

Duale Hochschule Baden-Württemberg
Mannheim

Systemanalyse - Fallstudie

Watch Tycoon 2017

Studiengang Wirtschaftsinformatik

Studienrichtung Software Engineering

Kurs:

WWI 16 SEA

Studiengangsleiter:

Prof. Dr. Julian Reichwald

Dozent:

Hr. Gregor Tielsch

Teammitglieder:

Luisa Karl (LK),
Rebekka Henn (RH),
Miriam Wolf (MW),
Ewald Anselm (EA),
Tillmann Heß (TH),
Erik Schmitt (ES),
Nico Feil (NF)

Bearbeitungszeitraum:

28.08.2017 – 13.11.2017

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Listingsverzeichnis	v
1 Einleitung	1
1.1 Aufbau der Arbeit	1
1.2 Herangehensweise	2
1.3 Grundgedanke des Spiels	2
2 Der Industriezweig Uhr	3
2.1 Geschichtliches zur Uhrindustrie	3
2.2 Marktsituation	3
3 Fachkonzept	4
3.1 Anforderungen	4
3.2 Klassen	5
3.2.1 Spielbrett	5
3.2.2 Unternehmen	7
3.2.3 Uhren	16
3.2.4 Interface iUhrenkategorie	17
3.2.5 Uhrmodell	17
3.2.6 Info	18
4 Spielplan	20
5 Mock-Up/UI	22
5.1 Mock-Up's	22
5.2 Finales UI	27
6 Abteilungen	33
6.1 Produktion	33
6.2 Forschung&Entwicklung	33
6.3 Marketing	34
6.4 Vertrieb	34
6.5 Einkauf	35

7 Markt	37
7.1 Aufbau	37
7.2 Klasse	41
8 Diagramme	42
8.1 UML-Diagramme	42
8.1.1 Idee/Erstes UML-Diagramm	42
8.1.2 Finales UML/Unternehmenssimulation	44
8.2 Zustandsdiagramm	46
9 Architektur	47
9.1 Einflussfaktoren für Architektur	47
9.2 Übersicht über technischen Aufbau	47
9.3 Laufzeitumgebung	48
9.4 Java Servlets	49
9.5 JavaServer Page	49
9.6 User Interface	50
9.7 Eingabeüberprüfung im UI durch JavaScript	52
10 Entwicklungsumgebung	58
10.1 Eclipse JAVA EE IDE	58
10.2 Git	58
10.3 GitHub	58
10.4 LaTeX	59
11 Qualitätssicherung mittels JUnit-Tests	60
12 Projektorganisation	63
12.1 Aufgabenverteilung	63
12.2 Meilensteine	64
13 Verbesserungs-/Erweiterungsmöglichkeiten	65
14 Fazit/Ausblick	67

Abbildungsverzeichnis

Abbildung 3.1	Vereinfachtes Use Case	5
Abbildung 3.2	Klassendiagramm Unternehmen	7
Abbildung 3.3	Klassendiagramm iUhrenkategorie	17
Abbildung 5.1	Mock-Up: Einstiegsbildschirm	22
Abbildung 5.2	Mock-Up: Übersicht der Produkte	23
Abbildung 5.3	Mock-Up: Übersicht F&E	23
Abbildung 5.4	Mock-Up: Übersicht Produktion	24
Abbildung 5.5	Mock-Up: Übersicht Einkauf	24
Abbildung 5.6	Mock-Up: Übersicht Vertrieb	25
Abbildung 5.7	Mock-Up: Übersicht Marketing	25
Abbildung 5.8	Mock-Up: Übersicht Statistik/Markt	26
Abbildung 5.9	Mock-Up: „Nächster Spieler“-Anzeige	26
Abbildung 5.10	UI: Einstiegsbildschirm	27
Abbildung 5.11	UI: Segmentauswahl für Spieler X	27
Abbildung 5.12	UI: Übersicht der Produkte	28
Abbildung 5.13	UI: Bestandsübersicht	28
Abbildung 5.14	UI: Übersicht F&E	29
Abbildung 5.15	UI: Übersicht Produktion	29
Abbildung 5.16	UI: Übersicht Einkauf	30
Abbildung 5.17	UI: Übersicht Vertrieb	30
Abbildung 5.18	UI: Übersicht Marketing	31
Abbildung 5.19	UI: Übersicht Statistik	31
Abbildung 5.20	UI: Übersicht Sieger	32
Abbildung 7.1	Logisches Wachstum	39
Abbildung 7.2	Markt Klasse	41
Abbildung 8.1	Erstes UML	43
Abbildung 8.2	Finales UML von „Watch Tycoon 2017“	45
Abbildung 8.3	Zustandsdiagramm einer Spielrunde	46
Abbildung 9.1	3-Schichten-Modell	48
Abbildung 9.2	Button: do-undo	53

Abbildungsverzeichnis

Abbildung 11.1 Coverage des Szenariotests	62
Abbildung 11.2 Gesamte Coverage	62
Abbildung 11.3 Gesamte Coverage (ohne Servlets)	62
Abbildung 12.1 Meilenstein-Diagramm	64

Listingsverzeichnis

3.1	Gewinnermittlung	6
3.2	Interface iUHrenkategorie	7
3.3	Neue Uhr erforschen	8
3.4	Produktion erweitern	9
3.5	Gewünschte Menge produzieren	10
3.6	Weiteres Segment freischalten	12
3.7	Weiteres Uhrwerk erforschen	12
3.8	Einkauf erweitern	13
3.9	Marketingkampagne starten	14
3.10	Überprüfung vorhandenes Kapital	15
3.11	Selbstkosten/Marktwert berechnen	16
3.12	Weiteres Merkmal entwickeln	18
3.13	Beispielaufruf aus Info.java	19
9.1	Geschriebener Quellcode mit JSP	52
9.2	Kompilierter HTML-Code	52
9.3	Überprüfung Kontostand	52
9.4	OnClick-Event mit Bedingung für Kontostandüberprüfung	52
9.5	Ausgabe bei zu geringem Kapital	53
9.6	Eingabeüberprüfung Produktion	54
9.7	Eingabeüberprüfung Vertrieb	56
11.1	Einfache Überprüfung durch Konsolenausgabe	60
11.2	Test einer Spielererzeugung	60

1 Einleitung

1.1 Aufbau der Arbeit

(NF) In dieser Arbeit wird die Erarbeitung einer Konzeption und Umsetzung eines Unternehmensplanspiels im Rahmen der Fallstudie beschrieben. Das Ziel dieser Arbeit ist die Entwicklung eines computergestützten Planspiels, auf dem die Spieler in Konkurrenz zueinander stehen und ein fiktiver Markt gebildet wird. Der Entwurf und die Entwicklung des Spiels bilden die Hauptaspekte dieser Arbeit.

Die Entwicklung wird in der Programmiersprache Java realisiert. Funktionen wurden zusätzlich in JavaScript geschrieben. Mit einem Tomcat-Webserver wird dann das Spiel lauffähig gemacht.

Betriebswirtschaftliche Aspekte werden in dem Spiel ebenfalls berücksichtigt. Dazu zählen zum Beispiel: variable Nachfrage der einzelnen Produkte, Betriebskosten oder Verkaufspreise. Die Marktveränderungen werden mittels eigen entwickeltem Algorithmus berechnet.

Die Entwicklung des grafischen User Interfaces wurde mit Hilfe von Bootstrap und HTML realisiert. Das Mock-Up, was ebenfalls in HTML statisch erstellt wurde, war dabei die Vorlage.

Mittels JUnit-Test werden die Tests durchgeführt und die Code-Coverage überprüft. Ziel hierbei ist eine Coverage > 65%.

Des Weiteren wird die Projektorganisation vorgestellt, welche während dem Projekt eine wichtig Rolle gespielt hat. Hierbei wurde zusätzlich ein Meilenstein-Diagramm in 12.1 auf Seite 64 verwendet um die einzelnen Meilensteine optisch darzustellen.

Zuletzt werden theoretische Verbesserungsmöglichkeiten aufgezeigt und ein Fazit über das gesamte Projekt gezogen.

1.2 Herangehensweise

(NF) Das Team setzt sich aus sieben Mitgliedern zusammen. Zu Beginn wurde in einem Kick-Off Meeting das Team bzw. die Industrie ausgewählt, aus welcher das Planspiel bestehen soll. Nach kurzer Überlegung konnte schnell in der Uhrenindustrie ein Nenner gefunden werden.

Auch bei der recht großen Anzahl an Teammitglieder konnte die Aufgaben sinnvoll aufgeteilt werden. So herrschte eine recht ausgewogene Balance, was die verschiedenen Schwerpunkte und Erfahrungen der Mitglieder betraf. Im Laufe des Projekts zeigte sich auch hierbei der Vorteil eines breiten technischen Portfolios. Dennoch konnten einige Problemstellungen nicht mit dem reinen Wissen aus dem Studium abgedeckt werden. Ebenso waren Kenntnisse in der Mathematik und Wirtschaftswissenschaft gefragt um beispielsweise einen fiktiven Markt mit einem realistischen Algorithmus zu versehen.

1.3 Grundgedanke des Spiels

(NF) Das Spiel „Watch Tycoon 2017“ soll in einem rundenbasierten computergestützten Planspiel mit bis zu vier Spielern eine nahezu reale Unternehmenssimulation in einem Industriezweig darstellen. Der hierfür verwendet Industriezweig ist die Uhrenindustrie.

Der Spieler hat die Abteilungen Produktion, F&E, Einkauf, Verkauf und Marketing zur Verfügung um eine bestmögliche Strategie auszuarbeiten und ein breites Spektrum an Varianz zu erhalten.

Der fiktive Markt soll mittels einem mathematischen Algorithmus so real wie möglich erstellt werden und den Mittelpunkt für Angebot und Nachfrage bilden.

2 Der Industriezweig Uhr

2.1 Geschichtliches zur Uhrindustrie

(LK) Die Schweiz ist derzeit das Land mit den meisten Uhrenexporten. Weltweit folgt der Schweiz Hongkong und China.

In den 1970er und 1980er Jahren erhielt die klassische schweizer Uhrenindustrie Konkurrenz durch die Entstehung der elektrischen Uhren und den asiatischen Markt. Nachdem sie sich bis 2015 wieder etwas erholen konnte und die Exporte von 4,3 Milliarden Franken im Jahr 1986 auf 21,5 Milliarden im Jahr 2015.

Mittlerweile schaffen es vor allem die Großmarken, sich gegen die neue Konkurrenz der Smart Watches durchzusetzen, indem sie entweder hochwertige Luxusuhren herstellen, die als Schmuckstück und Statussymbol dienen oder indem sie selbst Smart Watches entwickeln.

2.2 Marktsituation

(NF) Der reale Uhrenmarkt wird (im Gegensatz zu unserem fiktiven Uhrenmarkt) von nur wenigen Ländern beherrscht. Hong Kong, China und die Schweiz sind dabei an der Spitze. Hong Kong und China sind, wie in vielen anderen Branchen, mit Massenproduktionen die weltweit größten Uhrenproduzenten. Die Uhren sind im Niedrigpreissegment angesiedelt. Hingegen ist die Schweiz mit ihren berühmten „Schweizer Uhrwerken“ Weltmarktführer im hochwertigen Marktbereich. Aufgrund der hochpreisigen Uhren macht die Schweiz nur einen verschwindend geringen Anteil an der globalen Produktion aus, hingegen sie wertmäßig mit Abstand das führende Exportland ist.

Die Firmen, die symbolisch für die Weltmarktführerschaft stehen, sind Swatch, Richemont und Rolex.

Zusammen machen die drei Marken 40% bis 50% des weltweiten Uhrenumsatzes aus. Im Jahr 2016 exportierte die Schweiz Uhren im Wert von 19,8 Milliarden EURO. Gefolgt von Hong Kong (8,8) und China (5,3).

3 Fachkonzept

3.1 Anforderungen

(MW) Die Abbildung 3.1 zeigt ein vereinfachtes Use Case Diagramm für den Spieler. Diese Anforderungen müssen in der Benutzeroberfläche und im tatsächlichen Fachkonzept umgesetzt werden.

Um ein neues Spiel zu spielen, muss der Spieler ein Spiel starten können. Dies beinhaltet, dass ein Spielbrett erzeugt, ein Segment gewählt werden kann und in diesem Segment eine Standarduhr mit den ersten Attributen freigeschaltet wird. Je nach Segment werden verschiedene Kosten vom Kapital abgezogen um somit das Verhältnis von Billig zu Teuer zu gewährleisten.

Während dem Spiel kann der Spieler verschiedenes in seinem Unternehmen machen, wie zum Beispiel die Produktion erweitern, Attribute entwickeln, Uhren produzieren und verkaufen. Weitere Aufgaben können aus dem Use Case abgelesen werden. Sobald ein Spieler seine Uhr verbessert / produziert oder die Absatzmenge angegeben hat, wird die Runde beendet. In diesem Schritt werden die Spielerdaten gespeichert und somit auch die neue Konfiguration der Uhr berechnet. Anschließend ist der nächste Spieler an der Reihe. Sollte eine Runde von allen Spielern beendet worden sein ist die erste Periode zu Ende und die Marktsimulation wird gestartet.

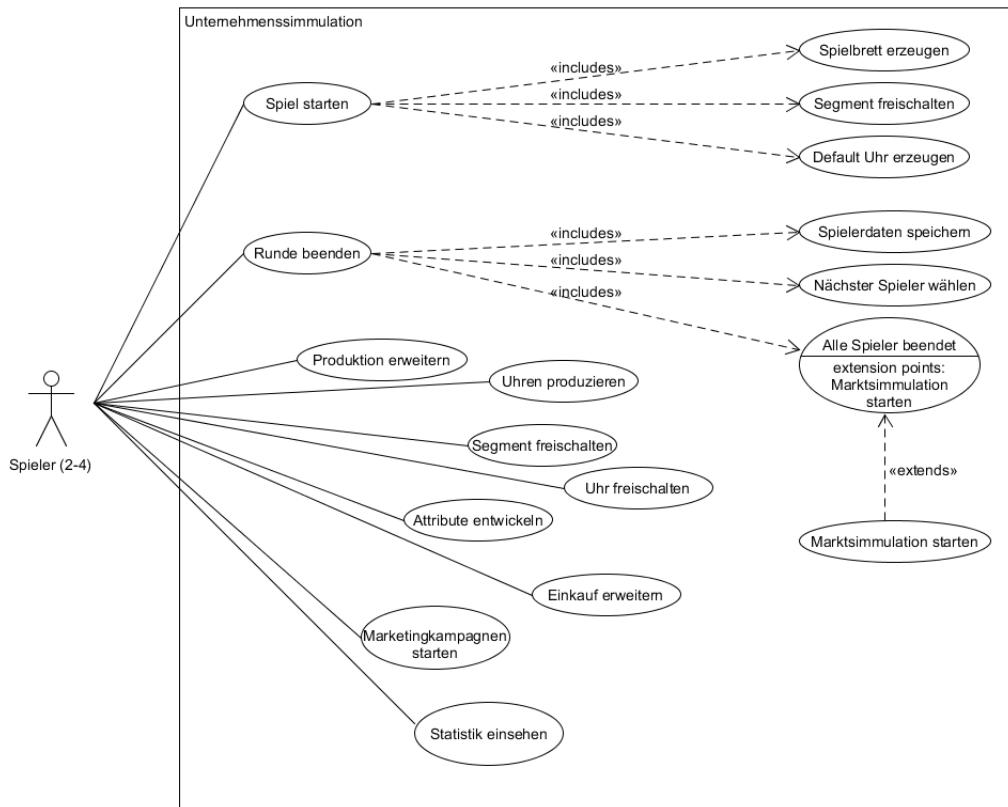


Abbildung 3.1: Vereinfachtes Use Case

3.2 Klassen

(MW) Im folgenden Abschnitt werden die einzelnen Klassen mit den wichtigsten Methoden kurz erläutert. Ein komplettes Klassendiagramm ist in Abbildung 8.2 dargestellt. Der Markt wird im Kapitel 7 genauer beschrieben.

3.2.1 Spielbrett

(MW) Das Spielbrett managed den kompletten Spielablauf. Im ersten Schritt wird ein Spielbrett mit der Anzahl der Runden, dem Marktvolumen und dem Einflussbereich für den Markt erzeugt. Anschließend kann der erste Spieler sein Unternehmen leiten. Sobald er die Runde beendet, wird der nächste Spieler ausgewählt bis alle Spieler die Periode abgeschlossen haben. Mit abschließen der Periode wird die Martsimulation gestartet. Genaueres dazu gibt es im Kapitel 7.

