine Richtung gehen kann. Führe dann wieder Schritt 3 aus.

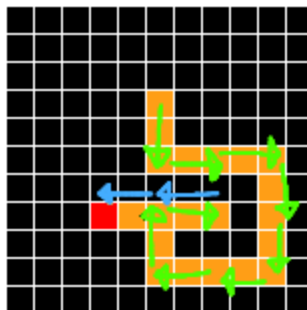


Abbildung 6: Depth First Search, Schritt 5
(<http://www.migapro.com>)

6. Wiederhole Schritt 3 bis 5, bis das ganze Labyrinth fertig gezeichnet ist.

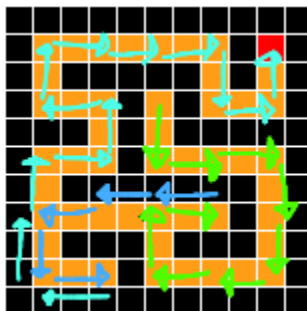


Abbildung 7: Depth First Search, Schritt 6
(<http://www.migapro.com>)

Prim Algorithmus

Obwohl wir den Prim Algorithmus schlussendlich nicht implementiert haben, befassten wir uns damit. Am Anfang wollten wir dieses Verfahren als zweiten Algorithmus zum Erstellen eines Labyrinths verwenden. Bald merkten wir jedoch, dass die Idee dahinter komplett verschieden ist. Im Vergleich zum Depth First Search basiert dieser nicht auf den Zellen, sondern auf den Wänden einer Zelle. Diese Tatsache machte es schwierig einen gemeinsamen Nenner zu finden, damit das GUI auf einfache Art und Weise gezeichnet werden konnte. Aus diesem Grund wurde der Algorithmus im Programm weggelassen.

Idee

Dieser Algorithmus basiert auf einem „Minimal Spanning Tree“. Ausgehend von einer zufälligen Zelle werden die Nachbarzellen ausgewählt, die noch nicht Teil des Labyrinthes sind. Dann wird eine zufällige Zelle dieser bereits definierten Nachbarzellen gewählt und die Wand zwischen der ursprünglichen Zelle und der gewählten Nachbarzelle wird „niedergerissen“.

Algorithmus

Der Algorithmus besteht im Wesentlichen aus drei Schritten:

1. Wähle eine zufällige Zelle und füge sie einem Labyrinth Set hinzu.
2. Wähle alle Nachbarzellen aus, welche noch nicht im Labyrinth Set sind und füge sie einem Nachbar Set hinzu.
3. Nehme eine zufällige Zelle aus dem Nachbar Set heraus, entferne die Wand zwischen der Nachbarzelle und der Zelle aus Schritt 2. und Füge diese Zelle dem Labyrinth Set hinzu
4. Wiederhole die Schritte von 2. und 3. so lange bis das Nachbar Set leer ist.

Am folgenden grafischen Beispiel kann man die Schritte gut nachvollziehen:

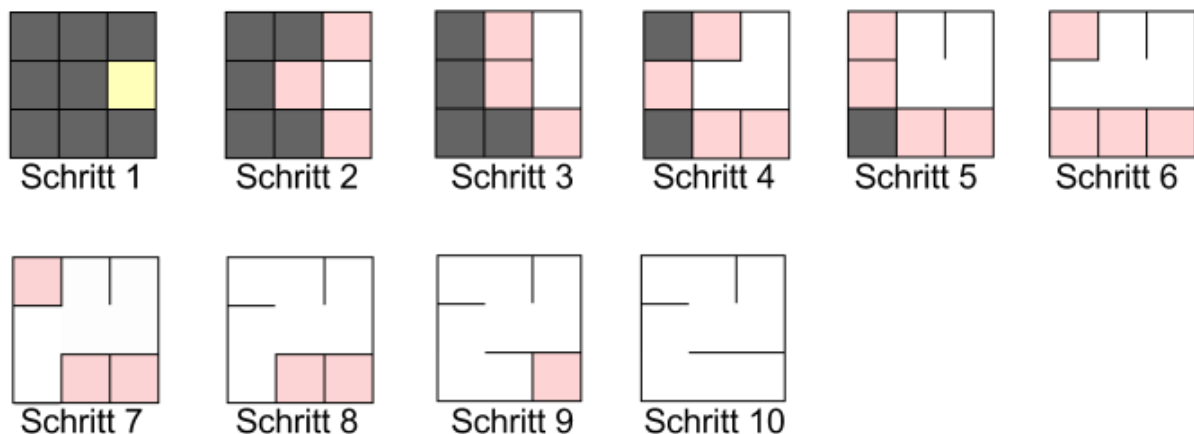


Abbildung 8: Grafisches Beispiel Prim Algorithmus

Right-Hand

Einführung

Der Right-Hand Algorithmus, auch bekannt unter dem Namen Wall-Follower, ist ein sehr einfacher Algorithmus. Um aus einem Labyrinth herauszufinden, wird bei jeder Kreuzung nach rechts abgebogen oder in anderen Worten ausgedrückt, es wird immer einer Wand gefolgt. Mit dieser Technik findet man immer aus einem Labyrinth raus.¹

Algorithmus

Folge immer der Wand zu deiner Rechten. Da ein Computer nicht weiss, wo rechts ist, mussten wir den Algorithmus etwas umbauen. Als erstes stellen wir fest, in welche Richtung wir ins Labyrinth hereingetreten sind. Als Vereinfachung legen wir fest, dass der Eingang bei uns immer entweder im Norden oder Westen ist. Wenn wir im Norden ins Labyrinth gekommen sind, so ist die Blickrichtung demzufolge nach Süden, und somit müssen wir bei der ersten Kreuzung in den Westen gehen. Wenn wir die erste Kreuzung passiert haben, wechselt unsere Blickrichtung in den Westen, rechts liegt nun also im Norden von uns.

