

Entwicklungsprojekt: Audit 3

Meike Jungilligens, Mauricio Köppen, Fabian Ngo



Gliederung

1. **Durchgeführte PoCs**
 - **Persistente Speicherung der Daten**
 - **Entscheidungsmechanik mit Einfluss auf Parameter**
2. **Modellierungen**
3. **Datenbank**
4. **Beispielhafter Spieldurchlauf**



Proof of Concept 1

PoC: Entscheidungsmechanik mit visuell erkennbarem Einfluss auf festgelegte Parameter

Implementierung einer interaktiven Entscheidungs-Mechanik, die den Spielern erlaubt, verschiedene Entscheidungsoptionen zu erkunden und die Auswirkungen ihrer Entscheidungen auf den Bauernhof zu verstehen. Jede Entscheidung führt zu einer Steigerung oder Minderung eines oder mehrerer Parameter (Kapital & Umweltfaktoren (Grundwassersauberkeit, Bodenvitalität und Biodiversität)).

EXIT-Kriterium

Dem Spieler wird eine Entscheidung mit zwei Antwortmöglichkeiten im Interface mit einer textuellen Beschreibung angezeigt. Eine der Antwortmöglichkeiten kann ausgewählt werden und senkt oder steigert bestimmte Parameterwerte. Die Änderung ist visuell in Form der Parameter-Icon-Farbe (grün, gelb, rot) und -Füllmenge erkenntlich.

FAIL-Kriterium

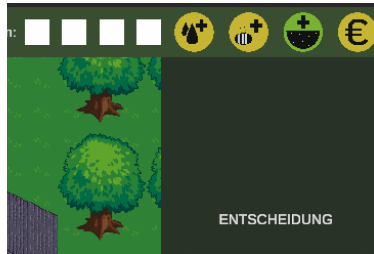
Die Entscheidungen beeinflussen die Parameter nicht oder fehlerhaft. Beispiel: Der Verkauf einer Maschine führt zu Geldverlust.

FALLBACK

Überprüfung der programmierten Funktionen für die Berechnung der Parameterwerte (& -änderungen) sowie Sicherstellung einer korrekten Kopplung in der Game Engine zwischen Parameterwert und UI-Anzeige (Parameter-Icon-Farbe, Assets)

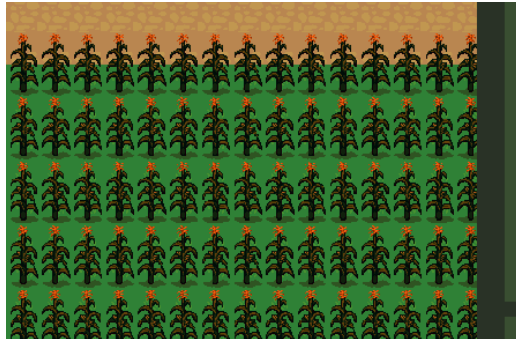
(Recap PoC1)

Entscheidungsmechanik- visuell erkennbarer Einfluss auf Parameter



Ausgewählte Antwort wirkt sich sofort auf die Farbe der Parameter-Icons aus (Farbverlauf von Grün zu rot)

Extreme Wertänderungen auch in Assets/Sprites; hier: welkender Mais



Proof of Concept 2

PoC: Persistente Speicherung der Parameterwerte (& evtl. Farm-States)

Dem Spieler ist es möglich, mehrere Entscheidungen hintereinander durchzuführen. Die in vorherigen Entscheidungen geänderten Parameterwerte werden fortlaufend weiter geändert und persistent gespeichert. Dafür wird eine lokale Datenbank (SQLite) genutzt.

EXIT-Kriterium

Die Speicherung nach der erfolgreichen Berechnung (PoC 2) erfolgt über eine korrekte Aktualisierung der Werte in der Datenbank. Auch das Unterbrechen des Spiels/Programms setzt die Parameter nicht auf Default-Werte zurück.

FAIL-Kriterium

Die Parameteränderungen werden gar nicht oder nur flüchtig gespeichert und sind nach Neustart des Programms wieder auf ihre Anfangswerte zurückgesetzt.

FALLBACK

Untersuchung der Verknüpfung zwischen Datenbank und Programmcode. Überprüfung der Richtigkeit der SQL-Befehle. Eventuell Einbinden einer alternativen Datenbank.


