PTP 2018 [1]
Projektdokumentation
BeepBoop - Das Roboterspiel

Pawel Rasch [2], Tim Runge [3]
Lehrender: Andreas Heymann, Dipl.-Inform.

12. August 2018

[1]Lehrveranstaltung 64-146b, Universität Hamburg, Fachbereich Informatik
[2]Matrikel-Nr.: 5616844, philopaw@gmx.net
[3]Matrikel-Nr.: 6347952, 6runge@informatik.uni-hamburg.de

Inhaltsverzeichnis

Kapitel 1

Einleitung

Im Rahmen des Praktikums: „Programmiertechnisches Praktikum“ wurde die dankbare und spannende Aufgabe gestellt, ein Projektthema selbstständig zu erarbeiten und umzusetzen, wobei das Vorgehen und die Ergebnisse zu dokumentieren waren. Diesem Zweck dient vorliegende Projekdokumentation, welche in drei Kapitel unterteilt ist:

- Einleitung
  Hier werden Inhalt und Aufbau der Dokumentation dargelegt
- Das Projekt
  Dieser Teil enthält die generelle Projektbeschreibung und einen Userguide, der den Nutzer durch das Demo Level führt.
- Die Entwicklung
  Dieses Kapitel ist dem Vorgehen bei Umsetzung und Planung, sowie den dabei zu meisternden Schwierigkeiten gewidmet.

Kapitel 2

Das Projekt

In unserem Entwicklungsteam wurde sehr schnell deutlich, dass im Rahmen der Projektfindung der Anspruch an das Thema stetig gestiegen ist. Zunächst war aber eine generelle Entscheidung zu treffen: Sollte als Projektthema eine nützliche Anwendung oder ein erheiterndes Spiel gewählt werden? Die Autoren waren sich nach einigem hin und her und etlichen gemütlichen Treffen einig, dass es ein Spiel werden sollte. Die persönlichen Preferenzen diktierten dabei einen strategischen Charakter des Spiels. So entstand folgende Idee:

2.1 Projektbeschreibung

'BeepBoop - das Roboterspiel' ist von der Grundidee her vom an der Universität Hamburg genutzten Lernspiel „Karel“ inspiriert. Dort kann auf einem kleinen Feld mittels einer an Java angelehnten und stark eingeschränkten Programmiersprache ein Roboter befehligt werden, um kleine Aufgaben zu lösen. 'BeepBoop' sollte folglich ein Strategiespiel werden, in dem Roboter programmiert werden. Als Aufgabe bot sich das Sammeln von Ressourcen auf einer ausgedehnten Spielfläche an, die sinnigerweise dazu verwendet werden können weitere Roboter zu bauen. Es sollte zudem einen Spieler geben, mit dem die Welt erkundet werden kann.

Dabei war von Anfang an klar, dass am Ende der vorgesehenen Projektzeit kein vollständiges Spiel stehen würde, sondern ein funktionsfähiger und ausbaubarer Prototyp, der ein gutes Gefühl dafür vermitteln kann, wie ein voll ausgereiftes und balanciertes Spiel aussehen könnte. Wichtige Zielvereinbarung war die generelle Spielbarkeit.

Die graphische Ausarbeitung sollte dabei vor allem funktional sein und auf unnötige Extras verzichten. Anvisiert wurde eine 2D Ansicht der Spielfläche, die sich auf einen Ausschnitt des gesamten Levels beschränkt. Dieser Ausschnitt würde sich mit dem Spieler mitbewegen und so die Illusion einer weiten Welt erzeugen. Um die Komplexität gering zu halten, wurde dabei an ein Raster gedacht, so dass sich Gegenstände nur über den Wechsel zum benachbarten Feld bewegen würden. Animationen oder eine kontinuierliche Welt waren nicht vorgesehen.

Roboter sollten im Spiel gebaut und programmiert werden können, Ressourcen abbaubar sein. Die nötigen Informationen sollten über die UI abrufbar sein. Eine einfache Skizze diente dabei als Grundlage.

Ein weiteres 'must-have' war zunächst die Persistenz: Spielstände sollten speicherbar sein und aus dem Spiel heraus geladen werden können. Dies wurde dann aber relativiert, da der Umfang bereits sehr groß erschien. Weitere Features sollten hauptsächlich während der Entwicklung hinzukommen, da Ideen dazu bereits vor Beginn geradezu sprudelten.

Das Ziel war also eine standalone singleplayer Java-Applikation mit Swing als Grundlage.

2.2 Userguide

Nach dem Starten der App befindet sich der Spieler sofort im BeepBoop Demo Level. Es beinhaltet neben dem Spieler Avatar (der Strohhut in der Mitte des Kartenausschnitts) große Mengen an Ressourcen, drei Roboter und ein Terminal.
Die Steuerung der Spielfigur erfolgt mittels Cursor Tasten. Läuft der Spieler gegen eine der Ressourcen, so baut er automatisch einen Teil davon ab. Abgebaute Ressourcen werden im Inventar, das sich unterhalb des Kartenausschnitts befindet, angezeigt. Sie können benutzt werden, um weitere Roboter zu bauen, die wiederum programmiert werden können, um weitere Ressourcen abzubauen.