Nachdem alle Runden gespielt und die Marktsimulation am Ende noch einmal durchgelaufen ist, wird der Sieger ermittelt. Die Methode der Gewinnermittlung ist im folgenden aufgezeigt:

```

1  private void gewinnermittlung() {
2      this.sieger = new int[spieler.length];
3      double kapi[] = new double[spieler.length];
4
5      // Kapital in extra array speichern
6      for(int i = 0; i < spieler.length; i++)
7          kapi[i] = spieler[i].getKapital();
8
9      // Kapitel nach der groesse sortieren und umdrehen
10     Arrays.sort(kapi);
11     double temp[] = new double[kapi.length];
12     int t = kapi.length-1;
13     for(int i = 0; i < kapi.length; i++)
14     {
15         temp[t] = kapi[i];
16         t--;
17     }
18     kapi = temp;
19
20     // Kapital den Siegern zuordnen und in siegerarray speichern
21     for(int i = 0; i < spieler.length; i++) {
22         for(int j = 0; j < kapi.length; j++)
23             if(kapi[i] == spieler[j].getKapital()) {
24                 sieger[i] = j;
25                 break;
26             }
27     }
28 }
```

Listing 3.1: Gewinnermittlung

Nachdem der Gewinner ermittelt und auf der Benutzeroberfläche angezeigt wurde, kann ein neues Spiel gestartet werden und die Simulation startet von vorn.

3.2.2 Unternehmen

(MW) Abbildung 3.2 zeigt zunächst ein Klassendiagramm, um einen groben Überblick zu erhalten. Üblicherweise sind die Attribute und Methoden in einem Klassendiagramm untereinander, jedoch wurde es hier aus Platzgründen nebeneinander dargestellt.

Unternehmen	
<pre> - name : String - kapital : double - kapitalAlt : double - verkauftUhrenRunden : int[10] - produktioslimitBillig : int - produktioslimitOeko : int - produktioslimitPremium : int - produktionskostenBillig : int - produktionskostenOeko : int - produktionskostenPremium : int - uhr : iUhrenkategorie[3] - freieSegmenteAllgemein : boolean[3] - freigeschalteneAttrBillig : boolean[3][3] - freigeschalteneAttrOeko : boolean[3][3] - freigeschalteneAttrPremium : boolean[3][3] - prodKostenSenkungStraßeBillig : boolean[3] - prodKostenSenkungStraßeOeko : boolean[3] - prodKostenSenkungStraßePremium : boolean[3] - kapaErwStraßeBillig : boolean[3] - kapaErwStraßeOeko : boolean[3] - kapaErwStraßePremium : boolean[3] - verbesserungEinkaufBillig : boolean[3] - verbesserungEinkaufOeko : boolean[3] - verbesserungEinkaufPremium : boolean[3]</pre>	<pre> + Unternehmen(name : String) + erforscheUhr(segment : String) : boolean + erforscheUhrwerk(segment : String, uhr : int) : void + erforscheGehaeuse(segment : String, uhr : int) : void + erforscheArmband(segment : String, uhr : int) : void + erweiterteProduktion(segment : String) : boolean + senkeProduktionskosten(segment: String) : boolean + erweiterteEinkauf(segment: String) : boolean + uhrenMarketing(uhr : int, kampagne : boolean[3]) : void + unternehmenMarketing(kampagne : boolean[3]) : void + produzieren (menge : int, uhr : int) : void + bieteUhren(menge : int, uhr : int, preis: double) : void + freischaltenSegment(segment : String) : void - zufallMarketing() : boolean - berechneMarketingkosten(kampagne : boolean[3], art : - testeMengeProduzieren(menge : int, uhr : int; limit : int) - erhoeheProduktionslimit(segment : String, stufe : int) : v - senkeProduktionskosten(segment : String, stufe : int) : v - isFreigeschaltenSegment(segment : String) : boolean - indexFreieUhr() : int - checkeKapital(kosten : double) : boolean</pre> <p style="text-align: center;">Getter & Setter</p>

Abbildung 3.2: Klassendiagramm Unternehmen

Im Unternehmen sind alle wichtigen Attribute gespeichert, die zu einem Unternehmen / Spieler gehören. Da der Spieler das Unternehmen leitet, wird im folgenden das Unternehmen benannt.

Das Interface iUhrenkategorie bietet die Methodensignaturen für die verschiedenen Uhren an und wird somit als Schnittstelle verwendet. Damit das Unternehmen alle Uhren in einem Array vereinen kann, wird Polymorphie angewandt. Im folgenden Codeabschnitt wird gezeigt, wie Polymorphie in Java verwendet wird.

```

1  private iUhrenkategorie uhr [] = new iUhrenkategorie [3];
2
3  uhr [0] = new BilligUhr();
```

Listing 3.2: Interface iUhrenkategorie

Das Attribut *uhr* wird als iUhrenkategorie-Array mit der Größe 3 erzeugt. Anschließend wird im ersten Feld eine BilligUhr erzeugt. Durch Polymorphy wird eine Referenzvariable vom Typ iUhrenkategorie erstellt. Somit können die verschiedenen Uhrenkategorien in einem Array erstellt und im Anschluss verwendet werden.

Im folgenden Abschnitt werden die wichtigsten Methoden der Klasse erläutert. Viele der Methoden sind durch ein Switch-Case in die Segmente unterteilt. Da sich der Codeteil nur um die Attributnamen verändert, wurden diese Codestücke durch [...] ersetzt.

Uhr erforschen

(MW) Das erforschen der Uhr ist einer der wichtigsten Methoden im Unternehmen. Sobald das Spiel gestartet und der Spieler ein Segment gewählt hat, wird eine Standarduhr in dem Segment erforscht. Alle weiteren Uhren können während des Spiels erforscht werden, sofern genügend Kapital vorhanden ist.

```

1 public boolean erforscheUhr(String segment) {
2     boolean result = false;
3     // Test auf naechste Freie Uhr
4     int index = indexFreieUhr();
5
6     // Wenn alle Uhren entwickelt oder nicht genug Kohle-> false
7     // zurueckgeben;
8     if(index != -1 ) {
9         // Checken ob Segment bereits freigeschalten
10        if(isFreigeschaltenSegment(segment)) {
11            switch(segment) {
12                case "Billig":
13                    if(checkeKapital(Info.
14                        getKostenUhrBillig())) {
15                        this.uhr[index] = new
16                            BilligUhr();
17                        result = true;
18                    }
19                else
20                    System.out.println("Nicht genug Kohle!
21                                ");
22                break;
23                case "Premium":
24                    [...]
25                case "Oeko":
26                    [...]
27                default:
28                    System.out.println("Falsches
29                                Segment");
30            }
31        }
32    }
33 }
```

```

25                                break;
26                            }
27                        }
28                    else {
29                        System.out.println("Segment noch nicht
30                                     freigeschalten");
31                    }
32                else {
33                    System.out.println("Es kann keine weitere Uhr
34                                     erforscht werden!");
35                }
36            return result;
}

```

Listing 3.3: Neue Uhr erforschen

Zunächst wird überprüft, ob noch eine weitere Uhr erforscht werden kann. Jedes Unternehmen kann maximal 3 Uhren entwickeln bzw. erforschen. Als nächstes wird geschaut, ob das passende Segment schon erforscht wurde, in dem die Uhr erstellt werden soll. Als letzte Abfrage wird über ein Switch-Case das Segment gewählt und die passende Uhr am passenden Indexplatz erzeugt.

Produktion erweitern

```

1 public boolean erweitereProduktion(String segment) {
2     // Erweitert die Kapazität -> erhöht also das
3     // Produktionslimit
4     switch(segment) {
5         case "Billig":
6             for(int i = 0; i < 3; i++) {
7                 if(this.getKapaErwStrasseBillig()[i]
8                     == false) {
9                     if(checkeKapital(Info.
10                         getKostenProduktionBillig
11                         ()[i])) {
12                         kapaErwStrasseBillig[i] =
13                             true;
14                         erhöheProduktionslimit(
15                             segment, i);
16                         return true;
17                 }
18             }
19         }
20         break;
21         case "Premium":
22             [...]
23     }
24 }

```

```

17         }
18     break;
19     case "Oeko":
20         [...]
21     }
22     break;
23     default:
24         System.out.println("Falsches Segment");
25         break;
26     }
27     return false;
28 }
```

Listing 3.4: Produktion erweitern

(MW) Diese Methode wählt zunächst über ein Switch-Case das Segment aus, in dem das Produktionslimit erhöht werden soll. Anschließend wird überprüft, ob genügend Kapital vorhanden ist um die Produktion zu erweitern. Diese Kosten werden aus der Klasse *Info.java* ausgelesen. Wenn genug Kapital vorhanden ist, werden die Kosten abgezogen und das passende Feld im Array wird freigeschaltet.

Uhren produzieren

(MW) Diese Methode ist auch ein wichtiger Bestandteil des Unternehmens. Hier werden die Uhren produziert. Als Übergabeparameter werden die Menge und die Uhr übergeben.

Zunächst wird überprüft, ob diese Uhr überhaupt existiert und somit wird ein falscher Aufruf abgefangen. Als nächstes werden die Selbstkosten der Uhr berechnet und gegebenenfalls mit einem Faktor Einkauf verrechnet. Der Faktor Einkauf wird dann gesetzt, wenn eine Erweiterung im Einkauf stattgefunden hat. Solange dieser nicht erweitert wurde, bleibt der Faktor 0.

Als nächstes wird mit der privaten Methode *testeMengeProduzieren* getestet, ob die gewünschte Menge produziert werden kann. Sollte nicht genügend Kapital vorhanden sein, wird die Menge berechnet, die mit dem letzten Kapital noch produziert werden kann. Im letzten Schritt wird der neue Bestand gesetzt und das Kapital um die Kosten verringert.

```

1 public void produzieren(int menge, int uhr) {
2     if(this.uhr[uhr] != null) {
3         int m = 0;
4         double s = this.uhr[uhr].berechneSelbstkosten();
5         double f = 0;
6         // Segment abfragen
7         switch(this.uhr[uhr].getSegment()) {
8             case "Billig":
```

```

9         f = sucheEinkaufsfaktor("Billig");
10        if(f != 0)
11            s *= f;
12        m = testeMengeProduzieren( menge, uhr
13                                , this.getProduktionslimitBillig
14                                () , s);
15        if(m != -1) {
16            this.setKapital( this.
17                getKapital() - (m * s));
18            this.uhr[uhr].setBestand(this
19                .getBestandUhr(uhr) + m);
20        }
21        break;
22    case "Oeko":
23        ...
24    case "Premium":
25        ...
26}
27}
28
29private int testeMengeProduzieren(int menge, int uhr, int limit,
30    double prodKostenStueck) {
31    int m = -1;
32    // Produktionslimit testen
33    if( (menge + this.getBestandUhr(uhr)) > limit)
34        menge = limit;
35    // Berechnen wie viele mit vorhandenem Kapital
36    // produziert werden koennen
37    for(int i = menge; i > 0; i --) {
38        double prodKosten = i * prodKostenStueck;
39        if(prodKosten <= this.getKapital()) {
40            m = i;
41            break;
42        }
43    }
44    return m;
45}

```

Listing 3.5: Gewünschte Menge produzieren

Segment freischalten

(MW) Diese Methode schaltet ein neues Segment frei. Als Übergabeparameter wird das Segment mitgegeben, welches freigeschaltet werden soll. Wenn genügend Kapital vorhanden ist, wird das Segment freigeschaltet.

```

1  public void freischaltenSegment(String segment) {
2      switch(segment) {
3          case "Billig":
4              if(checkeKapital(Info.getKostenSegmentBillig())) {
5                  if(this.freieSegmenteAllgemein[0] == false)
6                      this.freieSegmenteAllgemein[0] = true
7                      ;
8              }
9              break;
10             case "Oeko":
11                 [...]
12             break;
13             case "Premium":
14                 [...]
15             break;
16     }

```

Listing 3.6: Weiteres Segment freischalten

Attribute entwickeln

(MW) In diesem Abschnitt werden die Attribute Uhrwerk, Gehäuse und Armband entwickelt. Da die Methoden ähnlich sind, wird hier als Codebeispiel nur das Uhrwerk angegeben. Der Übergabeparameter ist das Segment und der Index, welche Erweiterung freigeschaltet werden soll. Nachdem das Kapital ausreichend ist, wird das Attribut entwickelt. Das Entwickeln wird in der Klasse Uhrmodell ausgelagert, da dies für alle Uhren identisch ist und somit kein Wiederholender Code geschrieben wird.

```

1  public void erforscheUhrwerk(String segment, int index) {
2      double kosten = 0;
3      switch (segment) {
4          case "Billig":
5              kosten = Info.getKostenUhrwerkBillig()[index
6                      ];
7              if(checkeKapital(kosten))
8                  freigeschalteneAttrBillig = Uhrmodell.
9                      entwickleUhrwerk(freigeschalteneAttrBillig
10                         , 2);
11             break;
12             case "Oeko":
13                 [...]
14             break;
15             case "Premium":
16                 [...]
17             break;
18     }

```

16 || }

Listing 3.7: Weiteres Uhrwerk erforschen

Einkauf erweitern

(MW) In dieser Methode wird der Einkauf erweitert und somit die Anschaffungskosten der Uhr geringer. Sollte genügend Kapital vorhanden sein, so wird ein Flag gesetzt, welches in der Methode *produzieren* abgefragt und mit den Selbstkosten verrechnet wird. Als Übergabeparameter wird das Segment übergeben.

```

1 public boolean erweitereEinkauf(String segment) {
2     switch(segment) {
3         case "Billig":
4             for(int i = 0; i < 3; i++) {
5                 if(verbesserungEinkaufBillig[i] ==
6                     false) {
7                     if(checkeKapital(Info.
8                         getKostenEinkaufBillig() [i
9                         ])) {
10                        verbesserungEinkaufBillig
11                           [i] = true;
12                           return true;
13                         }
14                         }
15                         break;
16                         case "Premium":
17                             [...]
18                             break;
19                             case "Oeko":
20                                 for(int i = 0; i < 3; i++) {
21                                     [...]
22                                     break;
23                                     default:
24                                         System.out.println("Falsches Segment");
25                                         break;
26                         }
27                         return false;
28     }
29 }
```

Listing 3.8: Einkauf erweitern

Marketingkampagnen starten

(MW) Als Marketingkampagnen sind zwei verschiedene Arten vorgesehen. Als erstes eine Kampagne speziell für die Uhr und ein Unternehmensmarketing. Das spezielle Marketing der Uhr verändert nur den Score einer Uhr, bei der eine Kampagne gestartet werden soll. Im Unternehmensmarketing hingegen wirkt sich die Kampagne auf alle Uhren des Unternehmens aus.