In jeder Zelle machen wir also folgende Checks:

1. Ist das aktuelle Feld der Ausgang?
2. Wenn nicht: Können wir nach rechts abbiegen?
 - a. Ja: biege nach rechts ab
 - b. Nein: können wir geradeaus?
 - i. Ja: gehe geradeaus
 - ii. Nein: können wir nach links?
 1. Ja: Gehe nach links
 2. Nein: gehe ein Feld zurück

Auf diese Weise manövriert der implementierte Algorithmus durch das Labyrinth und findet auch immer aus den Irrgärten raus.

Implementation

Der Algorithmus basiert, wie vorher erwähnt, auf den Himmelsrichtungen. Jede Himmelsrichtung implementiert das Interface Heading und erweitert die abstrakte Klasse AbstractHeading.

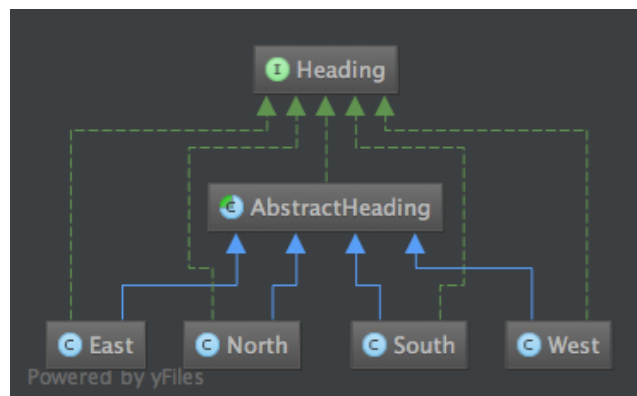


Abbildung 9: UML für Heading

¹ <http://www.nytimes.com/1989/09/06/opinion/l-why-right-hand-rule-for-mazes-works-075389.html>

Der eigentliche Lösungsalgorithmus wird also auf jeder Zelle angewendet. Der direkte Lösungsweg wird im GUI in Grün dargestellt. Blaue Wege sind Wege, welche mehr als einmal durchschritten wurden.

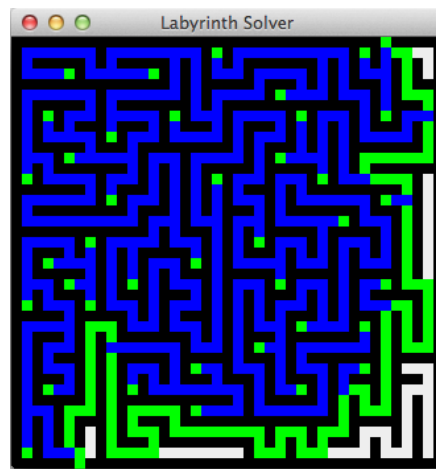


Abbildung 10: Wall-Follower in einem Labyrinth angewendet

Laufzeitkomplexität

Im schlechtesten Fall muss jede begehbare Zelle zweimal abgegangen werden.

$$\begin{aligned} \text{Anzahl Schritte} &= 2 * n^2 \\ &\Rightarrow O(n^2) \end{aligned}$$

A-Star

Einleitung

A* (A-Star, A-Stern) ist ein informierter² Pfad-Findungs-Algorithmus, welcher 1968 von Peter Hardt, Nils Nilsson und Bertram Raphael erstmals beschrieben wurde³. Der Algorithmus findet immer dann den optimalen Weg, wenn dieser in einem Graphen dargestellt werden kann. Was optimal heisst, kann definiert werden (kürzesten, schnellsten, einfachsten, etc.).

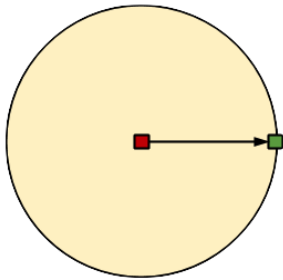


Abbildung 11: Dijkstra,
gelb = besuchte Felder,
Pfeil = gesuchter Weg

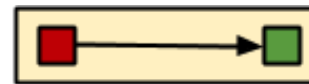


Abbildung 12: A*,
gelb = besuchte Felder,
Pfeil = gesuchter Weg

Grundsätzlich erweitert A-Stern den Dijkstra Algorithmus um eine Heuristikfunktion, welche den verbleibenden Weg vom aktuellen Punkt schätzt. Diese Erweiterung macht den A-Stern Algorithmus sehr schnell, da er nur einen Teil des Feldes absuchen muss.

