Architekturdokumentation
Mensch ärgere Dich nicht!

Martin Puse
853001

Marcel Pillich
863121

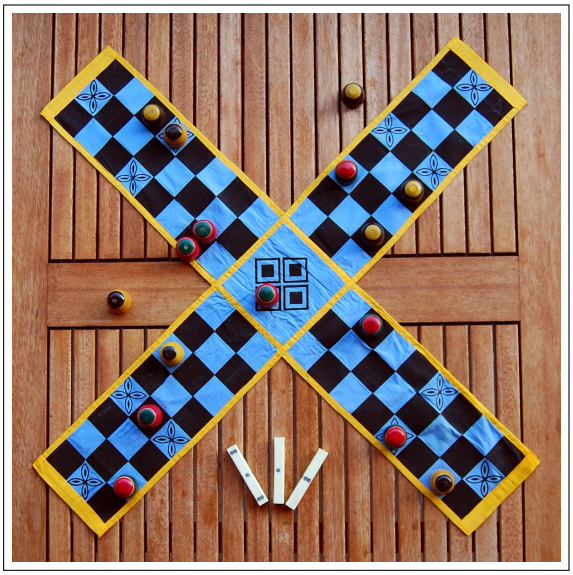

Fig. 1. Pachisi

***Abstract*—Die vorliegende Arbeit enthält eine detaillierte Beschreibung einer Java-Applikation für des Spiels "Mensch ärgere Dich nicht", welche für Spieleprogrammierer und Spieledesigner gleichermaßen geeignet ist. Ziel war es, dass der Programmierer sich um Erscheinungsbild und Regeln des Spiels kümmern kann, ohne selber Spielfelder designen zu müssen, wohingegen der Designer keine Programmierkenntnisse benötigt, um neue Spielfelder hinzuzufügen und zu testen. Zu diesem Zweck wurden eine Grammatik für die Spielfelder und ein Generator, welcher die Grammatiken verarbeitet und automatisiert Code erzeugt, erstellt. Die dahinterliegende Idee, die Spielfelder nicht auf Positions-, sondern Knoteninformationen zu basieren, wird ausführlich dargestellt. Schlussendlich werden Weiterentwicklungsmöglichkeiten des Prototyps präsentiert.**

I. Einleitung

"Mensch ärgere Dich nicht" ist ein Brettspiel für bis zu 4 Personen. Es zählt zu den Klassikern im deutschsprachigen Raum und wurde 1907/1908 von Josef Friedrich Schmidt in Anlehnung an das englische Spiel Ludo erfunden, die Ursprünge finden sich jedoch im indischen Spiel Pachisi. Erstmals 1910 erschienen, ging es 1904 in Serie. Mittlerweile existieren auch viele Abkömmlinge und Varianten mit veränderten Regeln oder Spielfeldern. Allen Varianten ist jedoch gemein, daßsie ein symmetrisches Spielfeld besitzen und analog entwickelt wurden. Man kann vermuten, dass mit asymmetrischen Spielfeldern experimentiert, dies jedoch nicht weiter verfolgt wurde, da es ungleich schwieriger ist, eine asymmetrische Konfiguration zu finden bei der die Chancengleichheit und somit der ausgewogene Spielspass für alle Spieler gegeben ist. Dennoch ist der Reiz dieses Szenarios nicht zu verachten, da die Freiheit bei der Gestaltung asymmetrische Spielfelder äußerst fruchtbar für die Entwicklung interessanter Spielvarianten, welche sich nicht in das relativ starre klassische Korsett einpassen müssen, ist. Die größte Hürde in diesem Bereich ist also die mühelose Erstellung eines experimentellen Spielfeldes, um dieses unmittelbar auf seine Spielspaßeigenschaften zu testen.

Zu diesem Zweck wurde eine Java-Applikation entwickelt, welche einen Spielfeldgenerator implementiert, der mittels einer DSL variable Spielfelder erzeugen kann. Im Verlauf der Architekturdokumentation wird sowohl auf die Besonderheiten der Implementierung, als auch auf die verwendeten Design-Patterns und den Code-Generierungsansatz eingegangen. Abschließend wird ein Fazit formuliert, das die Erfahrungen und Ergebnisse zusammenfasst.