Bei beiden Varianten wird eine Zufallskomponente eingebaut, die über den Erfolg bzw. Misserfolg einer Marketingkampagne entscheidet. Diese Zufallskomponente erstellt eine zufällige Zahl zwischen 0 und 1. Sollte der Wert zwischen 0,4 und 0,6 liegen, scheitert die Marketingkampagne.

```

1  public void uhrenMarketing(int uhr, boolean[] kampagne) {
2      // Kosten der Kampagnen berechnen
3      double kostenMarketing = berechneMarketingkosten(kampagne, "Uhr");
4
5      // Suche den Index
6      int boostIndex = -1;
7      for(int i = 0; i < 3; i++) {
8          if(kampagne[i] == true)
9              boostIndex = i;
10     }
11
12     // Nur wenn Kapital ausreichend ist
13     if(checkeKapital(kostenMarketing)){
14         // Nur Boosten, wenn Zufallszahl nicht zugeschlagen
15         hat
16         if(boostIndex != -1 && zufallMarketing()){
17             this.uhr[uhr].setMarketingboost(this.uhr[uhr]
18                 .getMarketingboost() + Info.
19                 getScoreMarketingkampagne()[boostIndex]);
20         } else {
21             this.uhr[uhr].setMarketingboost(this.uhr[uhr]
22                 .getMarketingboost());
23         }
24     }
25
26     public void unternehmenMarketing(boolean[] kampagne) {
27         // Kosten der Kampagnen berechnen
28         double kostenMarketing = berechneMarketingkosten(kampagne, "Unternehmen");
29         // Suche den Index
30         int boostIndex = -1;
31         for(int i = 0; i < 3; i++) {
32             if(kampagne[i] == true)
```

```

31         boostIndex = i;
32     }
33
34     // Nur wenn Kapital ausreichend ist
35     if(checkeKapital(kostenMarketing)) {
36         if(boostIndex != -1 && zufallMarketing()) {
37             // Marketingboost auf alle Uhren anrechnen wenn
38             // Zufallszahl nicht zugeschlagen hat
39             for(int i = 0; i < 3; i++) {
40                 if(this.uhr[i] != null)
41                     this.uhr[i].setMarketingboost(this.uhr[i].
42                         getMarketingboost() + Info.
43                         getScoreMarketingkampagne()[boostIndex]);
44             }
45         } else {
46             for(int i = 0; i < 3; i++) {
47                 if(this.uhr[i] != null)
48                     this.uhr[i].setMarketingboost(this.
49                         uhr[i].getMarketingboost());
50             }
51         }
52     }
53
54     // Zufall einbauen
55     private boolean zufallMarketing() {
56         // Zufallszahl zwischen 0 und 1
57         Random rand = new Random();
58         double z = rand.nextDouble();
59         // zwischen 0.4 und 0.6 (incl)
60         if(z >= 0.4 && z <= 0.6)
61             return false;
62         return true;
63     }

```

Listing 3.9: Marketingkampagne starten

Kapital überprüfen

(MW) Diese Methode ist zwar klein, aber fein. Sie bestimmt bei fast allen Methoden ob das Kapital ausreichend für die benötigten Kosten sind. Ohne diese Methode könnte das Unternehmen alles entwickeln und hätte sozusagen unendliches Geld zur Verfügung. Sollte genügend Kapital vorhanden sein, dann werden die Kosten vom Kapital abgezogen und es wird *true* zurückgegeben.

```

1 || private boolean checkeKapital(double kosten) {
2 ||     if( (this.kapital - kosten) >= 0) {

```

```

3         this.kapital -= kosten;
4         return true;
5     }
6     else
7     {
8     }

```

Listing 3.10: Überprüfung vorhandenes Kapital

3.2.3 Uhren

(MW) Darunter zählen die Klassen BilligUhr, OekoUhr und PremiumUhr. Alle Klassen implementieren das Interface iUhrenkategorie und müssen alle Methoden des Interfaces implementieren. Das Unternehmen hat Zugriff auf alle implementierten Methoden, die im Interface erstellt wurden, durch die Polymorphie.

Die Uhren-Klassen dienen nur als Datenverwaltung, damit die Konfiguration der Uhren im Unternehmen gespeichert werden können. Die wichtigsten Methoden sind:

```

1 public double berechneSelbstkosten() {
2     return( Info.getSelbstkostenArmbandBillig() [this.getArmband()]
3            ] + Info.getSelbstkostenGehaeuseBillig() [this.getGehaeuse()
4            ()]
5            + Info.getSelbstkostenUhrwerkBillig() [this.getUhrwerk()] );
6
7 public double berechneMarktwert() {
8     this.setMarktwert( this.getSelbstkosten() * (1 + ( Info.
9         getScoreArmbandBillig() [this.getArmband()] + Info.
10        getScoreGehaeuseBillig() [this.getGehaeuse()] +
11        Info.getScoreUhrwerkBillig() [this.getUhrwerk()])));
12
13    return this.getMarktwert();
14}

```

Listing 3.11: Selbstkosten/Marktwert berechnen

Beide Methoden sind mit `@Override` versehen. Dies zeigt, dass die Methode vom Interface überschrieben werden müssen, also ein Methodenbody eingefügt werden muss. Um die Selbstkosten zu berechnen, wird die Konfiguration der Uhr multipliziert mit den Kosten, die für diese Konfiguration stehen. Diese Kosten werden aus der Klasse `Info.java` ausgelesen.

Der Marktwert der Uhren wird berechnet aus den Selbstkosten multipliziert mit (1 + Score der Attribute). Je hochwertiger die Konfiguration der Uhr ist, desto höher ist der Score und somit steigt auch der Marktwert der Uhr an.

3.2.4 Interface iUhrenkategorie

(MW) Das Interface stellt alle Methoden für die Uhren zur Verfügung. An dieser Stelle nur das Klassendiagramm des Interfaces in Abbildung 3.3. Die wichtigsten Methoden werden im Abschnitt Unternehmen erläutert.

«interface» iUhrenkategorie
<pre>+ verkaufen() : void + getAbnahmepotential() : int + getAbnahmequote() : double + setSpielerDaten(armband: int, gehaeuse : int, uhrwerk: int) : void + getUhrwerk() : int + getArmband() : int + getGehaeuse() : int + getSegment() : String + setAngeboteMenge(menge : int) : void + setAbgenommeneMenge(menge : int) : void + getAbgenommeneMenge() : int + setBestand(bestand : int) : void + getBestand() : int + getAngebotspreis() : double + setAngebotspreis(preis : double) : void + getMarketingboost() : double + setMarketingboost(boost : double) : void + setMarktwert(wert : double) : void + getMarktwert() : double + berechneSelbstkosten() : double + getSelbstkosten() : double + setSelbstkosten(kosten : double) : void + berechneMarktwert() : double</pre>

Abbildung 3.3: Klassendiagramm iUhrenkategorie

3.2.5 Uhrmodell

(MW) Die Klasse Uhrmodell wird als statische Klasse implementiert und dient dazu, gleiche Methoden der Uhren auszulagern. Die Klasse besteht aus drei Methoden:

```

1  public static boolean[][] entwickleUhrwerk(boolean[][] uhrwerk, int
2      index) {
3          for(int i = 0; i < 3; i++) {
4              if(uhrwerk[index][i] == false) {
5                  uhrwerk[index][i] = true;
6                  break;
7              }
8          }
9      return uhrwerk;
10 }
11 public static boolean[][] entwickleArmband(boolean[][] armband, int
12     index) {
13     for(int i = 0; i < 3; i++) {
14         if(armband[index][i] == false) {
15             armband[index][i] = true;
16             break;
17         }
18     }
19     return armband;
20 }
21 public static boolean[][] entwickleGehaeuse(boolean[][] gehaeuse, int
22     index) {
23     for(int i = 0; i < 3; i++) {
24         if(gehaeuse[index][i] == false) {
25             gehaeuse[index][i] = true;
26             break;
27         }
28     }
29 }

```

Listing 3.12: Weiteres Merkmal entwickeln

Diese statischen Methoden erweitern das Uhrwerk, das Armband und das Gehäuse der Uhr. Als Übergabeparameter wird das gesamte Array und der feste Index übergeben. Anschließend wird das nächste Attribut freigeschaltet und das erweiterte Array zurückgegeben. Durch die statischen Methoden muss von dieser Klasse keine Instanz erstellt werden und die Methodenaufrufe erfolgen über den Klassennamen.

3.2.6 Info

(MW) Die Info.java dient ausschließlich der Informationsverwaltung. In dieser statischen Klasse werden die Attributnamen, die Kosten und die verschiedenen Faktoren bereitgestellt. Da dies eine statische Klasse ist, wird keine Instanz verwendet, sondern

die Methoden können über den Klassennamen aufgerufen werden.

Beispiel:

```
1 ||     double kosten = Info.getSelbstkostenGaeuseBillig() [0];
```

Listing 3.13: Beispielaufruf aus Info.java

4 Spielplan

(NF) Spielplan "Watch Tycoon"

Anzahl der Spieler: 2-4

Spielverfahren: Rundenbasiertes Strategiespiel mit Hot-Seat Verfahren

Maximal Anzahl der Spielrunden (insgesamt): 10 Runden

Jeder Spieler hat zu Beginn ein Grundkapital von 1.000.000 EURO. Zu Beginn des Spiels muss eine Uhr aus den Segmenten Masse, Öko oder Luxus ausgewählt werden. Je nach Auswahl stehen verschiedene Ausstattungsmerkmale zur Verfügung zudem wird bei der Auswahl des jeweiligen Segments ein unterschiedlicher Betrag zu Beginn abgezogen. Wenn alle Spieler an der Reihe waren, werden die gemachten Änderungen auf den fiktiven Markt übertragen.

Einkauf:

Im Bereich Einkauf können Verbesserungen für die jeweiligen Merkmale gekauft werden. Dadurch werden Kosten gesparte und es entsteht eine Art Rabatt für das jeweilige Merkmal.

Vertrieb:

Hier lassen sich der Verkaufspreis und die geplante Absatzmenge für die maximal drei Uhren eingeben.

Marketing:

Im Marketing dreht sich alles um das Bekanntmachen einer neuen Uhr und des Unternehmens. Hierfür stehen jeweils drei Werbemöglichkeiten zur Verfügung. Aber Achtung: Eine Werbekampagne muss sich nicht immer gut auswirken!

Produktion:

In der Produktion können durch Zukäufe von Produktionsstraßen (maximal zwei weitere Produktionsstraßen möglich) entweder weitere Segmente eröffnet werden oder eine Kostensenkung durch ein bereits vorhandenes Segment erzielt werden. Des Weiteren können für die vorhandenen Produktionsstraßen Erweiterungen gekauft werden (maximal drei weitere möglich) um ebenfalls die Kosten zu senken.

Forschung&Entwicklung:

Die F&E (Forschung&Entwicklung) ist für die Erforschung von besseren Ausstattungsmerkmalen der jeweiligen Segmente zuständig. Durch das Erforschen der zusätzlichen Merkmale wird die Uhr hochwertiger und kann für mehr Geld verkauft werden.

Die Unternehmensabteilungen Marketing, Verkauf, Einkauf, Produktion und F&E sind ab Runde 1 verfügbar.

Der Markt dient als Informationsinstrument für jeden Spieler. Dabei kann der Spieler die Nachfrage und das Marktvolumen einsehen und gegen Geld einmalig den Markt analysieren lassen. Sofern sich ein Spieler für die Produktion der Öko-Uhr entscheidet, erhält dieser zusätzlich einen Bonus, der später in der Endwertung mit einfließt.

Ziel des Spiels/Gewinnbedingung:

Ziel des Spiels ist es nach Ablauf der 10 Spielrunden den größten Gewinn erzielt zu haben.

5 Mock-Up/UI

(NF) Das Mock-Up für die Präsentation der 1. Iteration wie auch das finale UI wurden auf Basis von HTML und Bootstrap erstellt. Das finale UI ähnelt sehr dem Mock-Up, da innerhalb der Gruppe schnell ein gemeinsames Layout gefunden und auf diesem aufgebaut wurde. Die nachfolgenden Images zeigen die Veränderungen vom Mock-Up bis zum finalen UI. In 5.2 wird zudem kurz auf die einzelnen Anzeige-Inhalte eingegangen.