Eigenschaften

A* kann Zellen unterschiedliche Kosten zuweisen, um nicht nur den kürzesten, sondern auch den günstigsten Weg zu suchen. Die Qualität des gefundenen Weges hängt stark von der Qualität der Heuristik-Funktion ab. Deswegen wird er gerne in Navigationsgeräten und Spielen verwendet.

In unserer Implementation haben wir eine sehr einfache Heuristik-Funktion implementiert (Manhattan Funktion), deshalb ist unser gefundener Weg auch nicht immer der kürzeste Weg.

Algorithmus

Es wird jeder Zelle C die Kosten $f(C)$ zugeordnet.



































$$f(C) = g(C) + h(C)$$

$g(C)$: Kosten von der Startzelle bis C

$h(C)$: Geschätzte Kosten von C bis zur Zielzelle (Heuristikfunktion)

² Informierte Suchalgorithmen verwenden, im Gegensatz zu den uninformierten, eine Heuristikfunktion um schneller ans Ziel zu kommen.

³ http://de.wikipedia.org/wiki/A*-Algorithmus

- | | | | | | | | |
|---|--|---|--|--|--|--|---|
| 100

20 80 | 94

24 70 | 80

20 60 | 74

24 50 | | | | |
| 94

24 70 | 74

14 60 | 60

10 50 | 54

14 40 | | | | |
| 80

20 60 | 60

10 50 | | 40

10 30 | | 82

72 10 | 68

68 00 | 82

72 10 |
| 94

24 70 | 74

14 60 | 60

10 50 | 54

14 40 | | 74

54 20 | 68

58 10 | 88

68 20 |
| 100

20 80 | 94

24 70 | 80

20 60 | 74

24 50 | 74

34 40 | 74

44 30 | 74

54 20 | 102

72 30 |
| | | 100

30 70 | 94

34 60 | 88

38 50 | 88

48 40 | 88

58 30 | |

Seite 12 von 23

Vor- und Nachteile

Wie schon vorher erwähnt, ist der A* Algorithmus sehr schnell, da er nur ein begrenztes Areal nach dem Weg absuchen muss. Im Gegensatz zum Dijkstra Algorithmus, muss A* aber immer die Koordinaten des Ausgangs/Ziels kennen. Man kann den A* Algorithmus also nicht dazu einsetzen, um aus verschiedenen Zielen das nächste zu finden.

Implementation

`solve()` implementiert den eigentlich Algorithmus mit einer `while`-Schleife.

`checkValue()` wird aus der `while`-Schleife aufgerufen und überprüft ob eine Zelle schon im `openSet` ist. Falls nicht, ruft es die Methode `addAndCalculate()` auf, welche die Zelle zum `openSet` hinzufügt und die f , und h Werte berechnet. Falls die Zelle schon im `openSet` ist, wird mit der Methode `checkGValueAndUpdate()` der g -Wert überprüft und falls nötig neu berechnet und gespeichert.

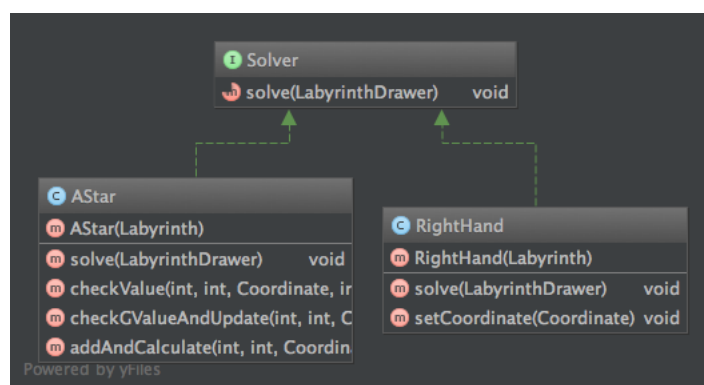


Abbildung 14: UML Diagramm des Solver-Package

Umsetzung

Programmiersprache

Im 3. Semester haben wir zusammen ein kleineres Softwareprojekt im Rahmen des Moduls “Methoden der Programmieren” gehabt. Damals verwendeten wir NodeJs. Im Nachhinein betrachtet, war dies jedoch eine schlechte Entscheidung, da wir für den geplanten Umfang noch zu wenige Wissen mit NodeJS hatten. Ebenfalls war das Testen und Ausbreiten der Applikation nicht so einfach wie erhofften. Aufgrund dieser Erkenntnisse haben wir uns für Java entschieden.

Umfeld

Damit testen, ausbreiten und analysieren des Projektes automatisiert von statten geht, haben wir uns eine automatisierte Buildumgebung aufgebaut. Als Source-Verwaltungs-System verwendeten wir Git und als zentrales Repository Github. In Verbindung mit dem Jenkins Server und dem Sonar Analyse System, wissen wir zu jedem Zeitpunkt genau über den Zustand der Applikation bescheid.