Um Roboter zu bauen, muss das Roboter Terminal aktiviert werden. Es befindet sich im Demo Level nördlich vom Spieler. Um es zu aktivieren, muss der Spieler lediglich dagegen laufen.
Im Terminal Menü, das sich rechts neben dem Kartenausschnitt befindet, lassen sich Roboter konstruieren, programmieren und ihre Logs auslesen. Nach-

dem der User mit der Maus durch das Terminal Menü navigiert hat, muss erst mit einem Klick auf den Kartenausschnitt der entsprechende Fokus wiederhergestellt werden, bevor die Spielfigur wieder bewegt werden kann. Diese Unannehmlichkeit würde bei einer Fortführung des Projekts ausgebessert.
Bisher gibt es nur eine Art von BeepBoop Robotern. Der Basic Robot kostet 100 Iron, 50 Silicon und 25 Copper. Ein Mausklick auf den Construct-Button im Construct-Robots-Submenü lässt einen Roboter neben dem Spieler erscheinen, wenn es neben ihm noch einen freien Platz gibt und er die nötigen Ressourcen im Inventar hat.
Außerdem kann der User ins Manage-Robots-Menü navigieren. Hier sieht man zuerst einmal das Error Log eines der Roboter. Über das Obere Drop Down Menü lässt sich der zu managende Roboter auswählen. Unterhalb dieses Menüs wird die Ladung des ausgewählten Roboters angezeigt. Über das Drop Down Menü unten links lässt sich via „Load Program“ das Programm des ausgewählten Roboters auslesen. BeepBoop Roboter Programme setzen sich aus folgenden Befehlen zusammen:

- U
  Der Roboter versucht sich ein Feld nach oben zu bewegen
- D
  Der Roboter versucht sich ein Feld nach unten zu bewegen
- L
  Der Roboter versucht sich ein Feld nach links zu bewegen
- R
  Der Roboter versucht sich ein Feld nach rechts zu bewegen
- LD r n
  mit r $\in \{U, D, L, R\}$, n $\in \mathbb{N}$. Befindet sich in Richtung R direkt neben dem Roboter eine Ressource, so baut er davon n ab, wenn er genug Kapazitäten frei hat.
- DP r n
  mit r $\in \{U, D, L, R\}$, n $\in \mathbb{N}$. Der Roboter legt n der Ressource, die er geladen hat, in Richtung r ab, wenn möglich. Befindet sich in der angegebenen Richtung ein Roboter Terminal, so landet die Ressource im Inventar des Spielers.
- IF s r
  mit s $\in \{FREE, RESOURCE\}$ r $\in \{U, D, L, R\}$. Der Roboter benutzt

seinen Sensor s in Richtung r. Meldet der Sensor ein negatives Ergebnis (zum Beispiel bei IF RESOURCE L, wenn sich links neben dem Roboter keine Ressource befindet), dann wird das Programm nach dem nächsten END weiter ausgeführt. Bei einem positiven Ergebnis geht es direkt mit der nächsten Programmzeile weiter.

- END
  Hier endet der Codeblock des letzten IFs

- GOTO n
  mit n $\in \mathbb{N}$. Sprung zur Programmzeile n.

Ist ein Programm geschrieben oder mittels des Import-Program-Buttons aus einer Datei geladen, lässt es sich per Klick auf Apply aktivieren. Von nun an führt der Roboter das neue Programm aus. Wenn er bei der letzten Codezeile angekommen ist, beginnt er es von vorn. Mit dem Export-Button lassen sich Programme und Error Logs als Dateien auf die Festplatte schreiben. Der Spielstand kann jederzeit gespeichert, oder ein alter geladen werden. Dies geschieht über das File-Menü.
Der Leser möge daran erinnert sein, dass oft ein Klick auf das Level nötig ist, um den Spieler wieder bewegen zu können.
Der BasicRobot mit der Nummer 2 (die Zählung beginnt bei 0) zeigt, wie mächtig die Programmierung bereits ist. Er durchläuft das Level von seiner Position systematisch in Richung Terminal und sammelt alles Gold auf, das ihm in den Weg kommt. Trifft er auf ein Hindernis, versucht er das Gold abzuladen. Schließlich trifft er auf das Terminal und kann die gesammelten Ressourcen dem Spieler verfügbar machen. Das Level enthält in diesem Bereich nur Gold als Ressource, da der BasicRobot zu jedem Zeitpunkt nur eine Art von Ressourcen tragen kann. Für eine einfachere Programmierung müssten weitere Sensoren (Check auf Art und Inhalt der gerade getragenen Ressource, etc.) implemetiert werden.
Ein aus Sicht des Teams erfreuliches Feature sind die Events, die nach einer gewissen Zeit stattfinden. Eine begrüßende und weitere, agitierende Nachrichten werden nach und nach angezeigt. Diese erscheinen als modales Fenster und schließen sich nach Ablauf von 2 Sekunden selbst. Es erfolgt zudem eine Meldung, wenn ein Roboter das erste Mal erfolgreich eine Ressource im RoboterTerminal ablegt. Zusätzlich zu diesen Message-Events, gibt es noch einen Ressourcen-Regen, der zunächst mit der Zeit zunimmt und schließlich versiegt. Auch hierauf wird der Spieler mit Nachrichten hingewiesen.