5.1 Mock-Up's

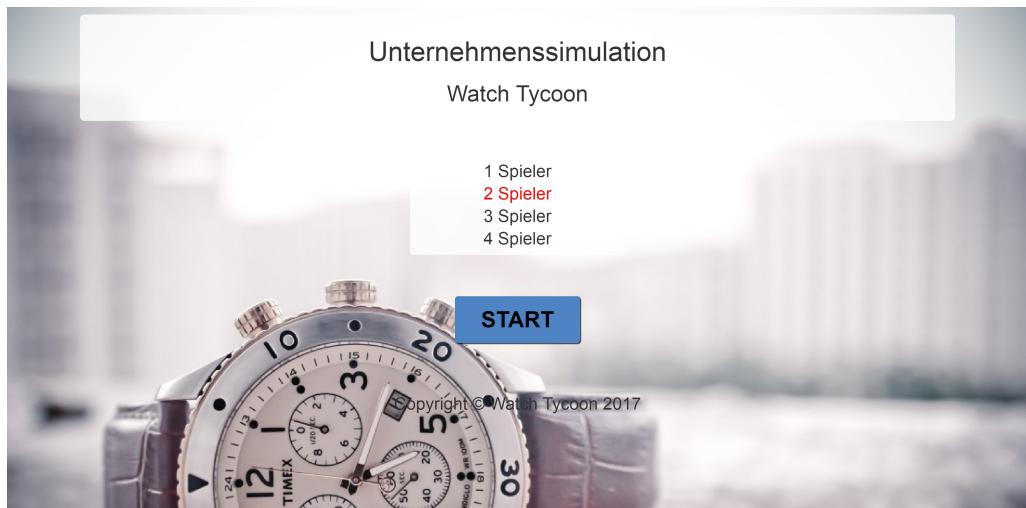


Abbildung 5.1: Mock-Up: Einstiegsbildschirm

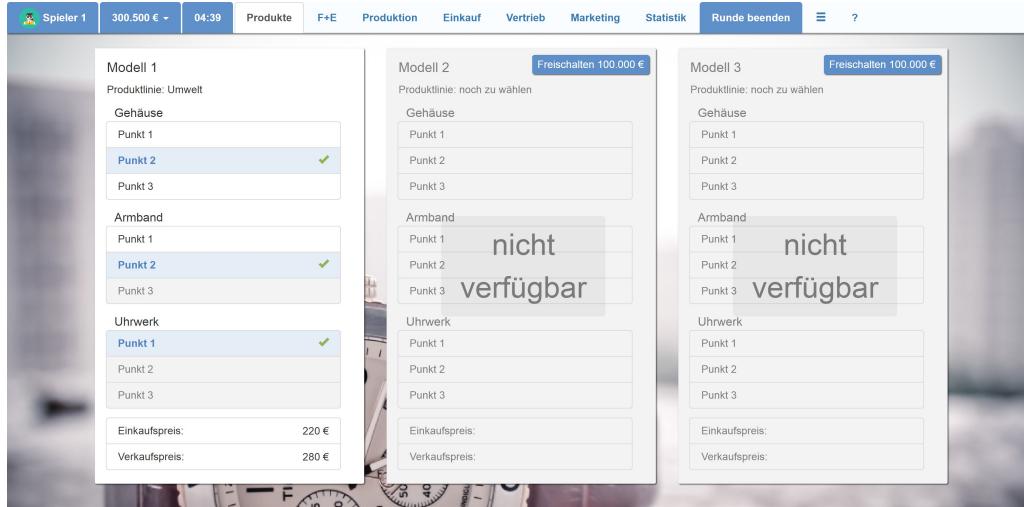


Abbildung 5.2: Mock-Up: Übersicht der Produkte

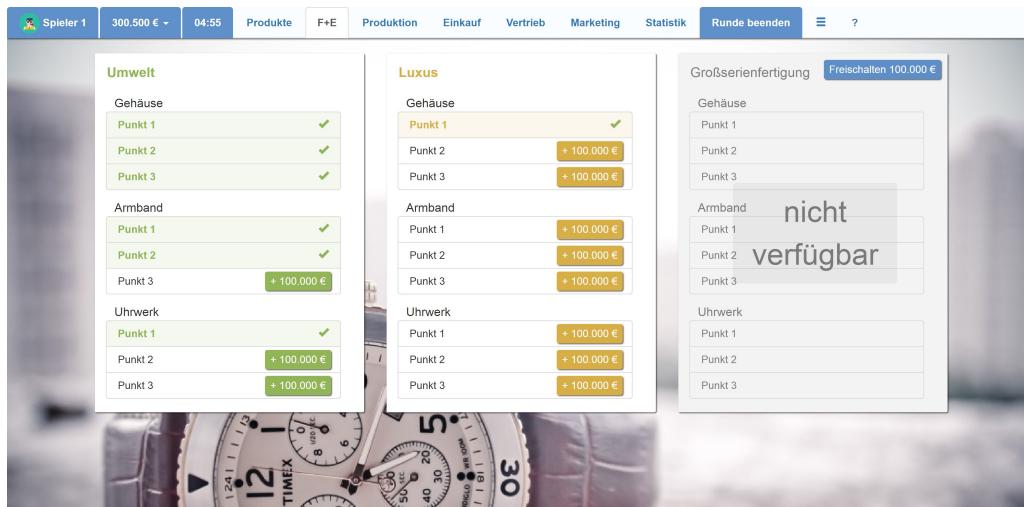


Abbildung 5.3: Mock-Up: Übersicht F&E

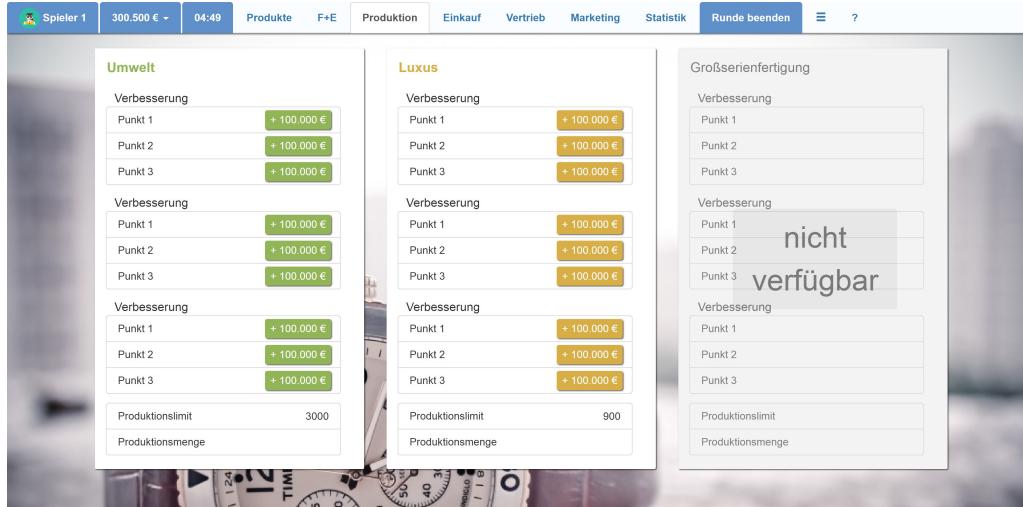


Abbildung 5.4: Mock-Up: Übersicht Produktion

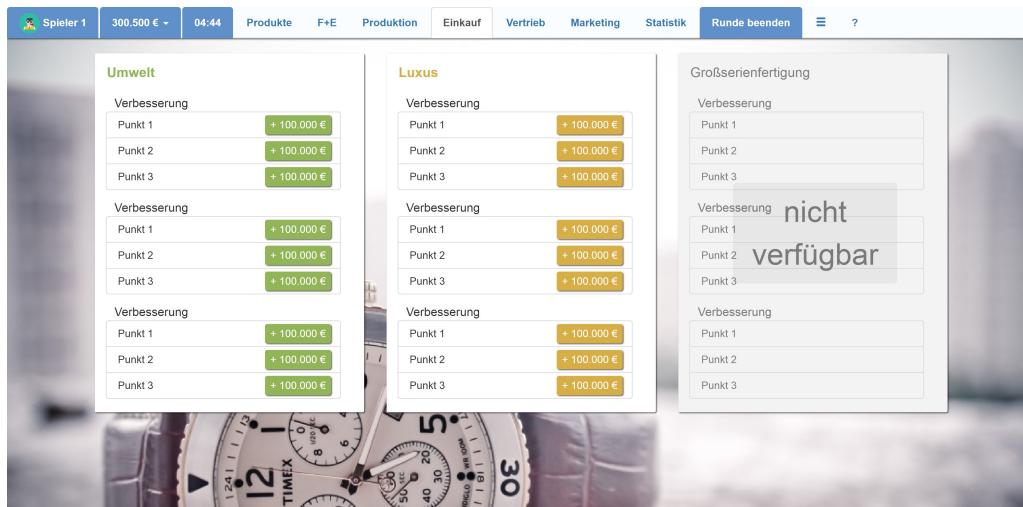


Abbildung 5.5: Mock-Up: Übersicht Einkauf

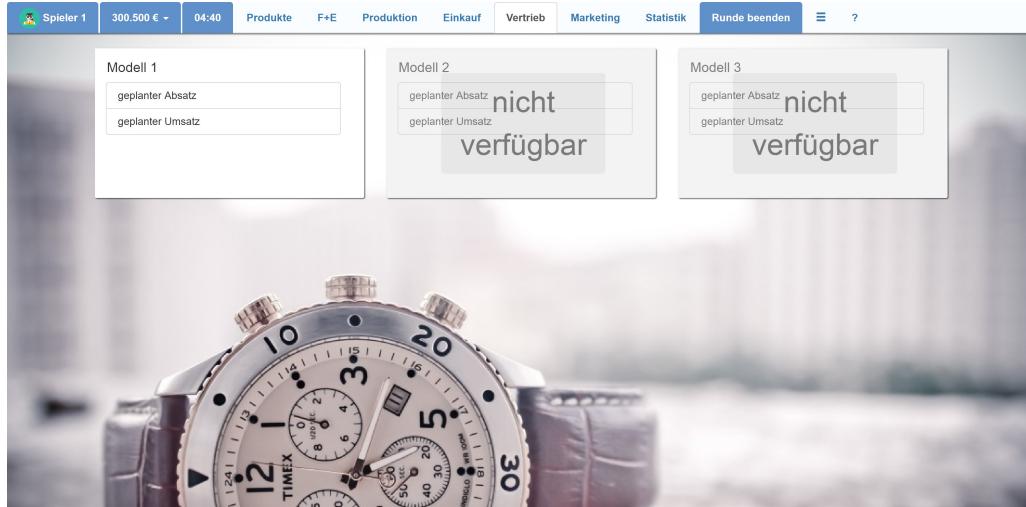


Abbildung 5.6: Mock-Up: Übersicht Vertrieb

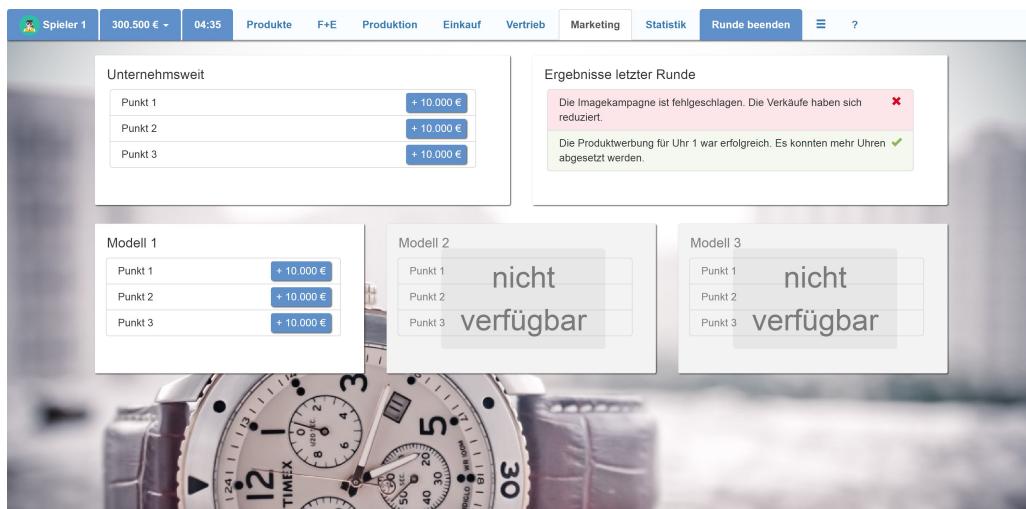


Abbildung 5.7: Mock-Up: Übersicht Marketing

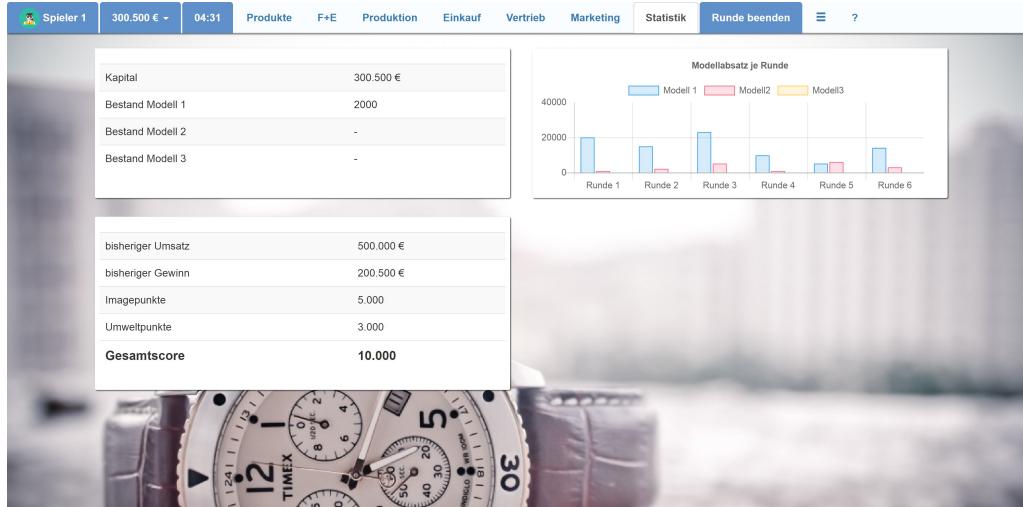


Abbildung 5.8: Mock-Up: Übersicht Statistik/Markt

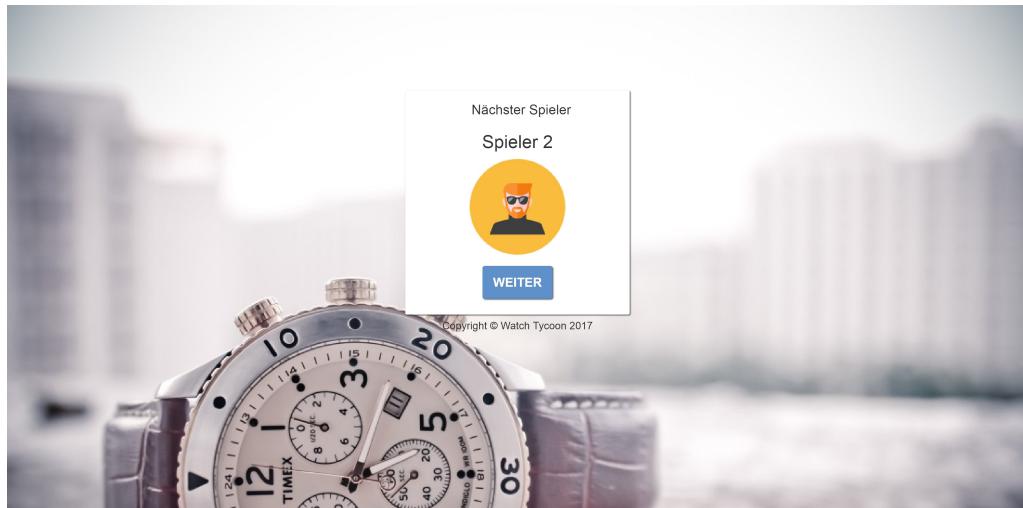


Abbildung 5.9: Mock-Up: „Nächster Spieler“-Anzeige

5.2 Finales UI

(NF) Der Einstiegsbildschirm zeigt die Auswahl der Spieleranzahl und startet mit einem Klick auf „START“ direkt mit dem Spiel und Spieler1.

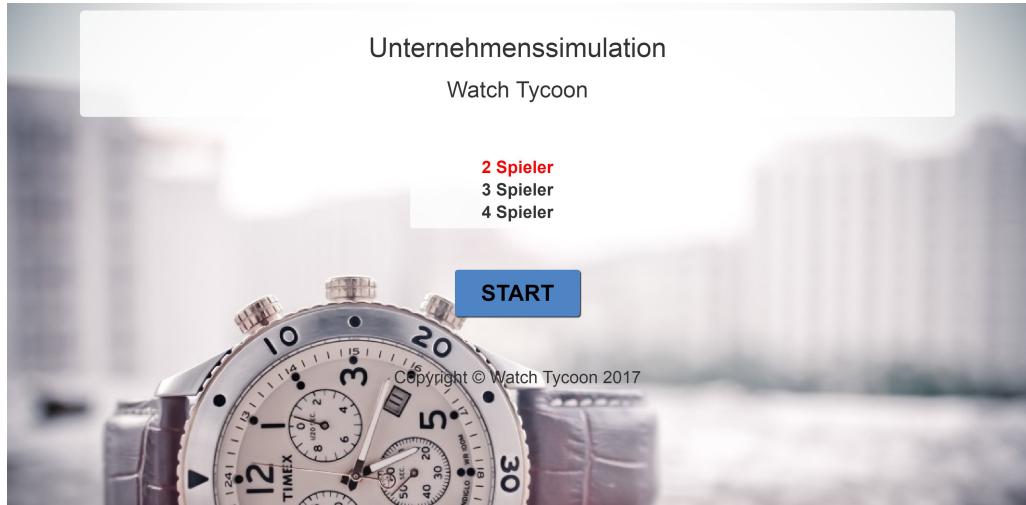


Abbildung 5.10: UI: Einstiegsbildschirm

Jeder Spieler bekommt in der 1. Runde die Auswahl zwischen den drei Segmenten und kann daraus ein Segment auswählen.

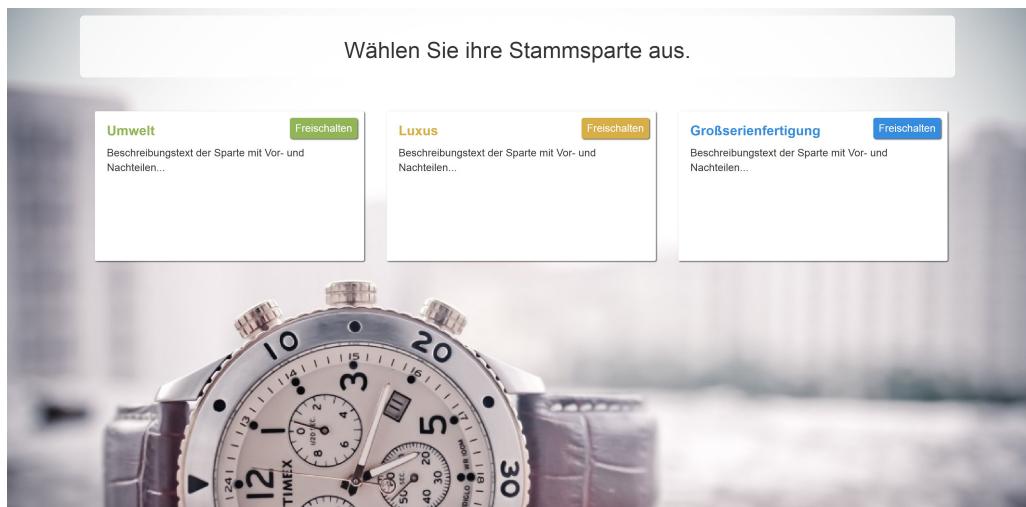


Abbildung 5.11: UI: Segmentauswahl für Spieler X

Hier erhält der Spieler eine Übersicht über seine max. drei Uhren und deren Materialien.

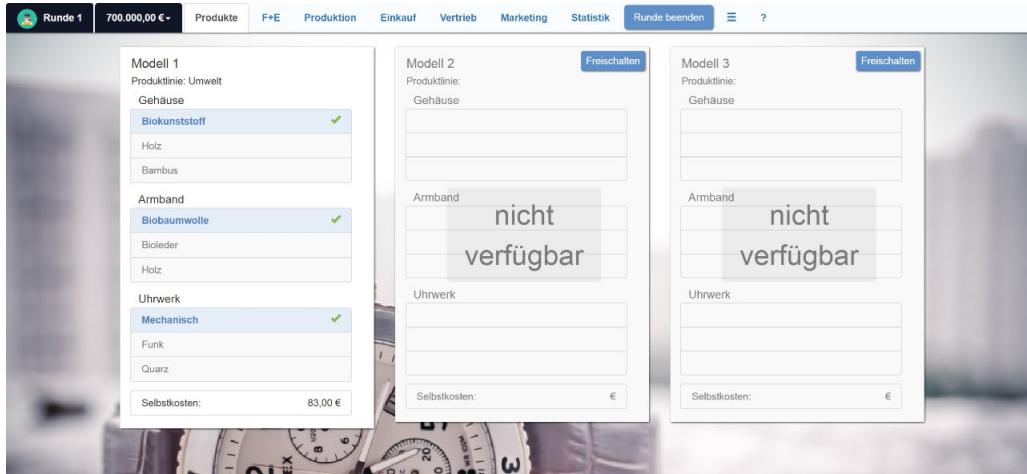


Abbildung 5.12: UI: Übersicht der Produkte

Zusätzlich erhält man mit einem „Klick“ auf das Kapital werden die verschiedenen Bestände angezeigt.

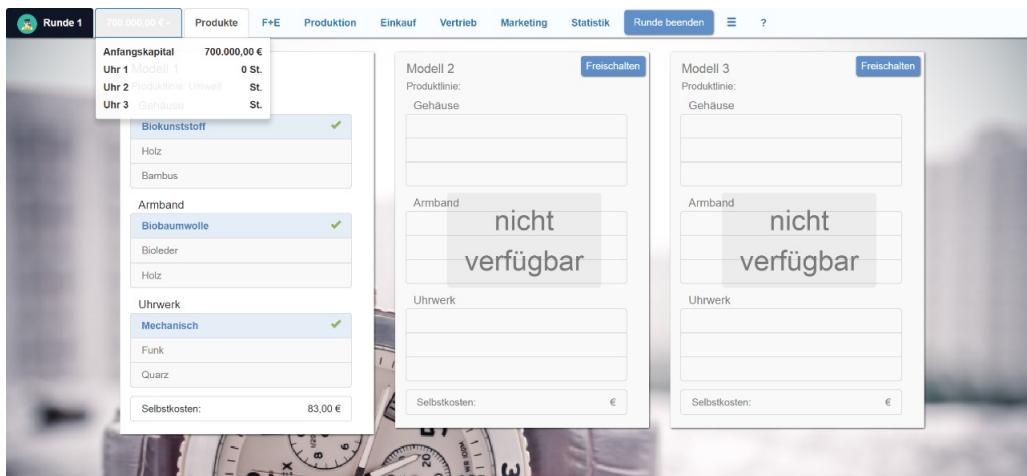


Abbildung 5.13: UI: Bestandsübersicht

Der Spieler kann Entwicklungen an seiner Uhr durchführen und hochwertigere Materialien erforschen.