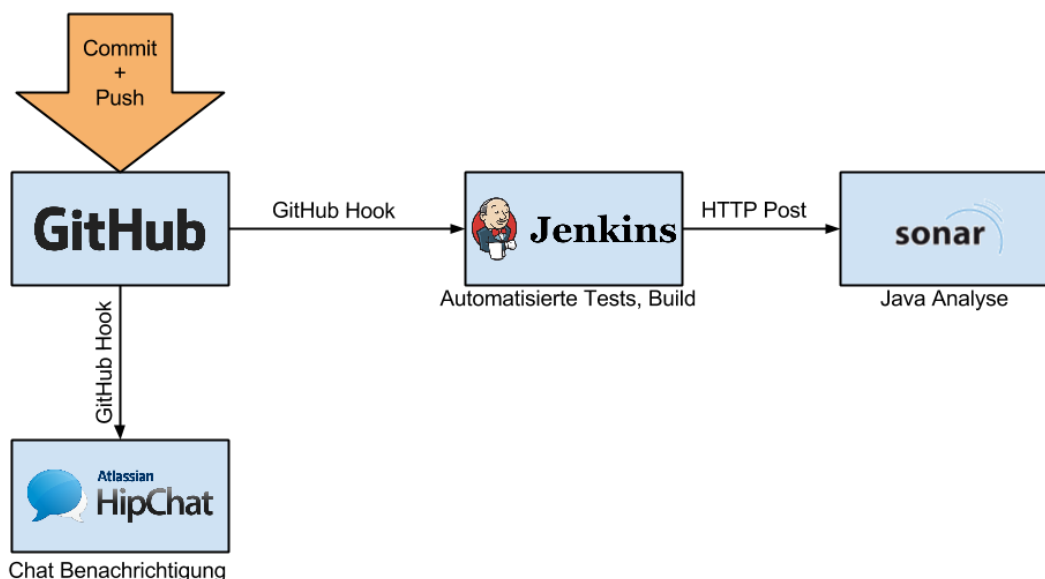


Abbildung 15: Entwicklungsumfeld

Pattern

Model-View-Controller

Um eine saubere Trennung zwischen den GUI-Elementen und dem Labyrinth Objekt zu haben, entschieden wir uns für das MVC-Pattern. Der Controller orchestriert die Kommunikation zwischen View und Model. Wie man in Abbildung 17 sehen kann, gibt es keine Abhängigkeiten zwischen Model und View.

Observer

Damit es möglich ist, in Echtzeit das GUI beim Erstellen und Lösen des Labyrinths zu aktualisieren, setzen wir das Observer-Pattern ein. Die MazePanelView ist dabei der Observer und der Controller informiert dabei die View bei jeder Aktualisierung.

Damit das GUI die schon zurückgelegten Schritte nicht vergisst, speichern wir nach jedem Aufruf das aktuelle Bild und verwenden dies beim nächsten Aufruf weiter. Das bremst zwar den Right-Hand und Depth-First-Search Algorithmus etwas aus, könnte aber noch verbessert werden.

Testing

Um einzelne Funktionen und Methoden testen zu können, haben wir uns für Junit entschieden. Dieses Framework ermöglicht es auf einfache Art und Weise Tests durchzuführen.