Kapitel 3

Die Entwicklung

3.1 Vorgehen

Die Planung: Im Gegensatz zum mehr oder weniger exakten Nachbau eines existierenden Spiels oder einer bekannten Anwendung, war klar, dass das Spielprinzip nur im Grundriss bekannt war und während der Umsetzung konkretisiert werden musste. In der Folge bedeutete dies einen erhöhten Aufwand, da nicht nur Umsetzungsfragen zu klären waren, sondern auch Entscheidungen zum Game-Design getroffen werden mussten. Es wurden zunächst zwei Phasen der Umsetzung festgesetzt:

- **1.Phase** - Die Erstellung eines ersten Prototypen, der mit wenig Funktionalität bereits eine Codestruktur vorgeben und die interaktive Testbarkeit weiterer Implementation herstellen sollte.
- **2.Phase** - Der Ausbau der Struktur, sowie das Hinzufügen der kompletten Grundfunktionalität und der gewünschten Features.

Die erste kritische Phase musste möglichst früh gemeistert werden, um das Projekt beherrschbar zu machen. Im Hinblick auf diese Forderung wurden geeignete Entwicklungskonzepte gewählt. Im Endeffekt wurde die Kombination mehrerer Softwareentwicklungsparadigmen angestrebt:

- Rapid Prototyping
- Domain Driven Design
- Test Driven Design

Schnelles Prototyping und das Augenmerk auf einer sauberen Modellierung des Spielkonzeptes mit einer ausgiebigen Nutzung objektorientierter Vorge-

Aufwandsschätzung:

1. Implementierung Domain-Model (spielrelevanten Entitäten):	
Feld, Karte:	2h
Spieler, Roboter	2h
2. basale graphische Konzeption I (als Grundlage für einen ersten Prototypen)	
einzelne Felder:	2h
Spieler,Ressourcen:	1h
Implementierung GUI I:	
Kartenanzeige:	3h
RessourcenMenu	2h
technische (Hilfs-)Klassen I:	
Modellierung des Spielzustandes:	2Std
Eventhandling:	3Std
Spielfluss I (Controller):	
Karten-Navigation:	3h
Ressourcen-Abbau durch Spieler:	1h
genereller Spielablauf, Integration Gui/Controller:	3h
— Milestone: Erster Prototyp, spielbares einfaches Level	
Implementierung GUI II:	
Roboter-Terminal:	3h
Feinschliff Gesamtfenster:	1h
basale graphische Konzeption II	
Roboter, Roboter-Terminal(auf der Karte):	1h
weitere Felder:	1h
Spielfluss II (Controller):	
Roboterbewegung, Erweiterung des Spielablaufs:	5h
Roboterherstellung:	3h
einfache Roboterprogrammierung:	5h
technische (Hilfs-)Klassen II:	
Speichern, Laden des Spielzustandes:	3h
Spielfluss III:	
Kollisionsbehandlung:	1h
Erweiterung der Roboterprogrammierung:	2h
Erweiterungen	(+Xh)
Gesamt	49+Xh

Abbildung 3.1: die im Projektplan enthaltene Aufwandsschätzung

hensweisen versprachen die zügige und nachhaltige Umsetzung eines Prototyps. Die laufende Kontrolle mittels Unit- und Integrations-Tests schien eine perfekte Ergänzung.

Als IDE wurde Eclipse gewählt, da alle Teammitglieder mit dieser Entwicklungsumgebung bestens vertraut waren. Die wichtigsten Tasks wurden zusammen mit der initialen Projetbeschreibung als Liste festgehalten und sollten erst im Laufe der Umsetzung verteilt werden. Zur Koordination wurde ein google-Spreadsheet verwendet, in dem die Entwickler sich für eine Aufgabe eintragen und ihren Stand dokumentieren konnten.

Da git zur Versionskontrolle genutzt werden würde, sollten die Mitglieder ihren Arbeitsfluss nach dem Git-Flow Prinzip ausrichten: Es gibt einen Develop und einen Master-Branch. Das grundsätzliche Setup und fertige Einzelfeatures werden direkt auf Develop committet. Der Stand auf dem Develop Branch muss hierbei immer stabil bleiben und keine weitreichenden Fehler enthalten, die die Programmausführung stören. Damit ist für alle Mitglieder eine solide Entwicklungsbasis gesichert. Die Arbeit an einzelnen Features findet auf entsprechenden Feature-Branches statt. Ist ein Feature fertig und gereviewt, wird es gefinisht (i.e. in Develop gemerged). Erreicht der Develop-

Branch einen Stand, den festzuhalten sich lohnt, wird eine Release auf dem Master-Branch erstellt und mit einem Tag versehen ('semantic versioning' war hierbei intendiert).

Die Realität: Tatsächlich hat sich das Prinzip des Rapid-Prototyping bei der Umsetzung bewährt. Das Vorgehen war dabei explorativ, experimentell und evolutionär zugleich. Die Domain und die grundlegenden technischen Klassen wurden zügig ausgebaut, auch wenn dabei viele Klassen enstanden, die später entfernt werden konnten. Darunter zum Beispiel auch ein Abstract-Controller, der nie mit Funktion gefüllt wurde.

Letztlich war insbesondere der Bereich der Domain überstrukturiert und musste später teilweise zurückgebaut werden. Die hohe Modularisierung war von großem Vorteil. Es gab bei der Umsetzung keine Sackgassen und alle Refactorings konnten in überschaubaren, kleinen Schritten durchgeführt werden, ohne dass immer *alles* angefasst werden musste.