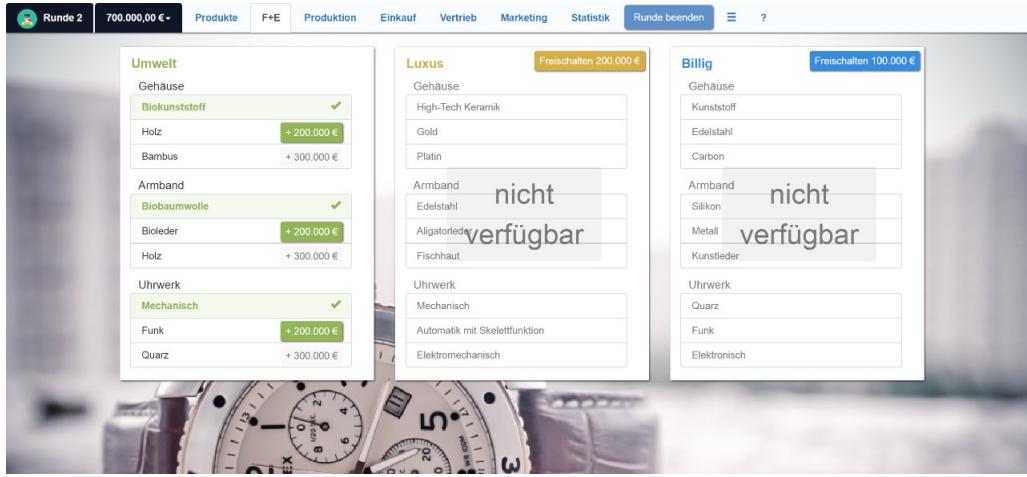


Abbildung 5.14: UI: Übersicht F&E

Im Reiter Produktion können weitere Produktionsstraßen und Produktionserweiterungen gekauft werden.

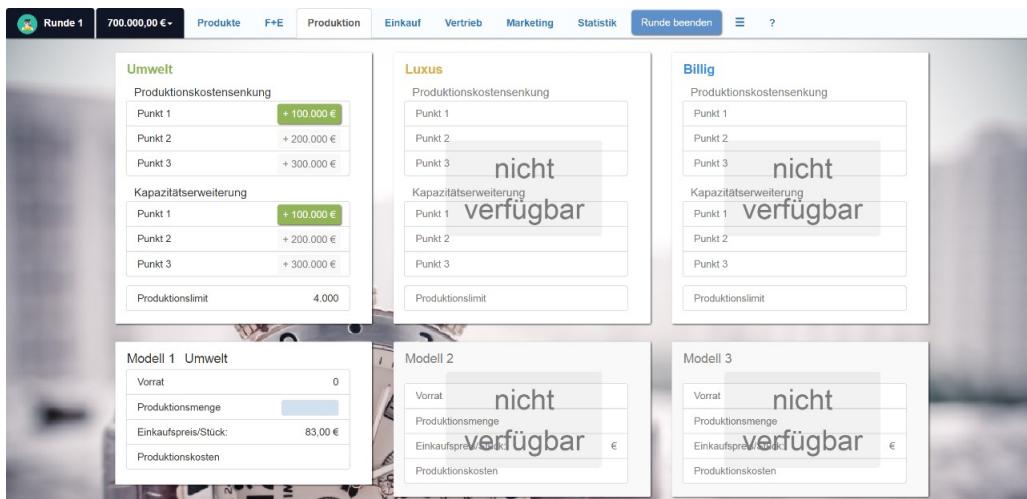


Abbildung 5.15: UI: Übersicht Produktion

Der Einkauf zeigt die jeweiligen Teilepreise für die Herstellung der Uhr.

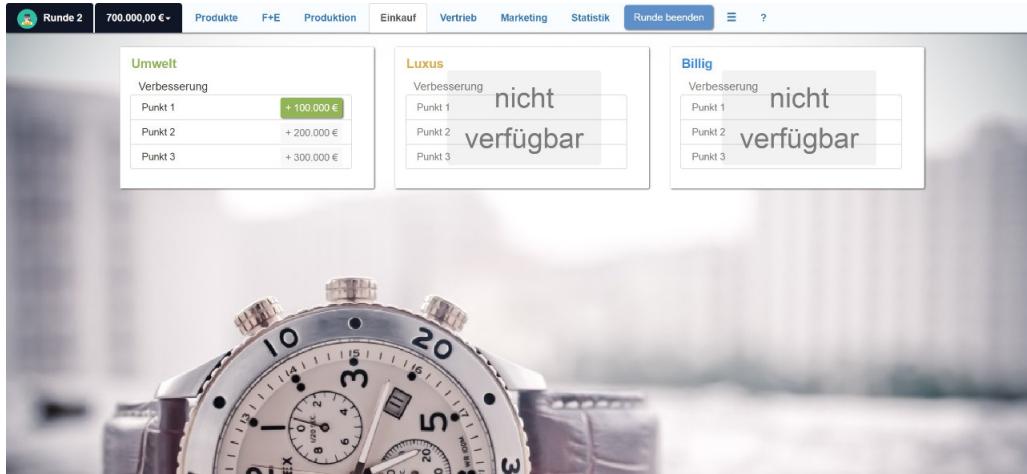


Abbildung 5.16: UI: Übersicht Einkauf

In den Spalten „Verkaufspreis“ und „geplanter Absatz“ können die jeweiligen Werte beliebig je nach Verfügbarkeit eingegeben werden.

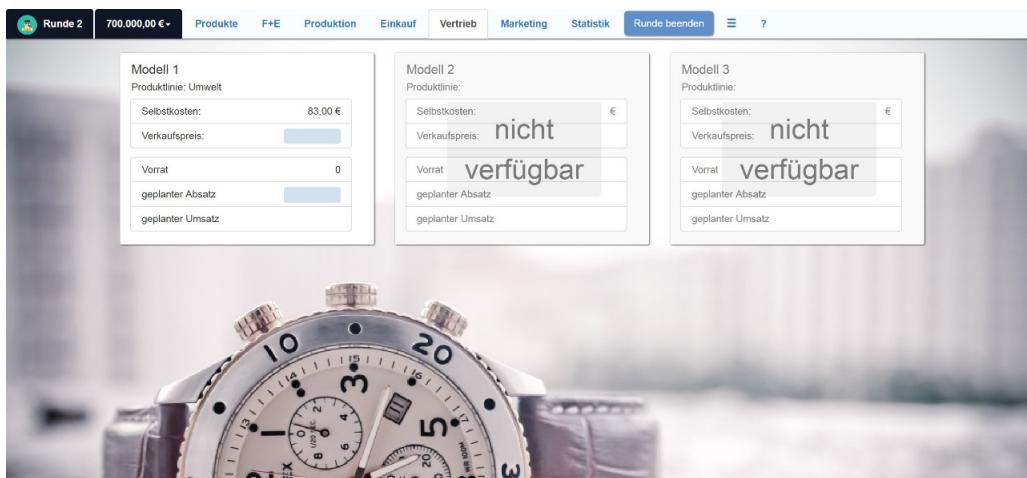


Abbildung 5.17: UI: Übersicht Vertrieb

Im Marketingbereich können verschiedene Unternehmens- oder Uhren-spezifische Kampagnen gestartet und deren Auswirkung eingesehen werden.

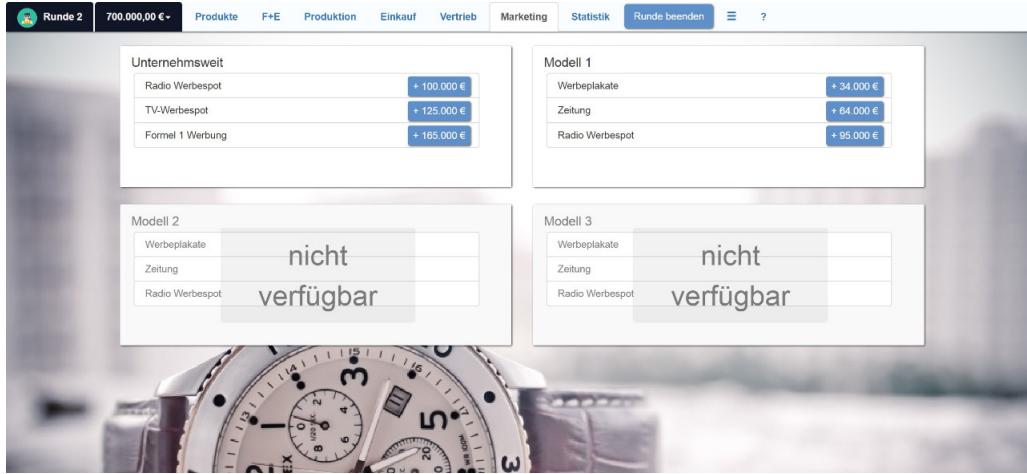


Abbildung 5.18: UI: Übersicht Marketing

Im letzten Reiter Statistik wird eine kleine Übersicht über alle wichtigen Spiel-Informationen geliefert.

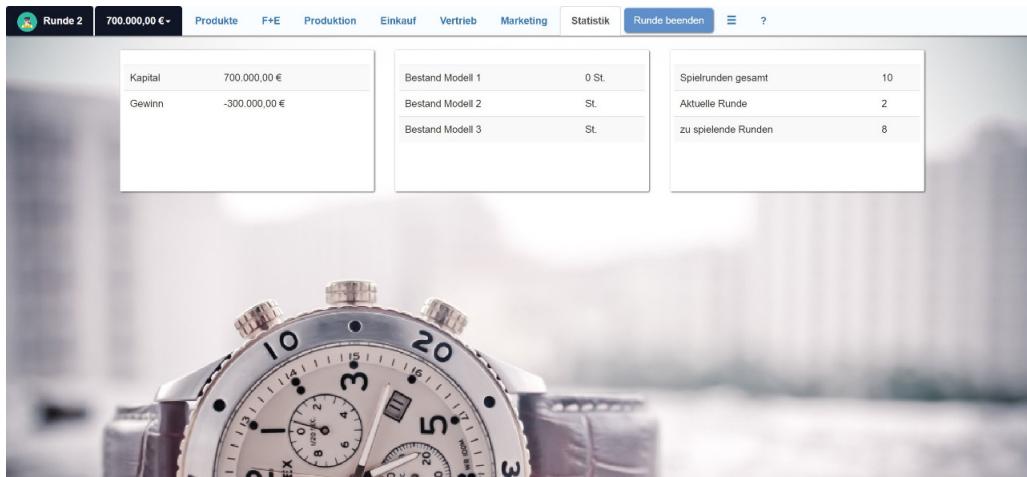


Abbildung 5.19: UI: Übersicht Statistik

Eine kleine Platzierungsübersicht gibt es am Ende auch.

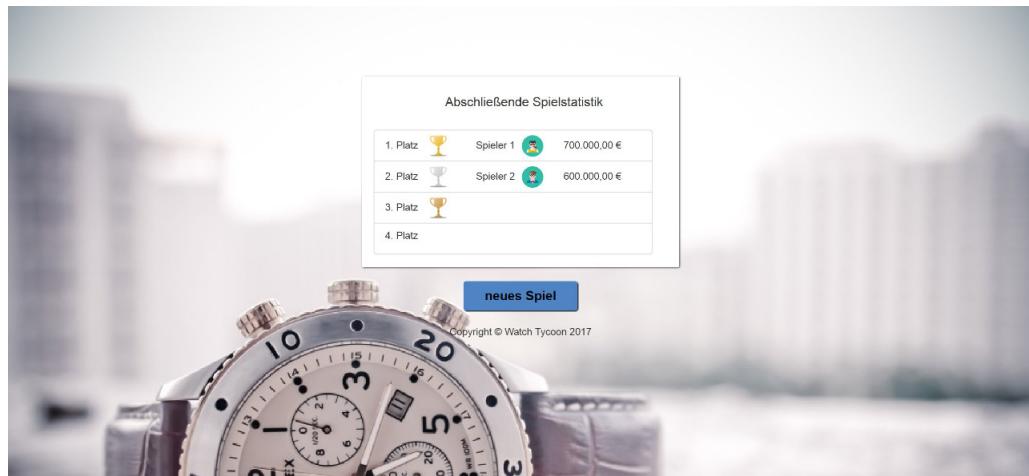


Abbildung 5.20: UI: Übersicht Sieger

6 Abteilungen

6.1 Produktion

(NF) Die Abteilung Produktion stellt für den Spieler die wichtigsten Funktionen zur Herstellung seiner Uhr zu Verfügung dar.

Der Spieler hat in dieser Abteilung zwei Möglichkeiten seine Produktionsstraßen auszubauen und seine Produktion um die zwei weiteren Segmente zu erweitern:

1. Erweiterung der Produktion

- In der Produktion können maximal zwei weitere Produktionsstraßen hinzugekauft werden. Dabei kann für jedes Segment (Masse, Öko, Luxus) genau eine Produktionsstraße eröffnet werden. Durch ist das Unternehmen in jedem Marktsegment vertreten und kann dort seine Uhren verkaufen.

2. Ausbau einer bestehender Produktionsstraße(n)

- Mit dem Ausbau einer bestehenden Produktionsstraße ist der Zukauf einer Produktionsstraße mit dem bereits vorhandenen Marktsegment gemeint. Dadurch erzielt das Unternehmen eine Kostensenkung der Produktion für das Marktsegment.

3. Erweiterung einer Produktionsstraße

- Eine bereits gekaufte Produktionsstraße kann ebenfalls erweitert werden. Dabei stehen insgesamt drei Stufen zur Verbesserung bereit. Dadurch wird eine Kostensenkung für die verbesserte Produktionsstraße erzielt.

6.2 Forschung&Entwicklung

(NF) Im Bereich Forschung und Entwicklung, kurz F&E, können für die diversen Segmente verschiedene Merkmale entwickelt werden. So ist es beispielsweise möglich für die Öko-Uhr ein Bambus-Gehäuse, für die Luxus-Uhr ein Fischhaut-Armband oder für die Massenproduktion ein elektronisches Uhrwerk zu entwickeln. Als diese und noch

einige andere Merkmale werten die Uhr auf und ermöglichen so einen höheren Preis, der bei dem Verkauf angesetzt werden kann.

6.3 Marketing

(RH) Der Unternehmensbereich Marketing teilt sich in unserer Unternehmenssimulation in Produktkampagnen und Unternehmenskampagnen auf. Produktkampagnen wirken sich in der Unternehmenssimulation nur auf das Marktsegment aus, in welches das Produkt zugehörig ist. Zum Beispiel wirkt sich eine Produktkampagne über das Produkt "Öko-Uhr" nur auf das Marktsegment dieser Uhren aus. Eine Unternehmenskampagne hingegen bewirbt das Unternehmen als Ganzes und nicht nur einzelne Produkte. Daher hat eine Unternehmenskampagne Auswirkungen auf alle Marktsegmente und somit auf den ganzen Markt.

Mögliche Werbungen, die in der Unternehmenssimulation zur Verfügung stehen, werden über die Medienmittel Zeitung, Plakat, TV und Radio an die Konsumenten vermittelt. Da die Kosten für z.B. eine TV-Werbung stark schwanken, aufgrund von unterschiedlichen Sendern und Uhrzeiten, haben wir uns auf einen Wert für das jeweilige Werbemittel geeinigt. In der folgenden Tabelle sind die Kosten für das jeweilige Werbemittel zu sehen.

Produktkampagne	€	Unternehmenskampagne	€
Plakate	34.000	Radio	100.000
Zeitung	64.000	TV	125.000
Radio	95.000	Sonderwerbung bei Formel 1	165.000

Es ist in jeder Runde möglich alle Werbemittel, sowohl die der Produktkampagnen als auch die der Unternehmenskampagne auszuwählen. Der Marketingeffekt wirkt sich auf die absetzbare Menge an Uhren aus und ist begrenzt auf eine Runde.

6.4 Vertrieb

(EA) In dieser simulierten Abteilung der Unternehmenssimulation „Watch Tycoon 2017“ kommt die Vorarbeit, der vorher erläuterten Abteilungen, zu einem Abschluss. Für jeden Uhrtypen soll die Möglichkeit existieren eine Verkaufsmenge anzugeben.