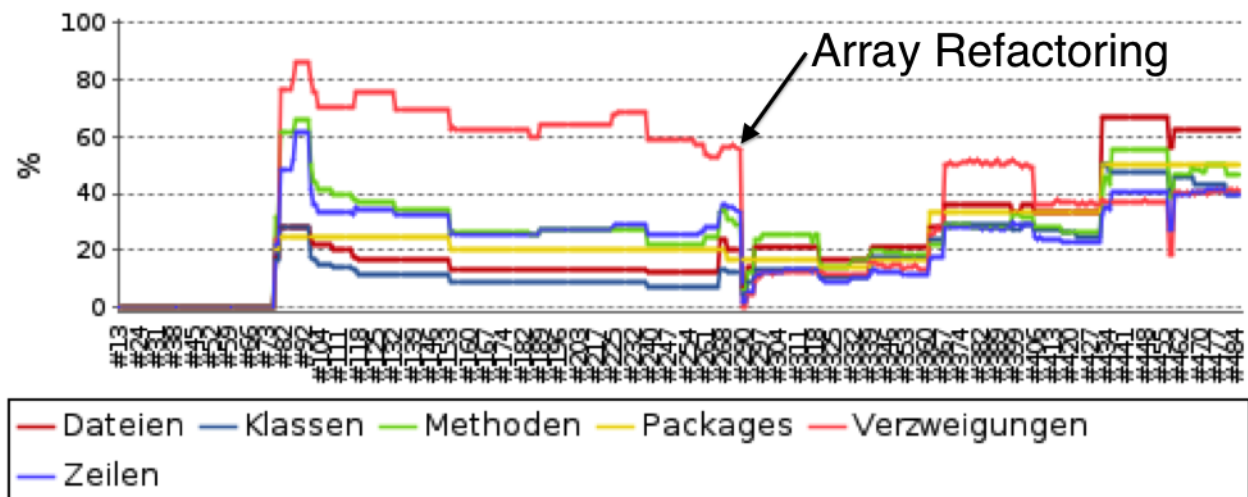


Abbildung 16: Test Coverage Report über die Projektzeit

Architekturdiagramme

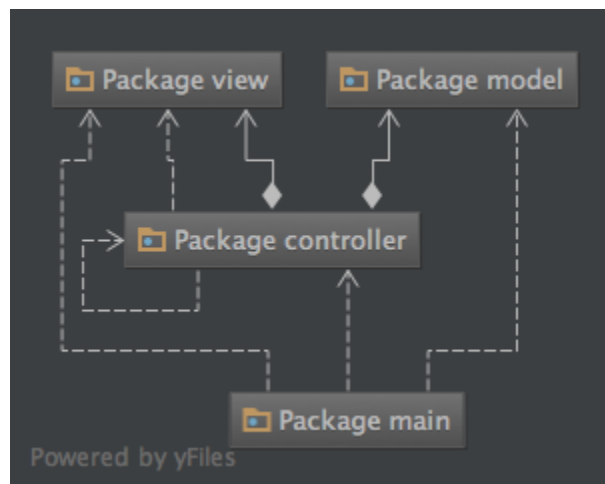


Abbildung 17: Abhängigkeiten zwischen den Packages

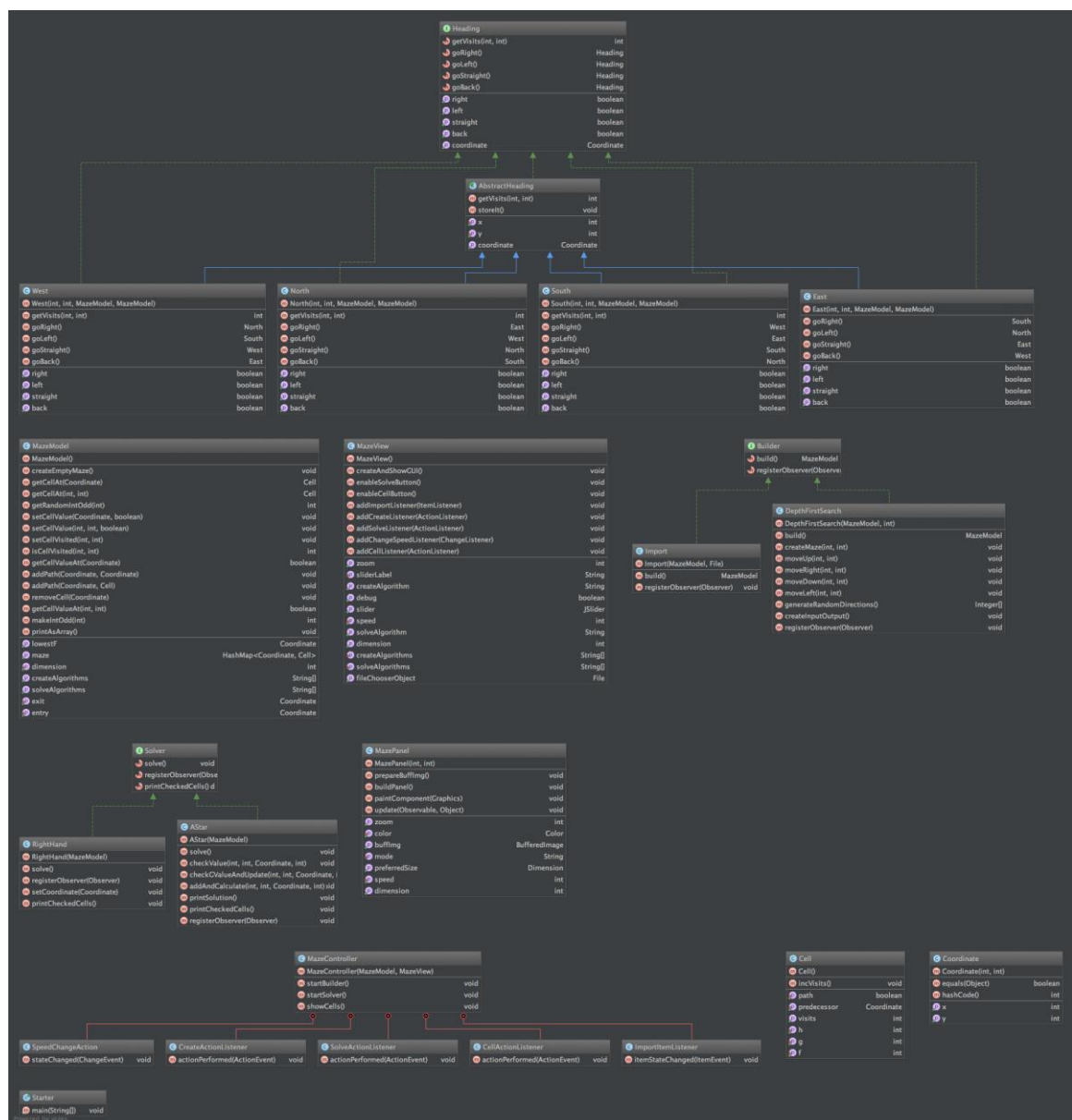


Abbildung 18: UML der gesamten Applikation (<http://sdrv.ms/13dwaZ0>)