II. Anforderungen an die Applikation

Grundlegend an die Anforderungen ist die Tatsache, dass das Programm für Programmierer und Designer gleichermaßen benutzbar sein muß. Während der Designer sich vollständig auf die Entwicklung interessanter Spielfelder konzentrieren können muß, d.h. man kann keine Programmierkenntnisse vom ihm verlangen, darf dies den Programmierer nicht daran hindern, die Applikation weiter zu entwickeln, also in den Bereichen grafisches Erscheinungsbild, Regeln, Künstliche Intelligenz und Spielfeld Veränderungen vorzunehmen. Aus diesen Gedanken heraus haben sich die grundlegenden Anforderungen gebildet. Die Applikation sollte möglichst unabhängig vom Betriebssystem lauffähig sein. Da eine starke Eigenschaft von Java die Portabilität und Lauffähigkeit auf vielen unterschiedlichen Geräten ist, wurde diese als Entwicklungsprogrammiersprache gewählt. Dadurch lassen sich auch Abkömmlinge des Prototypen (exemplarisch MobileApp) gut planen und umsetzen. Die verbindende Technologie zwischen Programmierer und Designer ist das ausgereifte Codegenerierungs-Tool Antlr, welches das *.csv*-Dateiformat von Haus aus beherrscht und für die programmierkenntnislose Erstellung von Spielfeldern geeignet ist. Diese Aufstellung des Technologiestacks erlaubt einen wechselseitigen Workflow ohne große Unterbrechungen. Der Designer kann mit

der aktuellen Version neue Spielfelder erzeugen oder bereits designte testen und Vorschläge zum Regelwerk machen. Der Programmierer kann solche Feature-Requests umsetzen und einspielen, woraushin der Designer wieder iterieren kann. Beide kommunizieren also vorrangig über die Grammatik des Spiels und sind nicht primär an technische Beschränkungen gebunden. Weiter kann der Programmierer kann unabhängig vom Designer an der graphischen Erscheinung der View und dem Verhalten der künstlichen Spieler arbeiten. Für den Designer ist eine einfache Bedienung wichtig, d.h. es wird nur von ihm verlangt mit der View zu kommunizieren und neue Spielfelder zu erzeugen, aber keine weiteren Deploy-Prozesse zu beachten.

- os unabhängig lauffähig - trotz development dauerhaft lauffähiger zustand (für designer) - keine programmierkenntnisse, um neue spielfelder zu erzeugen und zu testen - erweiterbarkeit der regeln - erweiterbarkeit für künstliche spieler - einfachste bedienung (nur maus + linke maustaste)

usage für designer: entwickele spielfeld, starte generator, starte applikation, vorschläge an den programmierer zur weiterentwicklung usage für programmierer: entwickele view erweitere regeln (controller: businesslogic, model: neue datenfelder) erweitere generator/grammatik nach implementierung designer neue version zur verfügung stellen

III. Genutzte Design Pattern

A. MVC

Für die Abbildung eines Brettspiels in digitaler Form bietet sich das klassische Model-View-Controller Pattern an.

Dabei ist zu beachten, das die Dreieckbeziehung im MVC nicht symmetrisch, sondern hierarchisch implementiert wird, da sonst die Verantwortlichkeiten von Model, View und Controller aufgeweicht werden und feste Kopplungen sowie Spaghetti-Code begünstigt werden. Konkret bedeutet das, das der Controller das Model und die View kennt, die View jedoch nur, um seine Callbacks auf asynchrone Events (z.B. Benutzereingaben) der View anzumelden. Als Controller darf er im Zuge des Spielablaufs im Model Veränderungen vornehmen. Die View kennt das Model, darf aber nur daraus lesen, um sich selbst, d.h. die grafische Darstellung der Applikation auf dem aktuellsten Stand zu halten. Das Model als Datenhalter kennt niemanden, stellt aber Lese- und Schreibfunktionen zur Verfügung. Diese Aufteilung ist noch weiter ausbaufähig, bei zunehmender Komplexität wäre es sinnvoll für die Einhaltung der Verantwortlichkeiten EventDispatcher-Pattern hinzuzufügen. Dadurch können das Model und die View Events bei Eingabeereignissen oder Datenveränderungen senden, ohne dass die lose Kopplung verlorengeht. Im Prototypen wurde sich dagegen entschieden, da ein einzelner Aufruf für die GUI-Aktualisierung pro Spielzug und die Anmeldung zweier Callbacks den Mehraufwand nicht rechtfertigten.