1. Einzelhandel

- In der Versorgungskette steht der Einzelhandel zwischen dem produzierenden Unternehmen und dem Endverbraucher (B2B). Insofern vertreibt der Einzelhandel die Uhren der Unternehmenssimulation. Logischerweise ist der

Preisgestaltung eine Grenze gesetzt, da der Einzelhändler bereit sein muss das Produkt zu kaufen und es selbst auch gewinnbringend verkaufen muss. Dieser Vertriebsweg ist für alle Uhrtypen geeignet.

2. Direktvertrieb

- Der Direktvertrieb überspringt, in der Versorgungskette, den Einzelhandel und bildet somit die Brücke des Unternehmens direkt zum Verbraucher (B2C). Insoweit ist es für das Unternehmen notwendig einen eigenen Groß- bzw. Einzelhandel zu errichten. Dies verursacht somit zusätzliche Kosten, welche über den normalen klassischen Vertriebsweg nicht anfallen. Der Vorteil besteht darin, dass der Preis nicht mit dem Einzelhändler verhandelt werden muss.

Dieser Vertriebsweg wird gewöhnlich für hochwertige Güter genutzt. In dieser Unternehmenssimulation ist er für Luxus-Uhren besonders geeignet.

6.5 Einkauf

(LK) Der Einkauf sollte dem Spieler zunächst drei Optionen bieten:

1. Der Einkauf von Rohstoffen

Da Uhren oft aus sehr individuellen Materialien bestehen, sollte es hier eine Liste möglicher Rohstoffe geben, die je nach Marktsegment ausgewählt werden können.

- Holz
- Textilien
- Kunststoffe
- Edelstahl
- Reintitan
- Aluminium

- High-Tech-Keramik
- Gold, Silber, Platin
- Edelsteine und Perlen
- Leder, Wildleder, Krokodilsleder
- Fischhaut
- Glas

2. Der Einkauf von halbfertigen und fertigen Erzeugnissen

Da die Hauptbestandteile einer Uhr aus vielen kleinen Einzelteilen bestehen, werden diese meist durch Fremdbezug erworben. Bei dieser Option sollen also die bereits hergestellten Hauptbestandteile eingekauft werden.

3. Die Eigenproduktion durch beispielsweise einen eigenen Holzanbau

Da die zweite Option auch in der Realität weit verbreitet ist, wurde beschlossen, dass sich auf die Auswahl eines Armbands, eines Gehäuses und eines Uhrwerks beschränkt wird.

Des Weiteren sollte es die Möglichkeit geben, Lieferanten gezielt auswählen zu können und dabei auf Faktoren wie Kosten, Regionalität und Transportmittel zu achten. Diese Option wurde aufgrund des hohen Aufwands jedoch nicht in das Spiel übernommen.

Auch Skonti und Rabatte sollen im Einkauf möglich sein. Sie sind vor allem mengenabhängig und richten sich nicht nach einzelnen Lieferanten.

Für die Produkte, die im Ökosegment angeboten werden können, sollte es beim Einkauf zusätzliche Informationen zu Siegeln wie beispielsweise dem Blauer Engel für Leder und Fairtrade für Gold und Edelsteine geben. Diese Möglichkeit kommt als Erweiterung des Spiels in Frage, da sie zunächst nicht spielentscheidend oder notwendig für die Funktionalität ist.

7 Markt

7.1 Aufbau

(TH) Der Markt stellt die zentrale Einheit dar, die für jede einzelne Angebotene Uhr die abgenommene Menge bestimmt. Dabei gelten folgende Vorgaben und Annahmen.

1. Der Markt soll in drei Marktsegmente unterteilt sein
Entsprechend der drei Uhrenkategorien soll auch der Markt in drei Segmente geteilt sein, die größtenteils getrennt voneinander agieren.
2. Jeder Markt soll jedem Spieler die gleichen Möglichkeiten bieten.
Um jedem Spieler, unabhängig von der Wahl seines Urtyps, die Chance auf den Sieg zu geben sollen die 3 Teilmärkte, in die der Markt gegliedert, jeweils das gleiche Volumen besitzen. Dies ist zwar nicht realitätsgerecht, verhindert jedoch ein zu langweiliges Spiel.
3. Die Uhren sollen sich je nach Preis und Ausstattung unterschiedlich gut verkaufen
Die durch den Markt simulierten Konsumenten sollen nicht einfach nur blind nach ihrem Budgetkaufen, sondern schlechte Angebote unterschiedlich stark nachfragen.
4. Die drei Marktsegmente sollen sich gegenseitig beeinflussen
Jede Uhr soll die Möglichkeit haben, die Anderen zu beeinflussen, auch wenn diese in anderen Marktsegmenten positioniert sind.
5. Die Uhren im selben Marktsegment sollen sich gegenseitig beeinflussen
Uhren im selben Preissegment und mit ähnlichen Preisen konkurrieren um die selben Kunden, dies muss sich auch in den Verkaufszahlen widerspiegeln.

Die genaue Implementierung des Marktes lässt sich auch in diese Unterpunkte unterteilen. Dazu wird von den anderen Teilen der Unternehmenssimulation erwartet, dass jeder Spieler, ein Objekt der Klasse /enquote/textttUnternehmen ist. Diese werden beim Starten der Marktsimulation als Array übergeben und besitzen wiederum ein Array an Uhren. Jede Uhr hat alle benötigten Attribute, um die Simulation durchzuführen, wie die offensichtlich benötigten Werte Angebotspreis, angebotene Menge und

Marktsegment. Als zusätzlichen Parameter muss der Uhr noch einen Wert zugewiesen werden. Dieser spiegelt wieder, dass Uhren mit besseren Uhrwerken, Gehäusen oder Armbändern auch zu teureren Verkaufspreisen angeboten werden können. Daher ist dieser Wert angelehnt an die Entwicklung dieser Attribute und die daraus resultierende Produktionskostensteigerung. Der Marktwert wird automatisch beim Verbessern der Uhren aktualisiert und steht immer in jeder einzelnen Uhr zur Verfügung. Die komplette Marktsimulation kann mit diesen Werten ausgeführt werden.

1. Der Markt soll in drei Marktsegmente unterteilt sein

Die Unterteilung des Marktes in die einzelnen Marktsegmente wird über eine Instanz der Klasse „**Gesamtmarkt**“ verwaltet, die intern über je eine Instanz der Klasse „**Teilmärkt**“ für jedes Segment verfügt. Diese werden zu Beginn, beim Erstellen des Marktes, automatisch mit dem im **Gesamtmarkt**-Konstruktor mitgegebenen Parametern gleichermaßen erstellt.

2. Jeder Markt soll jedem Spieler die gleichen Möglichkeiten bieten.

Dem **Gesamtmarkt**, und damit den Teilmärkten, wird ein „**volume**“ übergeben. Dieser Wert kann als Umsatz zu jedem Preis über das gesamte Preisspektrum erreicht werden. Die Formel zur Berechnung des Marktvolumens in Stückzahlen zu einem bestimmten Preis lautet demnach: $\text{Marktvolumen}(\text{Preis}) = \text{volume}/\text{Preis}$.

3. Die Uhren sollen sich je nach Preis und Ausstattung unterschiedlich gut verkaufen

Für die Simulation dieser Anforderung wurde angenommen, dass ein Konsument ein begrenztes Budget besitzt. Nach seinen Budgetvorstellungen versucht er nun eine Uhr zu kaufen, bevorzugt dabei aber natürlich besser ausgestattete Uhren bzw. Uhren mit besserem Preis-Leistungs-Verhältnis. Dazu wird in den Uhrenklassen eine Abnahmefrage berechnet. Diese soll eine spätere Bevorzugung von preislich aggressiveren Angeboten bei gleichzeitiger Benachteiligung von überteuernten Angeboten ermöglichen.

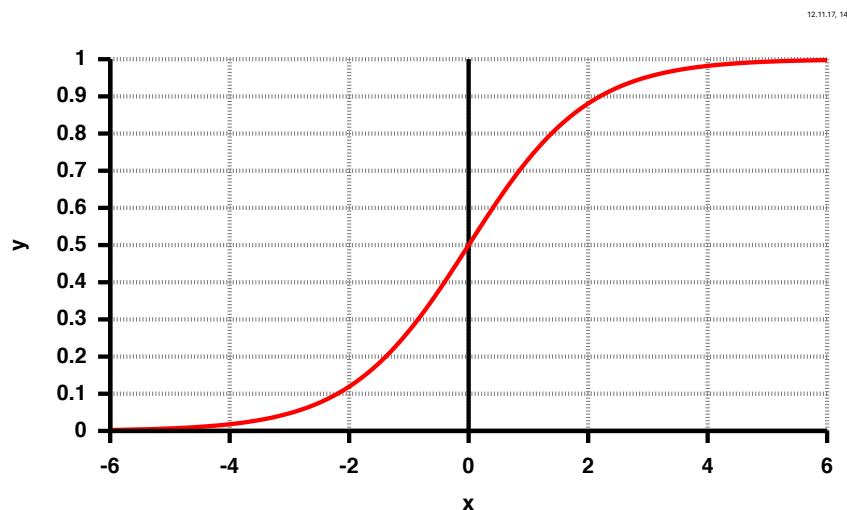
Der erste Schritt dabei stellte die Abbildung des Preis-Leistungs-Verhältnisses, das Werte im Bereich $[0; \infty]$ und das ausgeglichene Verhältnis bei 1 auf eine nicht-logarithmische Skala. Dies hat die Aufgabe, das z. B. ein Verhältnis von $2 : 1$ den gleichen Wert wie ein Verhältnis von $1 : 2$ bekommt, nur mit unterschiedlichen Vorzeichen, was durch folgende Funktion erreicht wird:

$$\frac{2,5 * \log(\frac{\text{Preis}}{\text{Marktwert}})}{\log 2}$$

Um diesen Wert nun wieder auf ein Quotient mit Werten im Bereich $[0; 1]$ abzubilden, wurde diese Funktion wiederum mit einem logistischen Wachstum kombi-

niert, wie in der nächsten Formel zu sehen.

$$\frac{1}{1 + e^{\frac{2,5 \cdot \log(\frac{\text{Preis}}{\text{Marktwert}})}{\log 2}}}$$



https://upload.wikimedia.org/wikipedia/commons/f/f7/Logistische_Funktion.svg

Page 1 of 1

Abbildung 7.1: Logisches Wachstum

Die aus dieser Formel resultierende Abnahmefrage wird nun noch mit der angebotenen Menge und dem sog. „Marketingboost“ multipliziert, was nachbildet, dass unattraktive Uhren, unabhängig von der angebotenen Menge, eher zu Ladenhütern werden als attraktivere und im Programm als „Abnahmepotential“ bezeichnet wird. Bei schlechten Angeboten wird mit diesem Algorithmus errechnet, welcher Prozentteil des Angebots zu Ladenhütern wird und welcher im Gegenzug überhaupt verkaufbar ist. Ebenso wird Uhren mit größerem Abnahmepotential später ein gewisser Vorteil eingeräumt, da diese eine größere Präsenz in den Läden haben.

4. Die drei Marktsegmente sollen sich gegenseitig beeinflussen
Um anzugeben, wie stark sich die Teilmärkte gegenseitig beeinflussen wurde davon ausgegangen, dass in der Ausgangslage die Kunden einem Segment zugeordnet werden können und so die unabhängigen Marktvolumina darstellen. Jedoch können Teile der Kundenbasis durch attraktive Angebote abgeworben werden. Dazu wird für jede Uhr der sog. „volumeChange“ berechnet, bevor diese

an die Teilmärkte übergeben wird. Bei guten Angeboten wird das Marktvolumen des korrespondierenden Teilmarktes um diesen `volumeChange` erhöht und die anderen Teilmarktvolumina um die Hälfte dieses Wertes verringert.

5. Die Uhren im selben Marktsegment sollen sich gegenseitig beeinflussen
Damit berechnet werden kann, wie stark sich alle Uhren im selben Marktsegment gegenseitig beeinflussen werden alle Uhren aller Spieler zuerst in drei Listen sortiert, die dann jeweils den drei Teilmärkten übergeben werden. So ist es egal wie viele Uhren in einem Segment angeboten werden oder von wem, denn auch die eigenen Uhren sind als Konkurrenz zu sehen. In den Teilmärkten wird dann für jede Uhr die sog. „**konkurrenz**“ berechnet, die Stückzahl aller konkurrierenden Uhren zu dem Preis der Ausgangsuhr. In dieser Rechnung wird jedoch bereits das in Punkt 3 berechnete Abnahmepotential statt der angebotenen Menge benutzt, das Preis-Leistungs-Verhältnis ist also bereits eingerechnet. Dabei ist eine Uhr definiert als eine Uhr, deren Preis (unabhängig vom Wert) innerhalb eines Preisspektrums zur Ausgangsuhr liegt. Dieses Spektrum ist definiert durch die beim Erstellen des Marktes angegebene Variable `impactRange`, ist die Differenz der Angebotspreise der beiden Uhren kleiner als das Produkt aus Angebotspreis der 2. Uhr mit diesem `impactRange`, zählen die Uhren als konkurrierend. Sind 2 Uhren in direkter Konkurrenz wird das **Abnahmepotential** zur **konkurrenz** addiert, nachdem es mit einem Faktor zwischen 1 und 0 multipliziert wurde, der kleiner wird je weiter auseinander die Preise liegen. Ist die gesamte **konkurrenz** kleiner als das Marktvolumen zu dem Preis der Uhr ist der Markt nicht gesättigt und die Uhr kann alle beim **Abnahmepotential** berechneten Uhren verkaufen, sollte die **konkurrenz** größer sein, werden die Verkaufszahlen aller Uhren soweit gleichmäßig geschrumpft, dass das Marktvolumen nicht überschritten wird.

Diese Marktsimulation hat zur Folge, dass Uhren die zu attraktiven Preisen angeboten werden bevorzugt werden, übererteute Uhren jedoch nicht komplett Machtlos sind und mit schierer Masse Marktanteile erobern können. Ebenso kann durch attraktive Angebote die Käuferschaft von anderen Marktsegmenten abgeworben werden und so über den gesamten Markt Einfluss ausgeübt werden.

7.2 Klasse

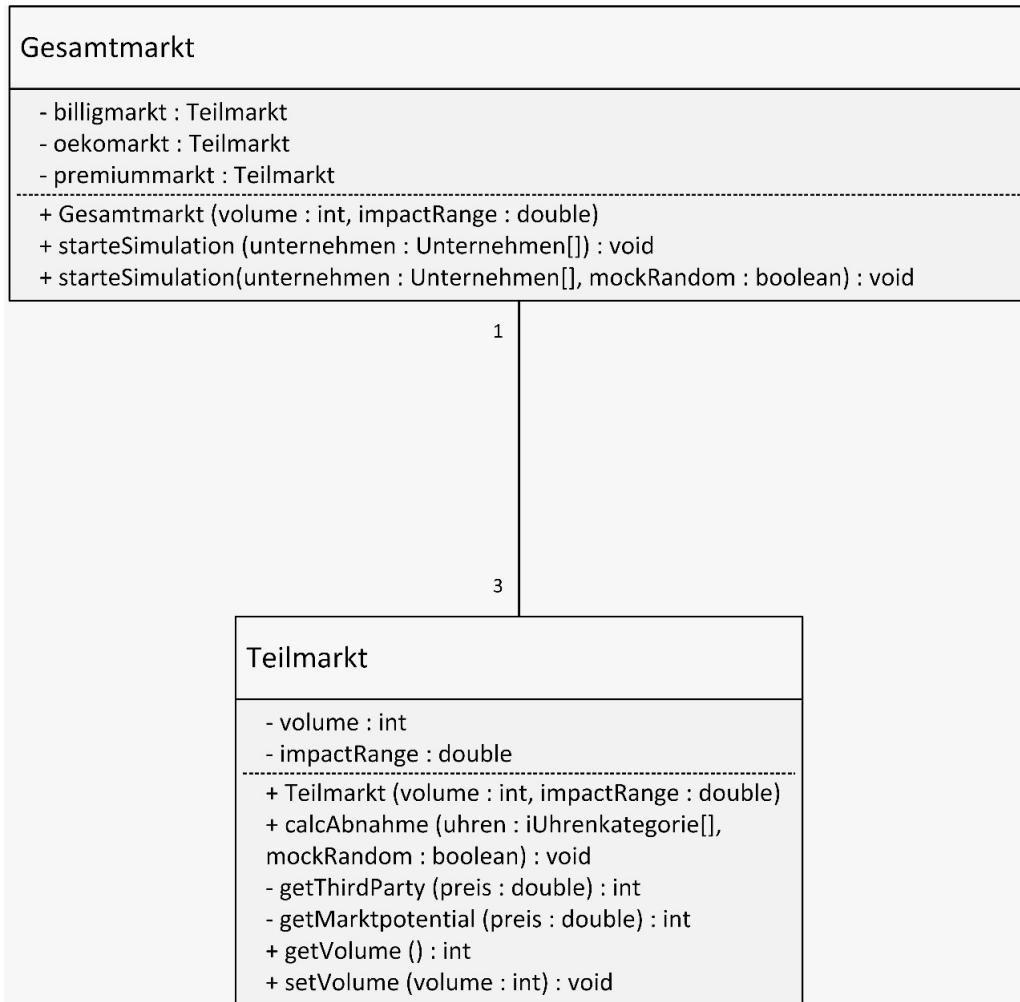


Abbildung 7.2: Markt Klasse

8 Diagramme

8.1 UML-Diagramme

8.1.1 Idee/Erstes UML-Diagramm

(MW) Das Erste UML-Diagramm entstand nach dem ersten Brainstorming in der Analysephase. Alle Teammitglieder diskutierten über das mögliche Spielkonzept. Dabei wurden die ersten Ideen gesammelt und aufgrund der Ideen wurde das erste UML-Diagramm entwickelt.