Die Domain lag stets im Fokus und wurde angelegt, bevor es galt Funktionalität in einem Bereich herzustellen. Dadurch war es möglich, das Spielprinzip immer früh genug zu konkretisieren, auch im Hinblick auf die Komplexität der nötigen Implementation, die oft erst beim explorativen Konzipieren der fachlichen Klassen hervortrat.

Der Prototyp war dann auch schneller als erwartet fertig. Es wurden lediglich 20 der geplanten 24 Stunden benötigt. Allerdings musste der anfängliche Taskplan häufig umfassend geändert werden. Die Zuständigkeiten wechselten durchgängig, sodass einige Aufgaben von beiden Teammitglieder im Wechsel bearbeitet wurden. Insgesamt hat die weitere Ausarbeitung dann merklich mehr Zeit als geplant gekostet, wobei der Zusatzaufwand mindestens auf ca. 100-150% des geplanten Gesamtaufwands zu schätzen ist.

Der Workflow mit git-flow hat gut funktioniert, auch wenn ein Großteil der Arbeit direkt auf dem Develop-Branch geleistet wurde, ohne die Nutzung von Feature-Branches. Bei einem zweiköpfigen Team war dies unproblematisch. Die Änderungen wurden meist erst gepusht, wenn ein Feature funktional fertig war. Die Ausführbarkeit des Developstandes war stets gegeben. Merge-Konflikte wurden durch häufiges Pullen von der Remote und kleinteilig angelegte Features bzw. eine schrittweise Umsetzung abgemildert.

Die laufende Kontrolle mittels Unit-Tests konnte erst spät und nicht vollständig umgesetzt werden. Durch das explorative Vorgehen und die hohe Modularisierung wurde die konkrete Struktur häufig und mit Leichtigkeit ummodelliert, wodurch der Aufwand zur Anpassung der Tests unbeherrschbar wurde. Die ersten Tests wurden daher entfernt und wohl strukturierte Tests erst nach Ablauf der ersten Phase mit dem Hervortreten eines stabilen Prototyps hinzugefügt (siehe Kapitel 3.4 Unit Testing).

Zur Zeit ist 'BeepBoop' in der Version 1.0.0 als git-Repository verfügbar.

3.2 Codeorganisation

Die Planung: Bei der Codeorganisation wollten wir uns an gängigen Pattern orientieren. Allem zugrunde liegen sollte das MVC Pattern. MVC steht für „Model - View - Controller"und die Zuordung aller Klassen zu einer dieser Kategorien sollte eine gewisse Grundordnung schaffen. Neben MVC haben wir uns eingangs lediglich noch für das Observer Pattern entschieden.

Die Realität: Bei der Realisierung der Codeorganisation haben wir uns tatsächlich weitestgehend an unseren Plan gehalten. Die meisten unserer Klassen lassen sich in eine der drei MVC Kategorien einteilen.
Ins Model gehören Klassen, die konkrete Dinge verkörpern, bei BeepBoop zum Beispiel die Klasse BasicRobot. Eine Instanz der BasicRobot Klasse hält für den Roboter wichtige Werte, wie zum Beispiel seine Position im Level und seine Ladung.
Unter View fallen Klassen, die sich mit der (grafischen) Darstellung des Programminhalts befassen. Die Darstellung der Roboter geschieht sowohl über die Klasse LevelUI, die alle sichtbaren Roboter auf die Karte zeichnet, als auch über die Klasse RTManageUI, in der einige der in den Roboter Instanzen gehaltenen Werte, namentlich sein Programm, sein Error Log und seine Ladung, angezeigt werden können.
Controller sind die Bindeglieder. Sie sind die Kanäle für die Kommunikation der Klassen untereinander. Die BeepBoop Controller übernehmen auch einiges an Funktionalität, da sie z.B. mittels Actions (die entsprechenden Methoden sind nach dem Schema `[name]Action` benannt) die Domain-Klassen manipulieren, die in der Spielewelt interagieren. Der RobotController ist desweiteren auch für die Verarbeitung der einzelnen Commands, aus denen sich ein Roboterprogramm zusammensetzt, zuständig und stellt eines der Herzstücke der Implementation dar.
Neben den Klassen, die sich aus dem MVC Pattern ergeben haben, haben