(recap PoC2)

Entscheidung

<u>entscheidung_id</u>	<u>entscheidung_phase</u>	<u>antwort_id1</u>	<u>antwort_id2</u>	entscheidung_text
1	Anbau	01	02	"Du kannst den Boden deines Feldes düngen, um den Ertrag(...). Beachte aber, dass (...)."

Antwortmöglichkeit

<u>antwort_id</u>	<u>parameter_id</u>	antwort_text	aenderung_direkt	aenderung_verzoegert
01	BODEN	"Boden nicht düngen"	0	-10
02	BODEN	"Boden düngen"	20	0
02	WASSER	"Boden düngen"	-5	0



Folgend aus der Anwendungslogik-Modellierung aus der letzten Abgabe und mit dem Ziel das erste PoC umzusetzen, wurde nun eine Datenbank aufgesetzt. Vor der tatsächlichen Programmierung wurde die Datenbank, nach Möglichkeit nach den ersten drei Normalformen modelliert, um eine möglichst hohe Wartbarkeit zu garantieren,

Datenredundanzen zu vermeiden und die Übersichtlichkeit zu behalten. Was zwar konzeptionell in den letzten Audits angesprochen wurde, sich aber noch nicht in der Anwendungslogik wiederfindet ist die Entität Event (nächste Folie). Wir wollten in periodischen Abschnitten (genauer gesagt in Abhängigkeit von den Parametern - näheres dazu auf der Game Loop Folie) mit gewissen Wahrscheinlichkeiten (im Code in AnswerManager.cs genauer erklärt) bestimmte Ereignisse triggern, die zusätzlichen Einfluss auf die Parameter haben.

Datenbankmodellierung

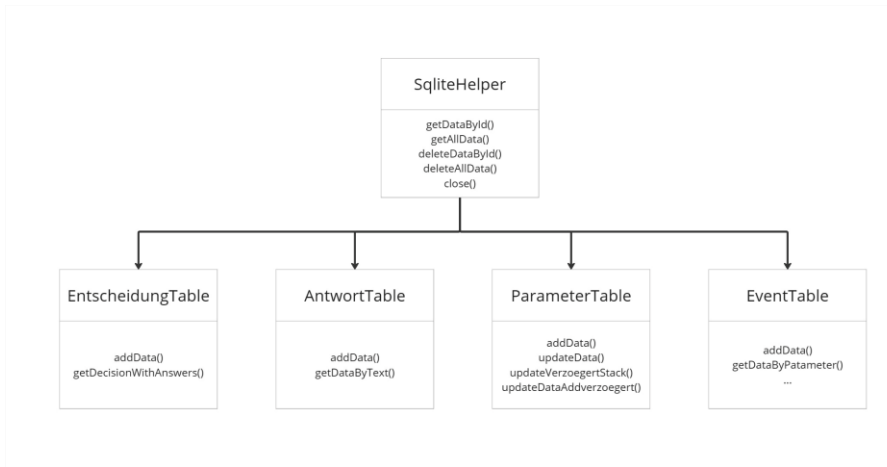
Parameter

<u>parameter_id</u>	parameter_wert	verzoeigert_stack
WASSER	50	0
BODEN	50	0
BIODIV	50	0
GELD	50	0

Event

<u>event_id</u>	event_text	parameter_id	aenderung_direkt
001	"Durch das Verwenden von Pestiziden kam es zu einem lokalen Massensterben der Insekten, die deine Ernte bestäuben. Du erhältst weniger Ertrag."	GELD	-30

(vgl. Notizen der vorherigen Folie)



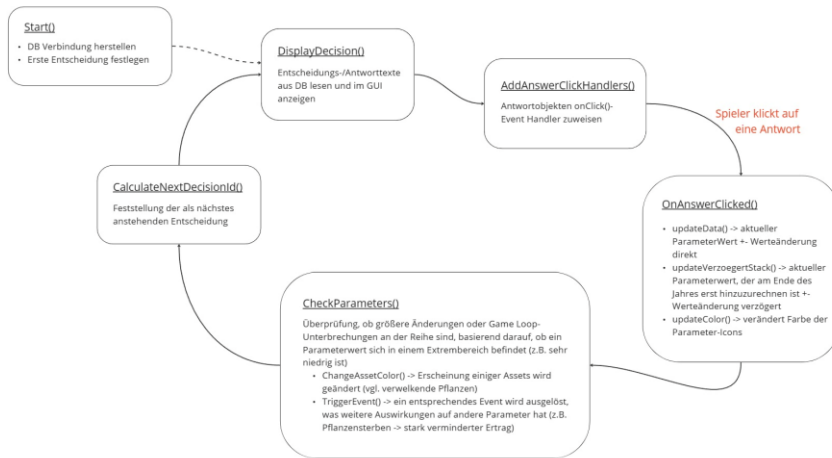
Im Code wurde, um die Vorteile der Polymorphie nutzen zu können, eine Klasse **SqliteHelper** geschrieben, die als Basisklasse für die von ihr abgeleiteten konkreten Tabellen-Klassen dient. Die Basisklasse stellt in ihrem Konstruktor eine Verbindung zur (lokalen) Datenbank **EcoDataBase.sqlite** her und stellt grundlegende CRUD-Funktionen wie