Nach weiteren Besprechungen und zu Beginn der Umsetzung des UML-Diagramms, traten verschiedene Fehler auf, aufgrund einer Implementierung zu diesem Zeitpunkt unmöglich war. Beispielsweise wurde die einzelnen Uhren nicht im jeweiligen Unternehmen erzeugt, d.h. der Spieler hätte keinen Zugriff darauf oder, dass ein Spielbrett zur Verwaltung des kompletten Spielablaufs gefehlt hat.

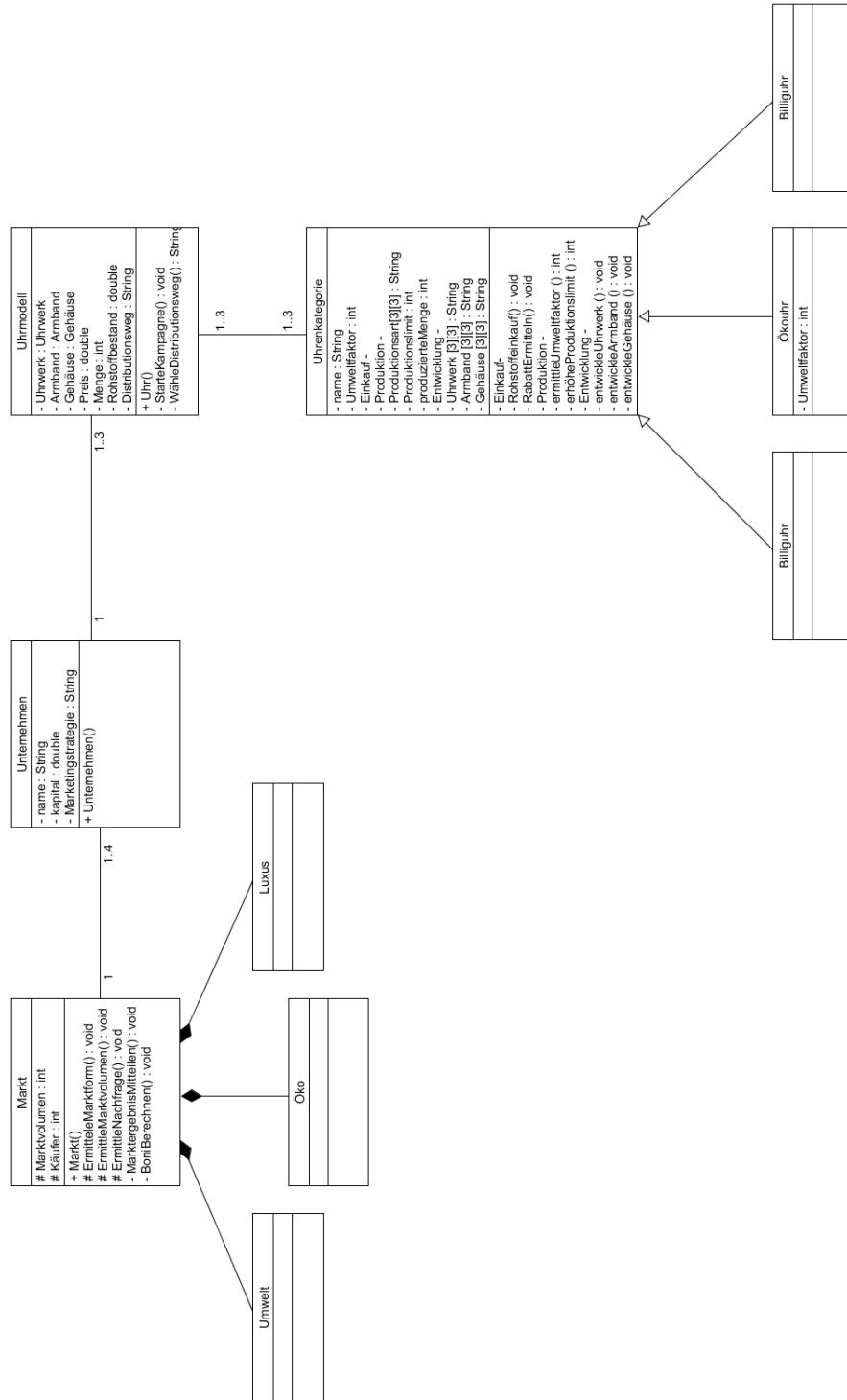


Abbildung 8.1: Erstes UML

8.1.2 Finales UML/Unternehmenssimulation

(MW) Das Finale UML-Diagramm der Unternehmenssimulation entstand bei der Erstellung des Fachkonzeptes. Während der Entwicklung des Diagrammes wurden mehrere Meetings gehalten, das UML-Diagramm verbessert und die Entwürfe ins Fachkonzept umgesetzt. Die anstehenden Fehler wurden wiederum im nächsten Meeting besprochen und das Diagramm angepasst. Am Ende der Entwicklungsphase war das Finale UML-Diagramm fertig.

Kapitel 8

Diagramme

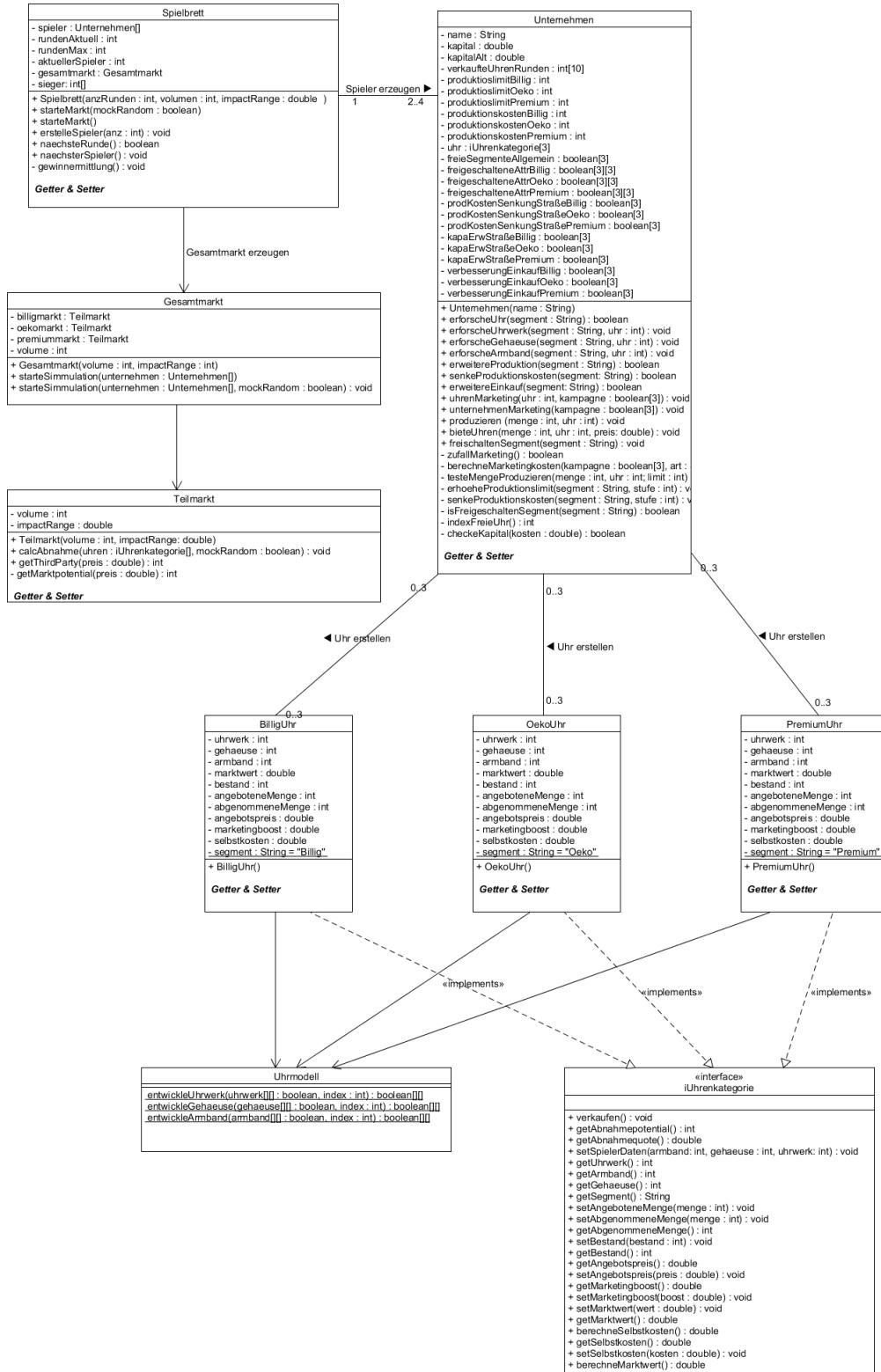


Abbildung 8.2: Finales UML von „Watch Tycoon 2017“

8.2 Zustandsdiagramm

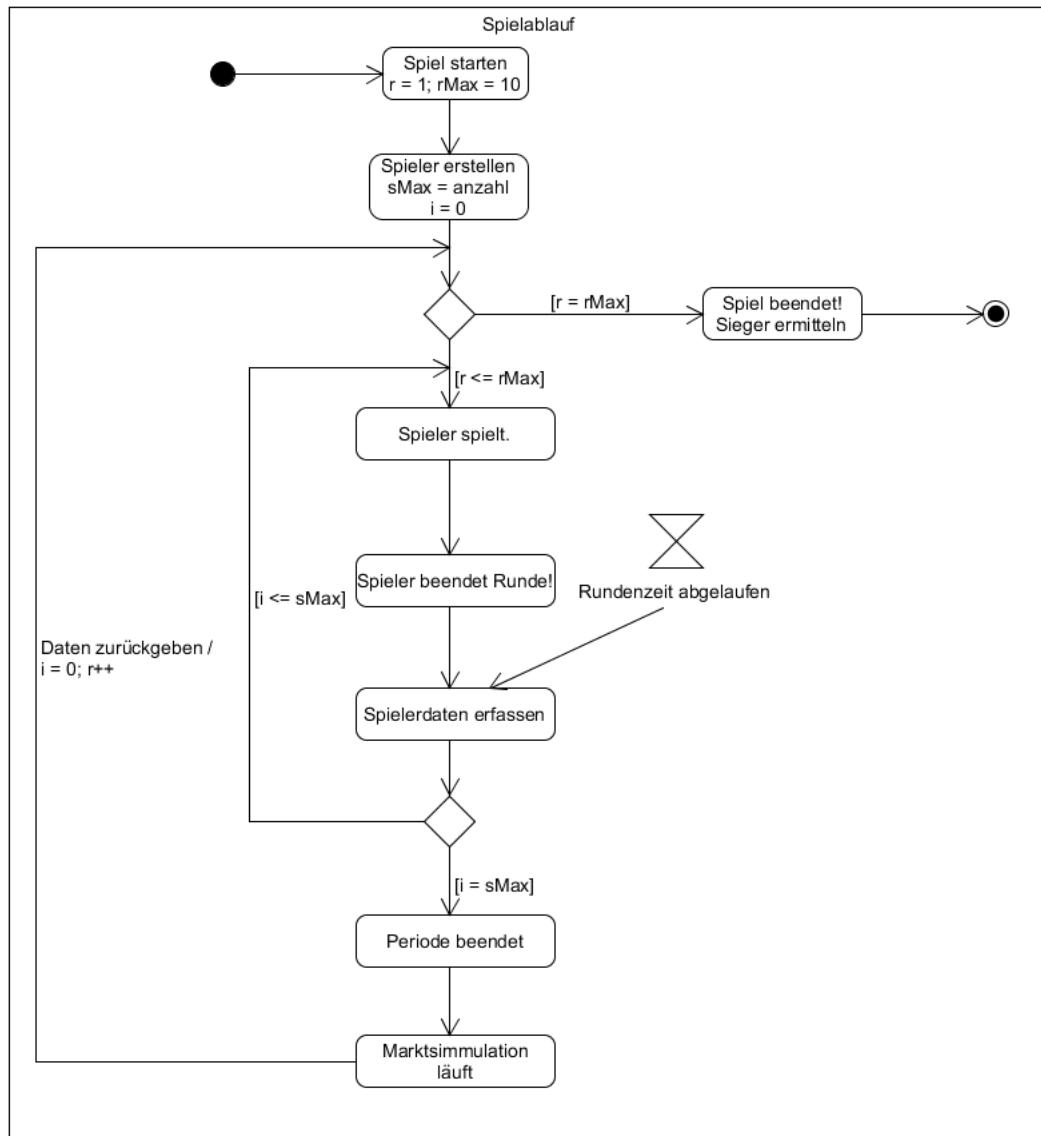


Abbildung 8.3: Zustandsdiagramm einer Spielrunde

9 Architektur

(ES) In diesem Kapitel wird der technische Aufbau des Spiels beschrieben. Hierbei wird auf die verwendete Architektur sowie die zugrunde liegenden Designentscheidungen eingegangen.

9.1 Einflussfaktoren für Architektur

(ES) Die Maßgebenden Einflussfaktoren für die Entscheidung der verwendeten Softwarearchitektur, sind neben dem im Team vorhanden Know-How bezüglich Programmiersprachen und Unit-Tests, die eigenen Anforderungen an die optische Repräsentation des Spiels. Nach einer Aufnahme der Kenntnisse der Teammitglieder war schnell ersichtlich, dass die Programmiersprache Java am verbreitetsten ist und sich in Bezug auf Unit-Tests auch hier die größte Übereinstimmung findet.

Der in der Vorlesung zur Fallstudie vorgestellte Tomcat-Server bietet die Möglichkeit, für ein in Java geschriebenes Fachkonzept ein Userinterface auf HTML-Basis bereitzustellen. Da die in Java zur Verfügung stehende SWING-Bibliothek nicht die gewünschten optischen Resultate erzielt, wurde diese schnell ausgeschlossen.

9.2 Übersicht über technischen Aufbau

(ES) Aus den zuvor genannten Beweggründen ergab sich schnell der in 9.1 dargestellte Aufbau der Software. Im ersten Schritt zur Gliederung der Architektur wurde die aus dem Studium bekannte 3-Schichten-Architektur zugrunde gelegt. Die Software wird hierbei in die ebenen Datenhaltung, Fachkonzept sowie Darstellung untergliedert. Bei den vorgegebenen Anforderungen an die Fallstudie, ist eine Datenhaltung als optional angegeben. Zur Reduzierung des Programmieraufwandes wurde auf eine Datenbankanbindung oder sonstige Datenhaltung verzichtet.