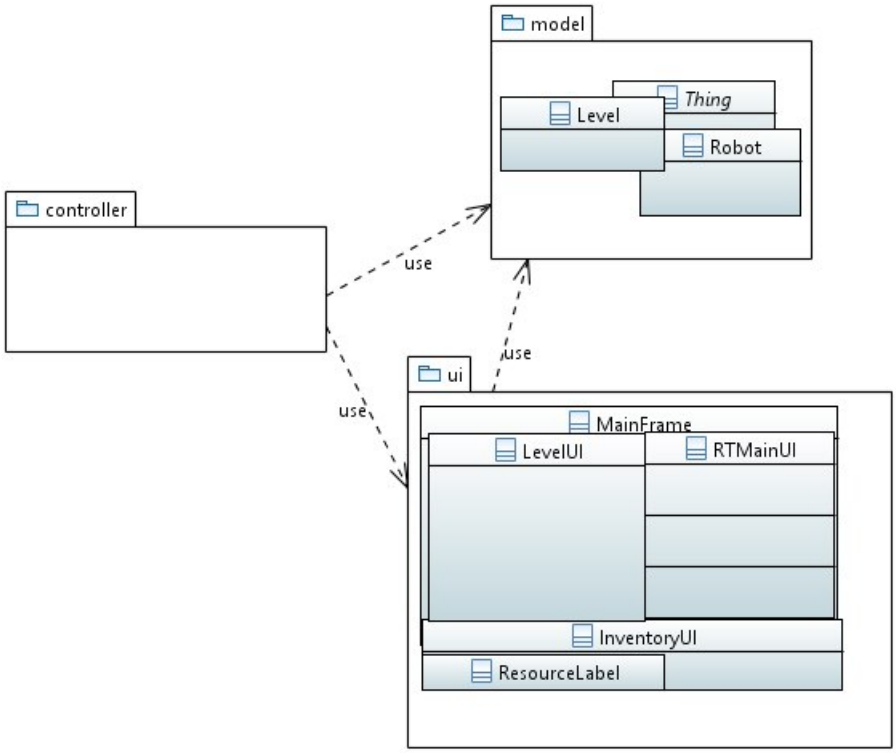

Abbildung 3.2: Dieses UML Diagramm zeigt die anvisierte Aufteilung von BeepBoop Klassen in die drei MVC Kategorien. Der Inhalt des ui-Packages soll hier als Skizze der anfänglichen Frontend-Konzeption dienen

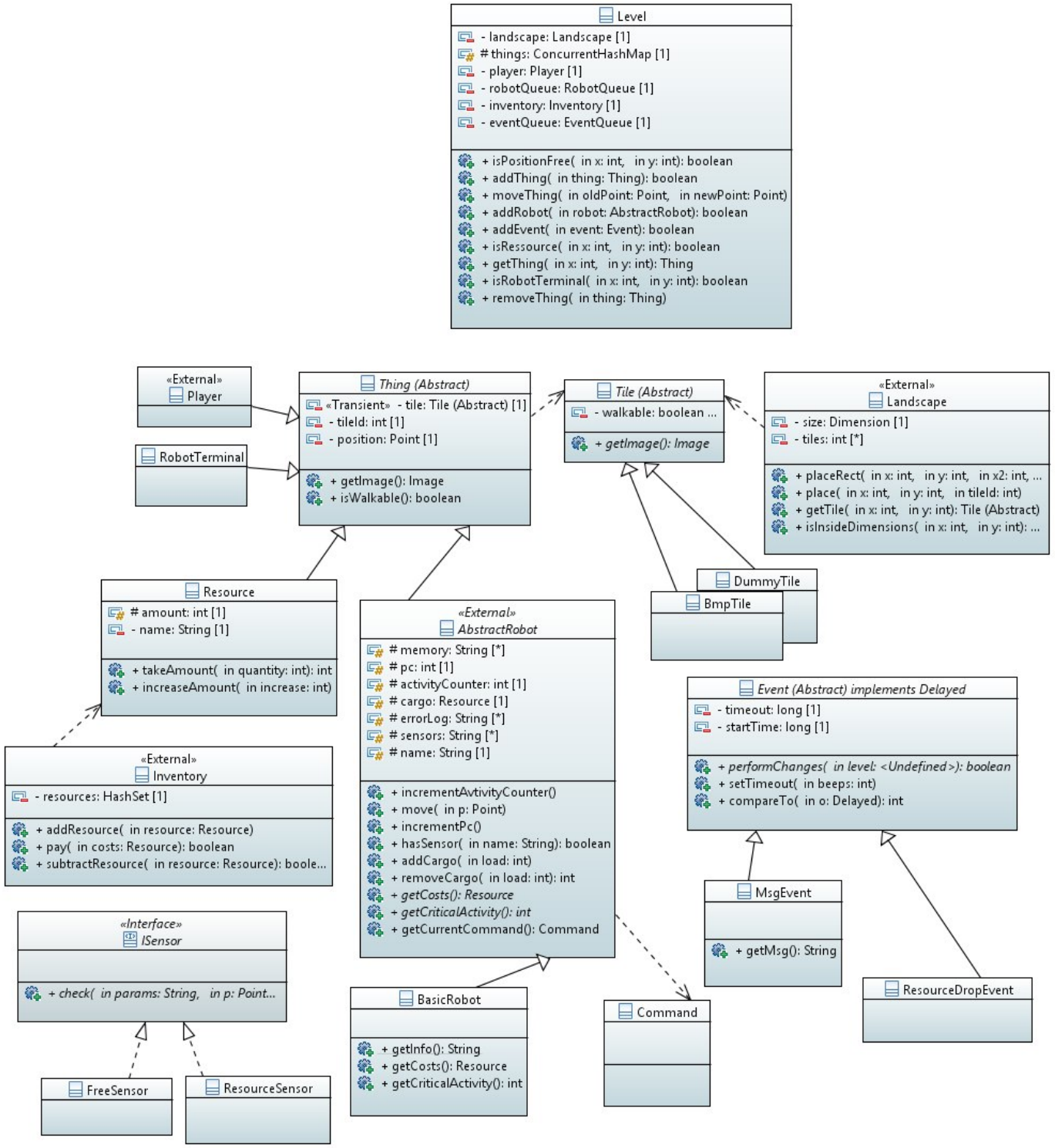

Abbildung 3.3: Dieses UML Diagramm gibt einen Überblick über das Model.