getDataById(), deleteAllData(), etc. Zur Verfügung. Die abgeleiteten Klassen repräsentieren je eine Tabelle aus der Datenbankmodellierung. Sie haben eine spezifizierte Funktion zum Einfügen von Daten, da die Struktur jeder Tabelle unterschiedlich ist und man diese Funktion (bzw. Das Sql INSERT Statement) deswegen nicht verallgemeinern konnte. Sie erben natürlich die grundlegenden Funktionen von der Basisklasse SqlHelper, verfügen aber noch über eigene spezifische Funktionen, die nur für die jeweilige Tabelle relevant ist. So hat die EntscheidungTable-Klasse beispielsweise eine Funktion, die einen Join zwischen der Entscheidungstabelle und der Antworttabelle durchführt und den Text der Entscheidung sowie die Texte der

beiden Antworten, auf die durch Fremdschlüssel in der EntscheidungTable verwiesen wird, ausliest. Im File jeder abgeleiteten Klasse ist außerdem eine Entitätsklasse (z.B. EntscheidungEntity) definiert, welches die Entität als ein Objekt mit den Attributen als Eigenschaften definiert, diese im Konstruktor entgegennimmt und setzt, was das Hinzufügen von Daten in die Tabelle sicherer (durch type checks, etc.) und einfacher macht.

3 der 4 (abgeleiteten) Tabellen im obigen Schema sind zum jetzigen Zeitpunkt vollständig implementiert, sodass im Code mit ihnen gearbeitet werden kann und alle Funktionen zur Verfügung stehen. Die vierte Tabelle "Event" ist

schon modelliert (siehe vorherige Folie)
und auch fest im Code eingeplant, muss
aber noch vollständig implementiert (und
getestet) werden.



Der Game Loop, also die laufende Schleife der verfügbaren Aktionen/Funktionsweisen des Spiels, der auch gecoded wurde (vgl. AnswerManager.cs-Datei im Repo), wurde hier noch einmal schematisch veranschaulicht. Ein Spieldurchlauf startet mit Drücken auf "Play" - zu diesem Zeitpunkt wird die Start()-Funktion der Klasse von Unity aufgerufen. Die

Datenbank-Verbindung wird hier nun geöffnet und ggf. (im Falle des Prototyps) gefüllt (das "Erstellen" von Instanzen erstellt hier keine neuen DB-Tabellen, sondern öffnet nur die Verbindung). Die allererste Entscheidung zu Beginn des Spiels wird hier manuell gesetzt, sie hat die ID "0". Mit dieser Entscheidung wird nun die Funktion `DisplayDecision()` aufgerufen, womit wir uns in den `GameLoop` begeben. Hier wird nun auf Funktionen zugegriffen, die in den entsprechenden Datenbankklassen (siehe vorherige Folien) definiert wurden, die SQL-Querys ausführen und die benötigten Daten aus der Datenbank auslesen – im Fall von `DisplayDecision` sind das die Text-Strings der Entscheidung und der beiden Antworten. Diese String-Werte werden den `TextMeshPro`-Objekten (Unityinterne UI-Textdarstellung)

hinzugefügt und somit im UI angezeigt. Der Nutzer kann die Entscheidung und die beiden Antwortmöglichkeiten also nun einsehen und die gewünschte Antwort auswählen. Um entsprechend auf einen Klick reagieren zu können, wurde den Antwort-TextMeshPro-Objekten eine Button-Komponente hinzugefügt, für die `OnClick()`-Event Handler definierbar sind. Diese sorgen dafür, dass im Falle eines Klicks auf eine Antwort die Funktion `OnAnswerClicked()` mit den antwortspezifischen Argumenten (z.B. auf welche Parameter hat diese Antwort Einfluss und wie viel?) aufgerufen wird. In `OnAnswerClicked()` werden dann zunächst die UI-Elemente, die nach jeder Entscheidung ihr Aussehen verändern (sprich: die Parameter-Icons), geändert.