B. Singleton

Die Applikation soll stets ein lauffähiges Mensch-ärger-dich-nicht-Spiel repräsentieren, welche aufgrund des Designeraspekts ohne komplizierte Laufzeiteinstellungen auskommen oder Kommandozeilenparametern zu starten sein muß. Es ist also nicht erwünscht, das mehrere Instanzen der Hauptklassen erzeugbar sind, da dies zu kompliziertem Initialisierungscode mit entsprechend notwendigen Konfigurationen von außen in den Toplevelschichten führen würde. Im Code wird dies dadurch reflektiert, dass die Hauptklassen von Model, View und Controller als Singletons implementiert sind. Die Eleganz dieser Variante zeigt sich in der übersichtlichen `Main`-Klasse, in der nur noch die Singletons instanziiert und miteinander verknüpft werden. Sämtliche Weiterentwicklung des Programms kann innerhalb der MVC-Komponenten erfolgen, ohne das die `Main` angepasst werden muß.

IV. Implementierung

Die Applikation wurde in einem 2er-Team entwickelt. Als erstes wurde die Grammatik definiert, damit dann parallel ein Grundgerüst der Applikation $MenschAergerDichNicht$ entworfen und der Generator $MADPlayfieldGenerator$ geschrieben werden konnte. Nachdem beide Projekte minimal lauffähig waren, haben sich die Entwicklungsrollen iterativ vermischt, bis der Prototyp ausreichend angereichert und getestet war.

Für die einfache und konfigurationslose Verknüfpung der beiden Projekte wurde ein relativer Pfad, welcher lediglich verlangt, dass sich beide Projekte nebeneinander in einem Ordner befinden und ein fester Name $GEN\_PlayfieldCreator.java$ für die erzeugte Quellcodedatei gewählt. Dadurch konnte die Code-Generierung mit einem einzigen Befehlpermanent angewendet werden.

A. Grammatik/DSL

Die Grammatik des Spiels enthält Symbole für alle möglichen Spielfeldtypen und deren Verknüpfungen zum Nachfolgefeld. Es können Spielfelder für 2 bis 4 Spieler definiert werden. Das reichte für den Prototyp aus und kann ohne besondere Schwierigkeiten erweitert werden. Die Spielfeldtypen eines klassischen Mensch-ärger-dich-nicht-Spiels sind wie folgt im Code und in der DSL realisiert:

TABLE I
TILETYPES

Java	DSL
HOME	H[p]
START	S[p]
GOAL	G[pn]
WAY	W[n]
TOGOAL	W[n]G[n]
NONE	NO

Das Kürzel n steht für die Knoteninformation, welches Feld dem betreffenden folgt, dies kann eine der 4 Himmelrichtungen N,S,W,E sein. Das andere Kürzel p hingegen enthält die Spielernummer 1-4. Einen kleinen Spezialfall stellt der Typ $TOGOAL$ dar, das dies ein Spielfeldtyp ist, welcher zwei Nachfolgeknoten enthält. Der zweite ist die Abzweigung in die Zielfelder des jeweils passenden Spielers. Durch die

knotenbasierte Definition der Spielfelder ist dies jedoch problemslos implementierbar (siehe **??**) und auch erweiterbar. Da das *.csv*-Format zeilenbasiert ist, wurde für den Prototypen entschieden, nur matrixartige Spielfelddefinitionen zuzulassen. Dies wird im unten erklärten `PlayFieldMetaListener` überprüft.