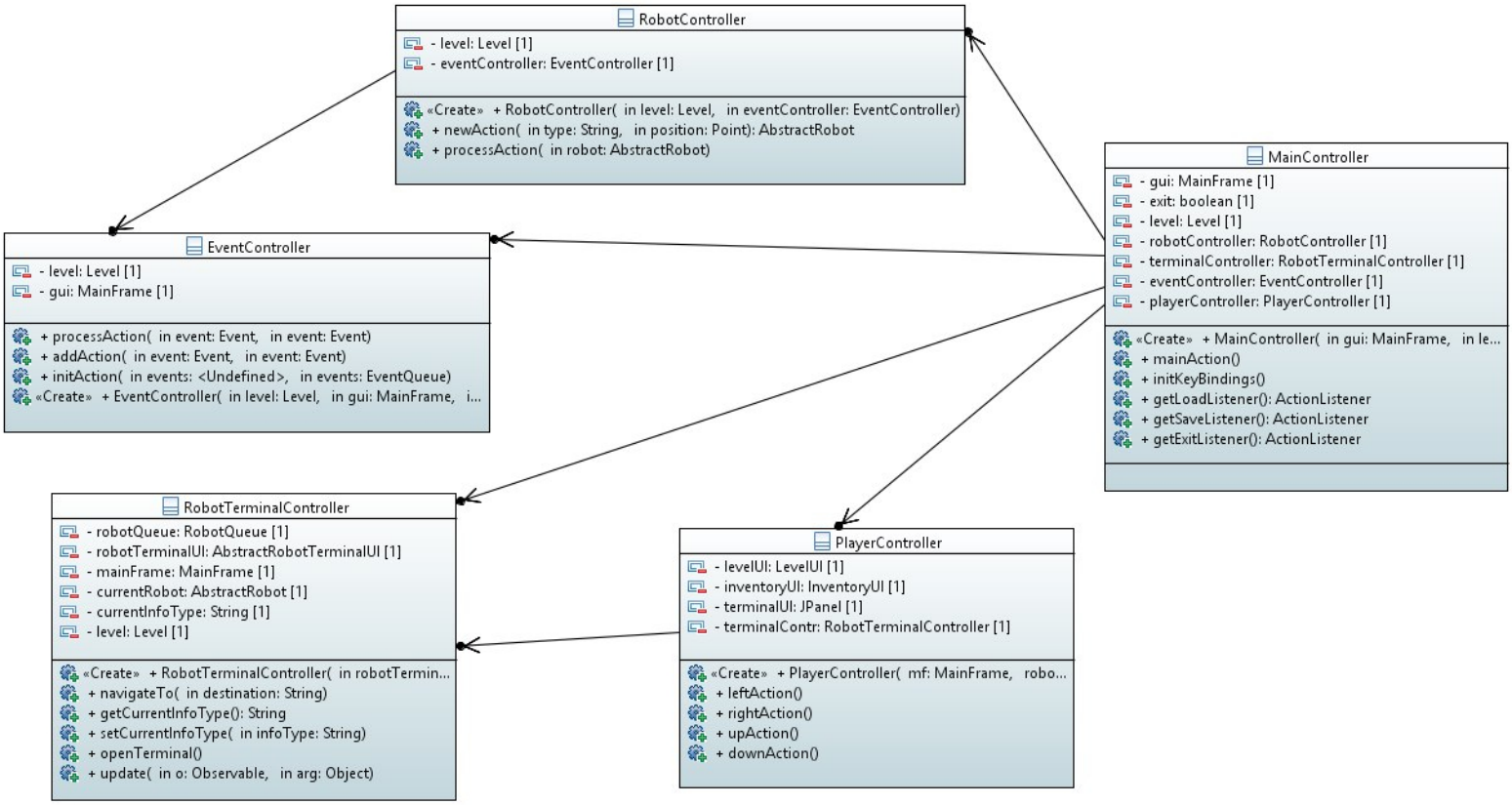

Abbildung 3.4: Dieses UML Diagramm zeigt die Abhängigkeiten der Beep-Boop Controller untereinander

wir auch noch ein paar Service Klassen. Sie bieten Zugriff auf Ressourcen. Die RobotQueue bietet zum Beispiel Zugriff auf alle Roboter. Über die TileFactory bekommt man Zugriff auf Tiles. Die Tiles halten jeweils eine kleine Grafik. Aus ihnen wird unter anderem in der LevelUI die Karte zusammengesetzt. Um den Speicher zu entlasten, existiert jedes Tile nur einmal.
Um das zu garantieren, haben wir bei der TileFactory das nächste Pattern verwendet, nämlich das Singleton Pattern. Es bewirkt, dass nie mehr als eine Instanz der TileFactory existiert. Benötigt man eine TileFactory, bekommt man diese nur über die öffentliche getInstance Methode. Der Constructor ist privat. Das Singleton Pattern kommt auch beim SensorService zur Anwendung.
Zu guter Letzt haben wir, wie geplant, das Observer Pattern genutzt. Es arbeitet mit zwei Typen von Klassen. Klassen, die das Interface Observer implementieren sind Klassen, deren Instanzen auf Zustandsänderungen von Instanzen anderer Klassen reagieren sollen. Die Klassen, auf deren Änderungen reagiert werden soll, implementieren das Observable Interface.
Bei BeepBoop implementiert zum Beispiel der BasicRobot das Observable Interface. Ist man im Manage-Submenü des Robot Terminals, so ist der RobotTerminalController, der das Observer Interface implementiert, stets bei dem ausgewählten Roboter als Observer angemeldet. Sobald sich nun das Error Log des Roboters ändert, beispielsweise, weil es einen neuen Eintrag erhält, wenn ein Roboter versucht durch eine Wand zu laufen, so wird dies allen angemeldeten Observern, also dem RobotTerminalController, gemeldet, so dass dieser veranlasst, dass das in der GUI angezeigte Error Log zu aktualisieren ist.
Kurzzeitig kam die spontane Idee auf, mittels Strategy Pattern eine Art Baukastensystem für die Erstellung von Submenü-GUI-Klassen zu erstellen. Da die Menüführung letztendlich viel weniger komplex ausfiel, als zwischenzeitlich gedacht, wurde diese Idee jedoch relativ schnell wieder verworfen.
Selbstverständlich wurde der Code in Java-Packages organisiert. Für jede Gruppe des MVC-Modells gibt es ein eigenes Sub-Package ('.model', '.controller', '.ui'), plus eines für die Services. Die Tests haben ein eigenes Sub-Package und nur die App liegt im Haupt-Package 'beepBoop'.