Außerdem werden die durch die ausgewählte Antwort ausgelösten Parameteränderungen in die DB überschrieben (die Berechnung findet in den DB-Klassen statt). Danach werden mit der Funktion CheckParameters() die aktuellen Werte der einzelnen Parameter geprüft (natürlich nachdem die letzte Änderung gespeichert wurde). Hier wird ausgewertet, ob der aktuelle Wert eines Parameters die Chance hat, das Aussehen der Assets (z.B. der Pflanzen) zu verändern oder ein Event auszulösen. Dieses Event kann ebenfalls Parameterwerte beeinflussen und ist, wenn es eingetroffen ist, nicht mehr abzuwenden. Es kann natürlich auch sein, dass in CheckParameters() gar nichts passiert. Anschließend wird (je nachdem,

in welcher Phase des Landwirtschaftjahres wir uns befinden) die ID der nächsten Entscheidung bestimmt. Diese wird an DisplayDecision() übergeben und der Loop beginnt erneut.

E1: Anbauplan: Getreide vs. Gemüse

Soll die Farm hauptsächlich Getreide oder Gemüse anbauen?

Auswahl: **Getreide**

Pro: Stabiler Markt, konstante Nachfrage

Direkt: -5 **GELD**

Jährlich: +50 **GELD**

Auswahl: **Gemüse**

Pro: Höhere Preise bei Direktverkauf,

Con: Anspruchsvoller in der Pflege, Risiko von Ernteaussfällen

Direkt: -10 **GELD**

Jährlich: +65 **GELD**



Beispiel: 1 Zyklus Entscheidungsbaum Durchgang

E2: Fruchtfolge: Monokultur vs. Diversifizierung

Soll die Farm auf Monokultur setzen oder verschiedene Pflanzen in einer Fruchtfolge anbauen?

Auswahl: **Monokultur**

Pro: Einfachere Pflege und Bewirtschaftung

Con: Bodenermüdung und Verlust von Nährstoffen

Jährlich: **+10 GELD, -5 BODEN**

Auswahl: **Diversifizierung**

Pro: Bessere Bodenqualität, Risikostreuung bei Ernteaussfällen

Con: Komplexere Bewirtschaftung, Mögliche Schwankungen in Marktbedingungen

Direkt: **+2 GELD**

Jährlich: **-1 BODEN**



E3: Düngemittelwahl Chemisch vs. Organisch

Soll die Farm auf chemische oder organische Düngemittel setzen?

Auswahl: **Chemisch**

Pro: Schnellere und präzisere Nährstoffversorgung

Con: Langfristige Bodenqualitätsprobleme, Umweltauswirkungen durch Chemikalien

Direkt: -5 **GELD**

Jährlich: -5 **BODEN**

Auswahl: **Organisch**

Pro: Nachhaltige Bodenverbesserung, Geringere Umweltauswirkungen

Con: Langsamere Freisetzung von Nährstoffen, Höhere Anfangsinvestitionen

Direkt: -7 **GELD**

Jährlich: -1 **BODEN**



Beispiel: 1 Zyklus Entscheidungsbaum Durchgang

Jahresende-Event:

Die Bodenqualität auf der Farm hat sich signifikant verschlechtert, was zu ernsthaften Ernteaussfällen führt. Der Boden ist anfällig für Krankheiten, Nährstoffmangel und geringe Fruchtbarkeit. Die einseitige Bewirtschaftung und Vernachlässigung ökologischer Landbauprinzipien haben zu einer Abnahme der Bodenqualität geführt, die nun den Ertrag beeinträchtigt. (Bearbeitet)



Ziele für Audit 4

- **Weitere Ausarbeitung der Entscheidungen und Auswirkungen**
- **Spiel über 10 Zyklen(Jahre) durchlaufbar**
- **Vollständige Implementationen**
 - Zeitsystem (bestimmte Anzahl Entscheidungen bis Übertritt in nächste Phase);
 - Event Trigger Mechanik (Wahrscheinlichkeiten aus Parametern ableiten; siehe Pseudocode in AnswerManager.cs)
- **Wissenschaftliches Poster**
- **Posterslam vorbereitet**