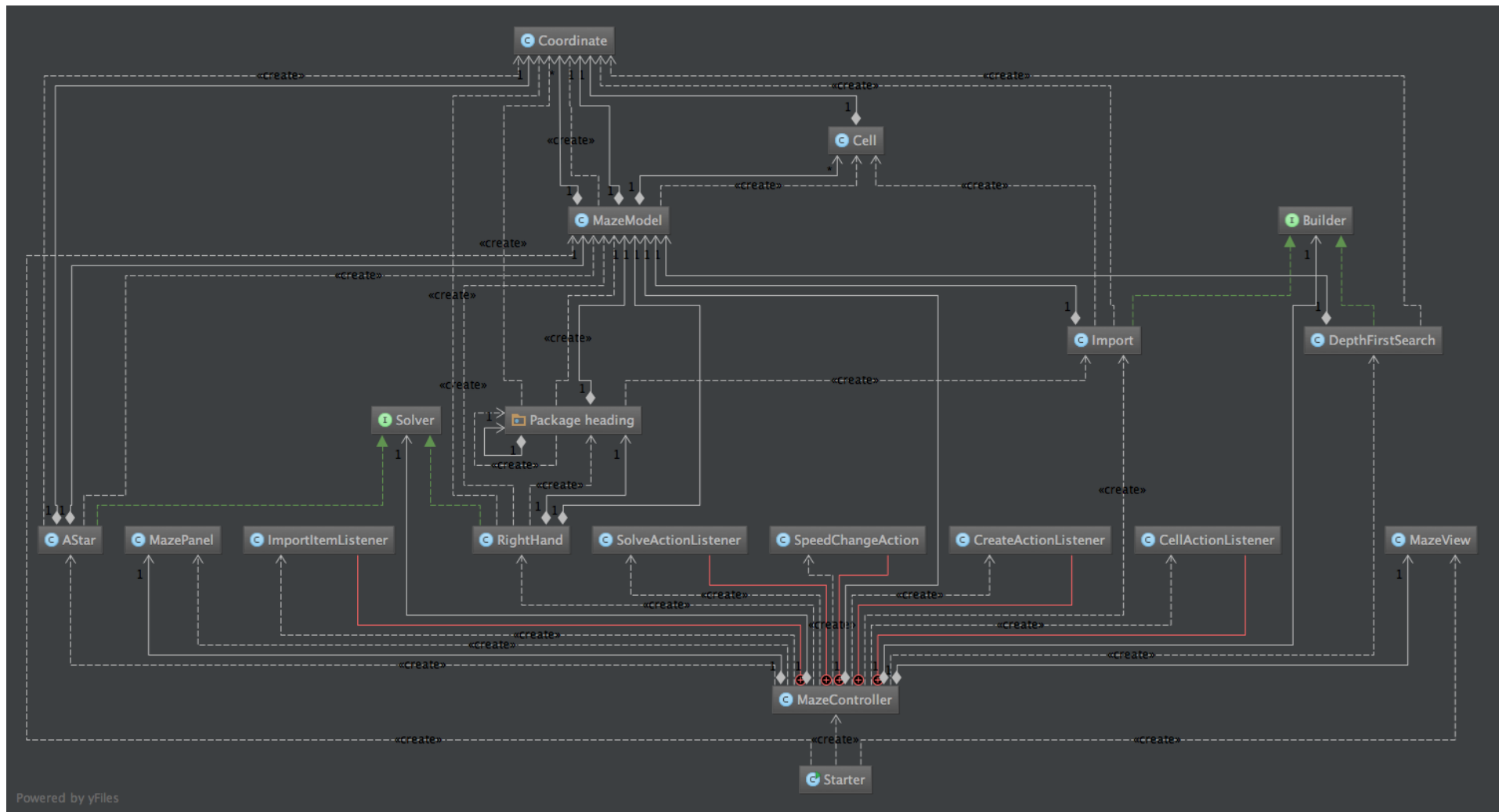


Abbildung 19: Abhängigkeiten der einzelnen Klassen (<http://sdrv.ms/18u0e30>)

Planung

Wir haben unser Projekt in drei Iterationen und eine Pufferphase unterteilt. Wir wollten uns als erstes um die Algorithmen kümmern, welche ein Labyrinth generieren und dazu das GUI erstellen. In der zweiten Phase des Projekts widmeten wir uns der Realisierung eines Algorithmus zum Lösen eines Labyrinths und implementierten weitere Erstellungsalgorithmen. Im dritten Teil haben wir weitere Lösungsalgorithmen implementiert und das GUI vervollständigt. Natürlich hielten wir uns einen gewissen Puffer für das Projektende auf. Dieser wurde für Code-Reviews, Bug fixing, Dokumentation und andere Verbesserungen verwendet.

Die Iterationsplanung erfolgt mittels der Webapplikation Scrumdo welche unter www.scrumdo.com erreichbar ist.

1. Iteration (06.03.2013 – 27.03.2013)

Zu Beginn unseres Projektes waren wir voller Elan und starteten nach der Planung direkt mit der Implementation der Tasks. Als Datenstruktur für ein Labyrinth wurde ein zweidimensionales Integer-Array gewählt. Das GUI wurde zu diesem Zeitpunkt mit den Basiselementen aus Java gebaut.