3.3 Arbeitsteilung

Die Arbeitsteilung verlief nach dem dem Prinzip „Jeder macht, was er gerade will“. Das hat mithilfe des Google Sreadsheets, auch tatsächlich sehr gut geklappt. Letztendlich haben wir beide an so ziemlich allem gearbeitet; es lassen sich aber ein paar Schwerpunkte ausmachen:

Pawel Rasch:

- Roboter Programmierung
- Events
- Gameloop

Tim Runge:

- Spielstand Persistierung
- Testing
- Robot Terminal UI

3.4 Unit Testing

Tests sind ein essentieller Bestandteil der Arbeit an einem Softwareprojekt. Um hilfreich zu sein, müssen Tests allerdings ein paar Anforderungen erfüllen: sie sollten lesbar, wartbar und vertrauenswürdig sein. Um diese drei Eigenschaften für unsere Tests zu realisieren haben wir uns an ein paar Richtlinien gehalten, die wir im folgenden kurz erläutern wollen. Sie geben unseren Tests eine einheitliche, übersichtliche Struktur.

Tests sollen transparent sein Um den Sinn eines Tests zu verstehen, muss es reichen, den Testcode zu lesen. Dafür haben wir als Erstes auf mmagic literals"verzichtet. Stattdessen haben wir Werte bewusst in Variablen gespeichert. Was für einen im Test verwendeten Wert wichtig ist, lässt sich weitestgehend aus dem Namen seiner Variablen schließen. Der Leser muss nun nicht mehr nachdenken, wofür irgendeine Zahl steht. Ob hier einfach irgendein Wert (someValue) benötigt wird, oder ob die Wertmenge, aus der geschöpft wird, eingeschränkt ist (someNegativeValue, numberOfResources), lässt sich direkt sehen.
Die Namen der Tests folgen einem klaren Muster. Kennt man das Muster, gibt auch hier der Name auf einen Blick alle nötigen Informationen preis:
nameDerZuTestendenMethode_kontextDesTests_erwartetesErgebnis
Schlägt der Test isUseful_itIsNotUseful_returnFalse fehl, muss man sich keine Gedanken mehr darum machen, was fehl geschlagen ist, oder was eigentlich hätte passieren sollen, sondern kann sich gleich ums warum kümmern.
Jeder Test ist unterteilt in Einleitung, Hauptteil und Schluss, beziehungsweise Arrange, Act und Assert. Im Arrange Teil wird der benötigte Kontext

geschaffen. Variablen werden deklariert und initialisiert und Objektzustände manipuliert, bis die für den Test nötigen Bedingungen geschaffen sind. Auch dies wieder möglichst logisch gegliedert. Im anschließenden Act-Teil findet nun der Aufruf der zu testenden Methode statt, dessen Ergebnis im Letzten, im Assert-Teil, überprüft wird.

Tests sollen trocken sein Damit die Struktur der Tests klar bleibt und ins Auge sticht, benutzen wir Hilfsfunktionen. Sie wirken nicht nur Coderedundanzen entgegen sondern verstecken auch für den Leser unnötige Details. Was allerdings nie versteckt werden sollte, ist der Aufruf der zu testenden Methode, denn dieser ist der elementare Teil des Tests.

Tests sollen flach sein Bei myDraw haben wir noch einen Test pro Methode geschrieben. Bei BeepBoop hingegen haben wir, wo es möglich war, einen Test pro Assert. Damit hat jeder Test eine klare Aufgabe. Mehrere Asserts in einer Testmethode haben außerdem zu Folge, dass Code, der nach einem Assert steht, bei einem Fail des Asserts überhaupt nicht mehr ausgeführt wird.

Tests sollen präzise sein Um die Aufgabe eines Tests klar zu halten, muss er möglichst präzise sein. Testet man zum Beispiel eine Methode, die einen Boolschen Wert zurückgibt, und ihrerseits drei Bedingungen überprüft, so lohnt es sich, neben einem Positiv-Test, für jede dieser Bedingungen je einen eigenen Test zu schreiben, in dem diese Bedingung nicht erfüllt ist.