B. Generator

Um aus den *.csv*-Dateien eine *.java*-Datei zu erzeugen, musste ein zweistufiger Parser geschrieben werden. Der erste Parser `PlayFieldMetaListener` ermittelt Metadaten des Spielfeldes, die der zweite Parser `PlayFieldSemanticListener` benötigt, um den spielfelderzeugenden Code generieren zu können. Dazu wird die *.csv*-Datei einmal durchlaufen um die Metadaten zu akkumulieren. Zu diesen gehren die Anzahl der Spieler, die gleichzahligen Home- und Endpositionen der jeweiligen Spieler, das Vorhandensein genau eines Startfeldes pro Spieler sowie die Anzahl der Zeilen und Spalten der *.csv*-Definition. Etwaige Fehler werden mit einer Abbruchsmeldung quittiert. Sind alle Metadaten gesammelt, was erst nach einem vollstndigem Scan garantiert ist, kann der `PlayFieldSemanticListener` gestartet werden, da dieser die Metainformationen bentigt, um den Code fr die Spielfeldverknpfungen mit den nun bekannten Indizes zu generieren. Die generierte Datei enthlt danach die Zugriffsfunktionen

`getNumRows`, `getNumColumns`, `getNumPlayers` und `getNumPiecesPerPlayer` fr die Metadaten und die spielfelderzeugende Funktion `createPlayfield`, welche wie in **??** benutzt werden. Wie man an einer beispielhaften Zeile `playfield.getTile(0, 6)//.setType(TileType.START)//.setNext(playfield.getTile(1, 6))//.setPlayerID(2);` innerhalb dieser Methode sehen kann, werden die matrixartige Struktur der *.csv*-Datei und die gewonnenen Metadaten verwendet, um die Knotenverknpfungen zwischen den einzelnen Spielfeldern zu erstellen. Nach dieser Zuordnung kann auf die Koordinatenwerte der Felder vollstndig verzichtet werden.

meta parser: liest spielfelddatei und ermittelt metadaten, also daten das ganze spielfeld betreffend und positionsunabhängig - spieleranzahl - ein startfeld pro spieler - gleiche anzahl an home und goal feldern pro spieler - höhe, breite der spielfeldmatrix - daten werden als getter im genierten code bereitgestellt

semantic parser - liest die einzelnen felder - verknüpft die knoten - generiert den erzeugungscode des spielfeldes

Die so generierte Datei enthält die Methoden `getNumRows`, `getNumColumns`, `getNumPlayers` und `getNumPiecesPerPlayer` mit den ermittelten Daten des Meta-Parsers sowie die Methode `createPlayfield` des Semantic-Parsers, welcher eine zu befüllende `Playfield` Instanz bergeben wird.

C. Applikation

Im folgenden werden die einzelnen Klassen der Applikation vorgestellt.

1) Game: Die Klasse `Game` ist das Herz der Applikation. Als Hauptcontroller hält er Referenzen zum Subcontroller `PlayerController`, zur View `GUI` zum Model `Playfield`. In der `Main`-Methode wird die Applikation über `Game game = Game.getInstance(); game.init(playerController, playfield, gui);` zusammengesetzt. Im `Game`-Controller finden sich Methoden zur Eingabebehandlung: mit `onStartButtonClicked` kann ein neues Spiel begonnen werden. Mit `onRollDiceButtonClicked` wird Würfeln der menschlichen Spieler verarbeitet und mit `onMouseClicked` werden Klicks auf das Spielfeld verarbeitet. Dabei handelt es sich um Spielfigurbewegungen der menschlichen Spieler. Zum Steuern des Spielablaufs existieren die Methoden `startGame`, mit der ein neues Spiel initialisiert wird. `checkForWin` prüft nach jedem Spielzug, ob ein Sieger feststeht und beendet gegebenenfalls das Spiel durch `endGame`. `turn` leitet einen Spielzug ein. Im Falle eines menschlichen Spielers wird anschließend auf Eingaben gewartet, bei einem künstlichen Spieler wird das Würfeln automatisch ausgeführt und die KI über `turnKI` zur Spielfigurenbewegung angesteuert. Die zufallsgenerierten Würfelwerte werden von `rollDice` zurückgegeben. Für die Logik innerhalb des Spielablaufs existieren die Methoden `prepareBonusRoll`, welche gesondert nach einer gewürfelten 6 angesteuert wird und im Falle einer anderen gewürfelten Zahl `prepareNextPlayer`, wenn der entsprechende Spieler seinen Zug abgeschlossen hat.