GUI Skelet (5 Stunden geschätzt)

- Ein erstes Layout für das Anzeigen der Labyrinth existiert (Ben)
- Der Benutzer soll im GUI auswählen können, welche Algorithmen verwendet werden (Ben)
- Die Aktionen auf den Buttons sollen implementiert werden, so dass man mittels dem GUI alles steuern kann (Ben)

Erster Labyrinth Erstellungsalgorithmus implementiere (8 Stunden geschätzt)

- Der Depth First Search Algorithmus wird implementiert (Dominic)
- Eine Basis Struktur für das erstellen von Labyrinths wird erstellt (Dominic)

Zeichnen des Labyrinths im GUI Step-by-Step (8 Stunden geschätzt)

- Labyrinth Punkt für Punkt zeichnen (Ben)

2. Iteration (03.04.2013 – 24.04.2013)

In der zweiten Iteration konnte nicht so viel an dem Projekt gearbeitet werden wie wir geplant hatten. Aufgrund von beruflichen Veränderungen bei Ben und viel Wochenendarbeit bei Dominic konnte die Iteration nicht planmässig beendet werden. Die verbleibenden Tasks wurden aber dem geänderten Zeitplan angepasst.

Während der Implementation des Prim Algorithmus und des Right-Hand Solvers haben wir bemerkt, dass die Grundstruktur unserer Applikation mit dem 2 dimensional Array nicht genügt. Aus diesem Grund haben wir uns für ein Refactoring der Applikation entschieden. Wir setzten dabei nun auf einen objektorientierten Ansatz mittels einer HashMap mit dem Objekt „Coordinate“ als Key und „Cell“ als Value.

Dieses Refactoring hat viel Zeit in Anspruch genommen, so dass wir mit den eigentlichen Implementationen der Algorithmen etwas zurück lagen. Dank guter Planung hatten wir aber einen Projektpuffer, der uns jetzt zugutekam.

An dieser Stelle des Projekts haben wir entschieden, dass wir den Prim Algorithmus für das Erstellen eines Labyrinths nicht implementieren, da für uns die grosse Schwierigkeit besteht, die verschiedenen Algorithmen auf einen gemeinsamen Nenner zu bringen. Dies wäre jedoch notwendig um eine einfache GUI Implementierung zu realisieren. Daher werden wir ein komplexeres vordefiniertes Labyrinth erstellen und dieses mit dem Right-Hand und einem weiteren intelligenteren Algorithmus lösen lassen.

Erster Lösungsalgorithmus implementieren (13 Stunden geschätzt)

- Right-Hand Algorithmus implementieren (Ben)

Lösung im GUI anzeigen (5 Stunden geschätzt)

- Mit einer anderen Farbe als schwarz die Lösung ins bestehende Labyrinth zeichnen (Ben)

GUI Optimierungen (3 Stunden)

- Debug Mode für GUI, der auf der Konsole das Labyrinth als Array ausgibt (Dominic)

Verbessern der Algorithmen (5 Stunden geschätzt)

- Ein und Ausgänge zeichnen beim Depth First Search Alge (Dominic)
- ~~Prim Algorithmus vervollständigen (Dominic)~~

3. Iteration (28.04.2013 – 25.05.2013)

In der dritten Iteration fokussierten wir uns klar auf die Fertigstellung des Projektes. Um unser Ziel zu erreichen mussten wir noch einen weiteren Lösungsalgorithmus implementieren und diverse kleinere Anpassungen am GUI vornehmen.

Wir stellten fest, dass wir für das interaktive Zeichnen im GUI unser Programm ändern müssen. Nach langer und intensiver Recherche stand fest, dass wir das Observer Pattern implementieren müssen, um diese Funktionalität gewährleisten zu können. Dies stellte sich jedoch nochmals als eine grosse Hürde heraus. Dafür lief die Implementation des zweiten Lösungsalgorithmus sehr gut und die Story konnte in Rekordzeit erfüllt werden.

Die Lösung interaktiv im GUI darstellen (13 Stunden geschätzt)

- Die Lösung Punkt für Punkt zeichnen (Ben)
- Geschwindigkeit für das Zeichnen ist einstellbar (Ben)
- Labyrinth zentriert zeichnen (Ben)

Weitere Algorithmen implementieren (13 Stunden geschätzt)

- A-Star Lösungsalgorithmus implementieren (Ben)

Dokumentation kompletieren (8 Stunden geschätzt)

- Grundgerüst erstellen (Dominic)
- Depth-First-Search detailliert beschreiben (Dominic)
- Lösungs-Algorithmus Right-Hand beschreiben (Ben)
- Lösungs Algorithmus A-Star beschreiben (Ben)

Schlusswort und Fazit

Erweiterungsmöglichkeiten

Da unsere Applikation ein Model für das Labyrinth verwendet, ist es nicht schwierig weitere Algorithmen für die Erstellung zu implementieren. Je nach gewähltem Algorithmus könnte es jedoch noch einigen Aufwand mit sich bringen, da der Algorithmus in der bereits vorhandenen Datenstruktur abgebildet werden muss. Falls dies nicht auf Anhieb möglich ist, müssten viele Teile der Applikation geändert werden. Dann wiederum müsste jedoch deutlich mehr Zeit investiert werden.