Wir haben unser Testing bei BeepBoop, im Vergleich zu myDraw, qualitativ erheblich verbessert. Unsere Tests sind gut strukturiert und dadurch gut lesbar und wartbar, was sie wiederum vertrauenswürdig macht. Sollten wir das Projekt nach dem Praktikum noch weiterführen, wäre vor allem an der Quantität der Tests zu arbeiten, da sie bislang nur einen Teil des Programmcodes abdecken.

3.5 Probleme

Da wir mit zwei unterschiedlichen Betriebssystemen gearbeitet haben (Windows 10 und Linux Mint), haben wir eine Menge Probleme erwartet. Einige Probleme entsprangen dieser Erwartungshaltung, weil wir bei kleinen Ungereimtheiten sofort die OS-Diskrepanz verantwortlich gemacht und erstmal nur in der Richtung nach Fehlerquellen gesucht haben. Letztendlich hat Java

seinem Namen in Punkto Plattformunabhängigkeit allerdings alle Ehre gemacht. Lediglich die Betriebssystem bedingt unterschiedlichen Eclipse Versionen führten zu einem Problem, da Eclipse Neon kein JUnit 5 beherrscht. Die bereits geschriebenen Tests für JUnit 4 umzuschreiben war dann jedoch nicht allzu aufwändig.
Ein schwer zu greifendes Problem ergab sich im Verlauf der Umsetung der Level-Klasse, die als Sammelbehälter der aktuellen Landschaft und aller Dinge der Spielewelt dient. Hier wurde unter anderem eine `HashMap` genutzt. In sehr selten Fällen wurde dann bei der Abfrage der Values der Hashmap eine `ConcurrentModificationException` geworfen, ohne dass der Spielfluss nachvollziehbar gestört wurde. Es gelang tatsächlich nicht diesen Fehler kontrolliert nachzustellen. Der Rückgriff auf `ConcurrentHashMap` scheint das Problem behoben zu haben.
Wie erwartet ist der In- und OutPut der ErrorLogs bei sehr langen Logs sehr schwerfällig und bereitet bei einem Wechsel des Robots Probleme. Der Programmablauf wird nicht gestört, aber der Wechsel erfolgt sehr langsam. Dies ist (je nach Spielweise) nach ca. 20-30min der Fall. Ein ToDo für zukünftige Versionen.
Größere Probleme hatten wir tatsächlich nicht. Allerdings sind bis zuletzt kleinere Bugs aufgetaucht, die schnell behoben waren. Daher können wir uns nicht sicher sein, dass bei entsprechendem Level Setup und einer speziellen Roboter Programmierung nicht doch Fehler auftreten. Interaktives Testen wurde vielleicht nicht exzessiv genug betrieben.

Kapitel 4

Erkenntnisse/Epilog

Insgesamt sind wir mit unserem Ergebnis recht zufrieden. Die geplanten Features wurden gemeistert und nebenbei konnten noch einige neue Ideen in das Projekt eingebaut werden. Allerdings ist die Applikation für den zeitlichen Rahmen etwas zu umfangreich. Das Projekt hat erst kurz vor dem Abgabetermin einen Stand erreicht, der den Aufbau eines echten Demo-Levels erlaubt. Aufgrund der fehlenden Zeit konnte die Mächtigkeit der modularisierten Implementation nur im Ansatz genutzt werden. Ein Beispiel hierfür ist das `ResourceDropEvent`, welches mit wenig Mühe realisiert werden konnte und es erlaubt Ressourcen im Spielverlauf der Landschaft hinzuzufügen. Zusätzlich wäre zu den umgesetzten 'timed'-Events, noch 'triggered'-Events schön gewesen, die nicht nach Ablauf einer Zeit, sondern nach Eintritt eines Spielereignisses stattfänden.
Aufgrund der begrenzten zeitlichen Ressourcen konnte leider kein wirklich zufriedenstellendes, vollständiges End-Refactoring durchgeführt werden. ToDos sind hier vor allem ein Umbau der zu komplizierten RobotTerminal-Ui, welche ein Relikt des zwischenzeitlich eingesetzten Strategy Patterns darstellt, und eine noch konzentriertere Umsetzung der MVC und ActionController Pattern. Außerdem ist der Code nicht überall generisch genug gehalten, allerdings könnte hier mit jeweils wenig Aufwand refactored werden.
Der Workflow unseres Teams, das schon in vergangenen Semestern erfolgreich zusammengearbeitet hatte, hat sich noch weiter verbessert. Dazu haben unter anderem auch die in diesem Team erstmals verwendeten Tools, git als Versionskontrolle und Google Spreadsheet als Koordinationsbasis, beigetragen. Die Einrichtung eines Webhooks, der alle Teammitglieder über jeden neuen Push zur Remote per Chatnachricht informiert hat, war ebenfalls eine große Hilfe.
Für eine Fortführung des Projekts wären mehrere Schwerpunkte denkbar. Als erstes wäre wohl die Schaffung eines Level-Editors, oder zumindest eines

besseren Systems zur Level-Erschaffung, sinnvoll. Anschließend könnten unterschiedliche Level kreiert werden. Die Programmierung der Roboter sollte durch Hinzufügen von while-Schleifen oder labels für die Sprungbefehle erweitert werden. Denkbar sind natürlich interne Register, oder hochzählbare Counter. Wie bereits erwähnt bedarf es zudem noch einer Menge weiterer Sensoren für ein angenehmeres Spielerlebnis. Auch ein oder mehrere Spielziele sollten implementiert werden, um BeepBoop zu einem richtigen Spiel zu machen.