Die zentrale Methode `tryMove` wird angesteuert, wenn eine Spielfigur von einem -gleichgültig ob menschlich oder künstlich- Spieler bewegt werden soll. In ihr werden die Regeln zur validen Bewegung der Spielfigur überprüft, der Zug wenn möglich ausgeführt (wodurch sich das Model ändert) und die Validität als `boolean` Wert zurückgegeben. Durch den Rückgabewert wird gesteuert, ob im Falle einer KI eine andere Spielfigur bewegt oder wieder auf Mauseingaben des Benutzer gewartet werden soll. War der Zug valide, wird zum nächsten Spieler im Spielverlauf weitergegangen. Unterfunktionen von `tryMove` sind `cantMoveOut`, welche die Zwangsregel, dass man eine Spielfigur -wenn möglich- heraussetzen muss und `isValidTarget`, welche überprüft, ob das Zielfeld einer Spielfigur ein valide Position darstellt. Auf die letztgenannte Methode soll kurz genauer eingegangen werden, da sich in ihr die Vorteile des knotenbasierten Spielfeldes zeigen. Zum Ermitteln eines legales Spielzugs der Spielfigur wird zuerst das Feld, auf dem die Spielfigur sich befindet, referenziert und nachfolgende die Methode `getTargetTile` auf dem referenzierten Spielfeld aufgerufen, dessen Implementierung in Abbildung **??** zu sehen ist. Mithilfe der Nachfolgeknoteninformation wird solange der nächste Knoten referenziert, bis entweder kein Nachfolgeknoten existiert (im Falle des letzten Zielfeldes eines Spielers) oder die gewürfelte Schrittweite erreicht ist. Zu beachten ist, daßkeinerlei planaren Positionsinformationen benötigt werden. Dadurch ist es möglich, Wege durch völlig frei gestaltete Spielfelder zu suchen. Weiterhin lassen sich zwanglos neue

Spielfeldtypen für exotischere Spielvarianten hinzufügen.

2) PlayerController: Der `PlayerController` dient lediglich der Verwaltung der Spieler. Durch diese zusätzliche Kapselung wird jedoch der `Game`-Controller nicht unnötig mit spielablauffremden Methoden aufgebläht, wenn die Spielerverwaltung durch erweiterte Spielmodi komplizierter wird. Dazu gehört beispielsweise die Anmeldung der Spieler (menschlich oder künstlich) beim Spielstart und die spätere Abmeldung für Spielmodi mit mehreren Siegerplätzen.

View: GUI -verantwortlich für den Rahmen, fenstergröße usw PlayfieldPanel - rendering des spielfeldes (tiles, pieces)

Model: Playfield - alle tiles Tile - knoteninformationen - referenzinformation über pieces Player - enthält seine pieces Ableitungen: KIPlayer/HumanPlayer

implementierungsvorteile durch knotenbasierte spielfeldelemente

-erzeugt eleganten regel code durch tiletypen und lookahead function -einfache erweiterung neuer tiletypen und regeln - natürlich abbildung der regeln in code

Typendefinitionen: tileType

beispiele zeigen: if (tile.TILETYPE == START && tile.hasPiece())) // cant move to start, its not free

V. Fazit

knoten ansatz sehr gut geeignet für erweiterung/generalisierung der spielfeldgenerierung einsatz passender pattern an den richtigen stellen für weiterentwicklung der software (ki subclass, spielregeln)

grammatik zeilen/spalten gebunden (keine planar frei plazierbare tiles ohne x, y beschränkung) aber möglich, tiles nur knotengebunden, dh zusätzlicher editor für graphisches design von csv-vorlagen möglich verschieben von graphischen elementen hat keinen einflußauf knotenverknüpfungen

Acknowledgment

The authors would like to thank...

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.