Solange die Struktur des Labyrinth-Objekts gleich bleibt, können weitere Lösungsalgorithmen mit der Verwendung des Solver-Interfaces einfach implementiert werden.

Eine weitere Erweiterungsmöglichkeit wäre, eine bessere Heuristikfunktion für den A-Stern Algorithmus zu entwickeln um bessere und schnellere Resultate zu erhalten.

Fazit

Auch ein scheinbar einfacher Algorithmus kann schwierig zu implementieren sein. Der Schritt von dem theoretischen Algorithmus zur effektiven Implementation braucht viel Denkarbeit. Aus diesem Grund sollte man in einem zukünftigen Projekt zuerst detaillierte Überlegungen anstellen, bevor man mit dem Programmieren startet. So könnte ein zeitintensives Refactoring vermieden werden.

Es lohnt sich auch, die User-Stories von Anfang an genau auszuformulieren. Der Kurs „Methoden der Programmierung“ war nicht nur graue Theorie.

In weiteren Java Projekten, welche ein graphisches Benutzerinterface benötigen, werden wir von Anfang an das Model, View, Controller Pattern (kurz MVC) verwenden. Das Designpattern hat sich in der Programmierwelt schon oft bewährt und vereinfacht die Interaktionen zwischen dem Benutzer und dem GUI.

Schlusswort

Das Labyrinth Projekt hat uns viel Spass gemacht, obwohl es viele Hürden zu überwinden gab. Insbesondere mit dem Java Swing GUI hatten wir sehr zu kämpfen. Ohne grosse Vorkenntnisse der Spezialitäten dieser Java Komponente, war es sehr schwierig, unser Ziel, das Realtime-Zeichnen der Lösungsalgorithmen, zu erreichen.

Es war toll die Applikation am Schluss in Aktion zu sehen und wir hätten tagelang unserem Programm beim Lösen der Labyrinth zusehen können.

Projekt Links

Github Project Site	https://github.com/do3meli/Labyrinth
Jenkins Build Server	http://travelbutlr.com:8080/job/Labyrinth/
Sonar Analyse	http://rob.nerdherd.ch:9000/dashboard/index/269
Online Javadoc	http://travelbutlr.com:8080/job/Labyrinth/javadoc/

Literaturverzeichnis

Wikipedia Labyrinth Allgemein http://de.wikipedia.org/wiki/Labyrinth	Stand: April 2013
Wikipedia Maze Solving Algorithm http://en.wikipedia.org/wiki/Maze_solving_algorithm	Stand: Mai 2013
Wikipedia Maze Generation Algorithm https://en.wikipedia.org/wiki/Maze_generation_algorithm	Stand: Mai 2013
Depth-First-Search, Maze Algorithm http://www.migapro.com/depth-first-search/	Stand: November 2011
Geschichte des Labyrinths, Labyrinth International http://labyrinth-international.org/cms/index.php?page=1867263843&f=1&i=12257	Stand: März 2013
Einführung in den A-Stern Algorithmus http://www.policyalmanac.org/games/aStarTutorial.htm	Stand: Mai 2013
Maze Generation by Jamis Buck http://weblog.jamisbuck.org/under-the-hood	Stand: Mai 2013

Abbildungsverzeichnis

Abbildung 1: Kretisches Labyrinth (http://de.wikipedia.org)	4
Abbildung 2: Irrgarten (http://www.pbs.org)	5
Abbildung 3: Depth First Search, Schritt 2 (http://www.migapro.com)	6
Abbildung 4: Depth First Search, Schritt 3 (http://www.migapro.com)	6
Abbildung 5: Depth First Search, Schritt 4 (http://www.migapro.com)	7
Abbildung 6: Depth First Search, Schritt 5 (http://www.migapro.com)	7
Abbildung 7: Depth First Search, Schritt 6 (http://www.migapro.com)	7
Abbildung 8: Grafisches Beispiel Prim Algorithmus	8
Abbildung 9: UML für Heading	9
Abbildung 10: Wall-Follower in einem Labyrinth angewendet	10
Abbildung 11: Dijkstra, gelb = besuchte Felder, Pfeil = gesuchter Weg	11
Abbildung 12: A*, gelb = besuchte Felder, Pfeil = gesuchter Weg	11
Abbildung 13: Das Koodrinen System nach Anwendung des A* Algorithmus (http://www.policyalmanac.org/games/aStarTutorial.htm)	12
Abbildung 14: UML Diagramm des Sovler-Package	13
Abbildung 15: Entwicklungsumfeld	14
Abbildung 19: Test Coverage Report über die Projektzeit	15
Abbildung 17: Abhängigkeiten zwischen den Packages	16
Abbildung 18: UML der gesamten Applikation (http://sdrv.ms/13dwaZ0)	16
Abbildung 18: Abhängigkeiten der einzelnen Klassen (http://sdrv.ms/18uOe30)	17