Zur Umsetzung des Fachkonzeptes wird auf die Programmiersprache Java gesetzt. Wobei die gesamte Spiellogik so umgesetzt ist, dass hierauf ein beliebiges Userinterface aufgesetzt werden kann. Zum Testen des Fachkonzepts kommt JUnit zum Einsatz. Da

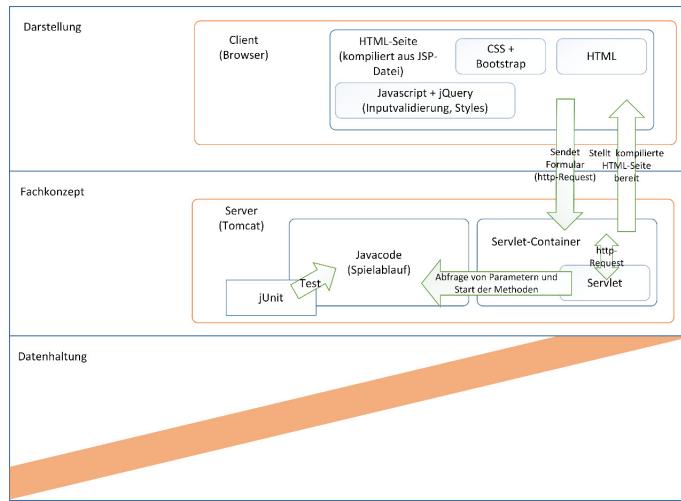


Abbildung 9.1: 3-Schichten-Modell

als Userinterface eine Webseite dienen soll, wird das Fachkonzept nicht in einer lokalen JVM sondern auf einem Webserver ausgeführt. Der zum Einsatz kommende Tomcat-Server kann sowohl den Java-Code des Fachkonzepts ausführen, als auch über die gewünschte Webseite Eingaben des Anwenders entgegen nehmen, bzw. entsprechende Ausgaben zurückgeben. Die dabei zum Einsatz kommenden Techniken werden in den folgenden Kapiteln näher erläutert.

Bei der Darstellungsebene des Spiels werden die gestalterischen Möglichkeiten von HTML, CSS sowie JavaScript genutzt, indem der Tomcat-Server für den Anwender eine entsprechende Webseite als Userinterface zur Verfügung stellt. Die Bedienung des Spiels erfolgt somit für den Anwender in einem Webbrowser.

Folglich ergibt sich, neben einer logischen Trennung von Fachkonzept und Darstellung, eine technische Trennung, in Form einer Client-Server-Architektur. Auf dem Server wird die Spiellogik ausgeführt, eine dynamische Webseite generiert und auf Eingaben des Anwenders gewartet. Im Browser des Clients wird hingegen die dynamisch generierte Webseite mit den aktuellen Werten des Spielstandes angezeigt, als auch eine Eingabemöglichkeit zur Ausführung von gewünschten Spielaktionen geboten.

9.3 Laufzeitumgebung

(ES) Da die Spiellogik in Java programmiert ist, wird zur Ausführung des Programms eine Java-Laufzeitumgebung benötigt. Während der Entwicklung wird hier im speziellen die Java Runtime Environment in Version 8 verwendet. Eine weitere wichtige Komponente ist der Servlet-Container. Die an den Server übermittelten Eingaben,

in Form von Http-Requests, werden vom Webserver entgegen genommen und an den Servlet-Container weitergeleitet. Dieser verarbeitet die Requests und leitet diese an das Servlet, welches im folgenden Kapitel näher beschrieben wird, weiter. Zur Bereitstellung einer serverseitigen Java-Laufzeitumgebung, als auch einem Servlet-Container, dient der bereits angesprochene Tomcat-Server von Apache.

9.4 Java Servlets

Auf dem bereits erwähnten Tomcat-Server wird eine Instanz der Klasse Servlet ausgeführt. Hierüber wird es ermöglicht, auf Eingaben des Anwenders zu reagieren und entsprechende Rückgaben auszuliefern. Die Kommunikation zwischen Server und Client erfolgt hierbei mittels Http-Requests. In diesem Fall wird die doPost()-Methode der Klasse HttpServlet überschrieben und mit den gewünschten Funktionalitäten versehen.

Der Aufruf der Methode erfolgt von Seiten des Clients durch absenden eines HTML-Formulars. Im Kapitel 9.6 auf Seite 50 wird hier näher drauf eingegangen. Innerhalb der doPost()-Methode können über das request-Objekt auf die vom Anwender getätigten Eingabeparameter zugegriffen werden. Aufgrund der Eingaben des Anwenders werden die entsprechenden Methoden der Spiellogik mit den benötigten Übertragungsparametern aufgerufen. Bei einem ggf. stattzufindenden Seitenwechsel im Browser werden weiterhin entsprechende Übertragungsparameter dem request-Objekt hinzugefügt und die neue Seite unter Mitgabe des Objekts aufgerufen.

Zur weiteren Vereinfachung des Programmieraufwandes wird innerhalb der Servlet-Instanz nicht nur die Kommunikation mit dem Client koordiniert, sondern auch eine Instanz des Spiels erzeugt. Alle innerhalb der doPost()-Methode getätigten Aufrufe bezüglich der Spiellogik werden somit an die Instanz des Spiels übergeben. Auch wenn auf abstrakter Ebene das Servlet und die Spiellogik als zwei getrennte Elemente betrachtet werden, ist die Spielinstanz real gesehen eine Instanz innerhalb des Servlets.

9.5 JavaServer Page

(ES) Wie im Kapitel 9.4 auf Seite 49 beschriebenen werden in der doPost()-Methode neue Webseiten unter Mitgabe des request-Objekts aufgerufen. Da eine nur mit HTML erstellte Seite mit dem request-Objekt nicht umgehen kann, kommt zur Erstellung der Webseite JavaServer Page bzw. Expression Language zum Einsatz. Der Einsatz beider beschränkt sich jedoch innerhalb dieses Projektes auf die Angabe des Ziels des

HTML-Formulars sowie als Platzhalter für Variablen zur dynamischen Erstellung von Seiteninhalten.

Die Webseite wird wie gewohnt mit HTML erstellt, wobei die dynamisch zu erstellenden Inhalte im Quellcode durch Value Expressions (\${ Value }) ersetzt werden. Beim Aufruf der Seite innerhalb der doPost()-Methode werden die Value Expressions mit den gleichnamigen Parametern aus dem request-Objekt ersetzt und der so entstehende HTML-Code an den Browser des Clients ausgeliefert. In diesem Projekt werden hierdurch vor allem Textinhalte innerhalb von HTML-Elementen generiert bzw. bestimmte Werte dem Class-Attribut der HTML-Elemente hinzugefügt.

9.6 User Interface

(ES) Zur Erstellung des Userinterface kommen überwiegend die für Webanwendungen üblichen Technologien in Form von HTML5, CSS3 und JavaScript sowie aktuell gängige Frameworks zum Einsatz. Die logische Strukturierung der Seiten erfolgt mittels der Auszeichnungsprache HTML. Die Gestaltung des Userinterface in Bezug auf Layout, „Look and Feel“ erfolgt überwiegend mittels CSS3 sowie zur Reduzierung des Programmieraufwands mittels dem CSS-Framework Bootstrap. JavaScript sowie das JS-Framework jQuery werden zur logischen Validierung von Benutzereingaben sowie zur Umsetzung einiger kontextbezogener optischer Effekte bzw. Layoutumgestaltungen, welche sich mittels CSS schwierig oder nicht umsetzen lassen.

Die Aufteilung des Userinterface auf mehrere JSP-Dokumente ergibt sich in erster Linie aus an den Server zu schickenden Formularen und dem im Servlet definierten Requests. Für jedes zu versendende Formular wird ein eigenes JSP-Dokument verwendet. Die Formulare wiederum repräsentieren logische Spielabschnitte.

Die Startseite beinhaltet eine Auswahlmöglichkeit für die gewünschte Spieleranzahl sowie einen Button zum Start des Spiels. Mit Klick auf den Start-Button wird das Formular an das Servlet gesendet. Es wird das Spiel-Objekt mit der ausgewählten Anzahl an Spielern erzeugt und anschließend die erste Spielrunde mit dem ersten Spielzug des ersten Spielers begonnen.

In der ersten Spielrunde eines jeden Spielers wird dieser auf eine weitere Seite geführt. Hier hat der Spieler zu entscheiden auf welche Sparte bzw. welches Marktsegment sein Unternehmen zu Beginn ausgerichtet ist. Mit Versandt des Formulars an den Server erfolgt in der doPost()-Methode eine Abfrage welcher Button das Formular versendet hat, genauer gesagt, welcher Button zum aktuellen Zeitpunkt einen Value dem Http-Request mitgegeben hat. Dementsprechend erfolgt die Ausführung der entsprechenden Methoden im Fachwerk zum Freischalten der ausgewählten Sparte beim jeweiligen Spieler und Erzeugung der ersten Uhr für das ausgewählte Marktsegment. Der letzte

Schritt in diesem Abschnitt ist dann die Weiterleitung des Spielers an die für ihn benötigten Auswahl- sowie Anzeigeelemente für eine Spielrunde.

Aufgrund des Quellcodeumfangs zur Darstellung der Eingabemaske, für die vom Spieler während eines Spielzugs zu tätigen Aktionen, bzw. zur Anzeige seiner aktuellen Spielinformationen, ist der entsprechende Quellcode auf mehrere Dateien aufgeteilt. Eine Datei beinhaltet die Navigationselemente sowie das „Grundgerüst“ der Seite. Die durch die Navigationselemente umschaltbare Spielelemente sind zur besseren Übersicht über den Quellcode in weitere Dateien ausgelagert. JSP bietet hier die Möglichkeit die ausgelagerten Dateien mittels `jsp:include` einzubinden. Während der serverseitigen Kompilierung werden aus den einzelnen Dokumenten ein einzelnes Dokument für den Client bzw. dessen Browser erstellt.

Mittels Buttons kann der Spieler wählen, welche Aktionen in der aktuellen Spielrunde durchgeführt werden sollen. Ein Script, mittels JavaScript umgesetzt, übergibt an das jeweilige versteckte Inputfeld einen entsprechenden Wert. Nach Tätigung aller während der aktuellen Spielrunde gewünschten Spielaktionen, kann der Spieler den Button „Runde beenden“ betätigen. Die in den versteckten Inputfeldern gesammelten Spielaktionen werden über das HTML-Formular in Form eines Http-Requests an die `doPost()`-Methode innerhalb des Servlet-Objekts weitergeleitet. Innerhalb der Methode wird geprüft welche Buttons angeklickt wurden und die äquivalenten Methoden im Fachkonzept ausgeführt sowie die Methode zum Wechsel des Spielers aufgerufen. Am Ende dieses Abschnittes wird eine neue Seite mit Anzeige der aktuellen Runde sowie des nächsten Spielers angezeigt. Dieser kann gemäß des Hot-Seat-Verfahrens den Platz am Bildschirm einnehmen und seine Runde beginnen. Zum Ende der letzten Spielrunde wird das letzte Dokument zur Darstellung des Gewinners, bzw. der Gewinnreihenfolge der Spieler angezeigt. Unterhalb der Gewinneranzeige gibt es die Möglichkeit ein neues Spiel zu starten.

Neben der Dokumentenstruktur ist die Verwendung von CSS-Regeln und CSS-Klassen ein weiteres wichtiges Element der Umsetzung des Userinterface. Die CSS-Regeln sind derart gestaltet, dass je nachdem was der Spieler freigeschalten bzw. nicht freigeschalten hat, eine entsprechende optische Darstellung erfolgt. Als Selektoren kommen hierbei überwiegend CSS-Klassen zum Einsatz. In den entsprechenden HTML-Elementen sind im Value-Bereich des `class`-Attributs Value Expressions eingebettet. Je nach freigeschalteten Spielelementen werden im request-Objekt entsprechende Werte hinterlegt. Beim Kompilieren der Webseite werden die Value Expressions ggf. durch die entsprechenden Werte im request-Objekt ersetzt und somit die definierte CSS-Klasse beim HTML-Element definiert. Im Spielerobjekt hinterlegte Werte wie zum Beispiel das aktuelle Kapital oder der Warenbestand der Uhren werden auf die gleiche Art und Weise der Webseite übergeben. Die Value Expressions sind dann jedoch so im Quellcode plaziert, dass diese als Text ausgegeben werden und nicht dem jeweiligen Element als `class`-Attribute mitgegeben werden.

```

1 ||<div id="watch0" class="card ${watch0}>
2 ||<h4>Modell 1</h4>
3 ||<p>Produktlinie: ${m0s}</p>
```

Listing 9.1: Geschriebener Quellcode mit JSP

```

1 ||<div id="watch0" class="card card-aktive">
2 ||<h4>Modell 1</h4>
3 ||<p>Produktlinie: Umwelt</p>
```

Listing 9.2: Kompilierter HTML-Code

9.7 Eingabeüberprüfung im UI durch JavaScript

(EA) Da der Java-Code nicht im im HTML-basierten UI abgebildet ist benötigt es auf UI-Ebene eine weitere Logik. Diese ist notwendig, um die vom Benutzer getätigten Eingaben zu überprüfen und somit keine ungültigen Werte an das Spiel zu übergeben.

Kontostand prüfen

```

1 ||function getMoney()
2 ||{
3 ||return parseFloat($('#money').text().replace(/[\.\,\u20ac]/g, ''));
```

Listing 9.3: Überprüfung Kontostand

(EA) Jedes mal wenn eine Aktion ausgeführt werden soll, die Geld kostet, wird zuerst der Kontostand überprüft bevor sie ausgeführt wird. Die Aktionen eines Buttons werden über das „onclick-Event“ abgefangen. Damit diese jedoch nicht immer ausführbar ist wird in die „onclick“ Eigenschaft des Buttons eine Bedingung eingefügt.

```

1 ||<li class="${c10c1} list-group-item">Holz
2 ||<span class="glyphicon glyphicon-ok"></span>
3 ||<a class="addBtn" onclick="if(getMoney() >= $(this).text().replace
   (/[\.\,\+\u20ac]/g, '') && !$(this).hasClass('d')){research(
   researchCaseOeko,1); change($(this)); }else if($(this).hasClass(
   'd')){research('researchCaseOeko',1); change($(this)); }else{
   notEnough()}>+ ${c1C0c1} &euro;</a>
```

4 || </11>

Listing 9.4: OnClick-Event mit Bedingung für Kontostandüberprüfung

(EA) In dieser Bedingung wird geprüft, ob die Kosten der Aktion durch den aktuellen Kontostand gedeckt sind. Ergibt diese Prüfung „true“, so wird die jeweilige Aktion ausgeführt, es erfolgt eine Meldung auf dem Bildschirm, der Button ändert seine Beschriftung und die jeweiligen Kosten werden direkt vom Konto abgezogen.

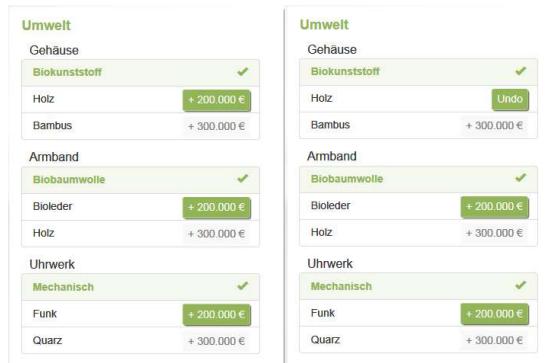


Abbildung 9.2: Button: do-undo

Beim erneuten Drücken auf den Button wird der Kontostand wieder um den vorher abgezogenen Betrag erhöht und die Kosten für die Aktion werden wieder auf dem Button ausgewiesen.

Wenn der Kontostand jedoch nicht ausreicht und der Button auch nichts bereits einmal gedrückt wurde, so wird die Funktion „notEnough()“ ausgeführt, welche eine Meldung auf dem Bildschirm ausgibt, dass das Geld nicht ausreicht.

```
1 || function notEnough()
2 | {
3 | alert('Das Kontoguthaben reicht nicht aus! ');
4 | }
```

Listing 9.5: Ausgabe bei zu geringem Kapital

Folglich wird die Aktion des Buttons auch nicht ausgeführt.

Eingabeüberprüfung in den Abteilungen

(EA) In den diversen Abteilungen müssen verschiedene Eingabemöglichkeiten berücksichtigt werden.

Forschung & Entwicklung

(EA) Hier ist es nur möglich Erweiterung für das jeweilige Segment freizuschalten, wenn das Kontoguthaben ausreicht. Des Weiteren ist es auch nur möglich ein neues Segment freizuschalten, wenn ebenfalls das Geld ausreicht.

Es wird aber auch ein bereits bestehender Bestand berücksichtigt, da Produktveränderungen auch an vorhandenen Uhren durchgeführt werden müssen. Dies verursacht somit einen zusätzlichen Aufwand.

Produktion

```

1  $('#output0').blur(function()
2  {
3    if($('#this').val())
4    {
5      if($('#this').hasClass('d'))
6      {
7        let undoProdCost = $('#this').attr('class').replace('d', '').replace(
8          'numInput', '');
9        let undoSum = getMoney() + undoProdCost * parseFloat($('#ekVal0').
10          text().replace(/[\.\u20ac]/g, '')); 
11        $('#money').text(undoSum.toLocaleString('de-DE', {
12          minimumFractionDigits: 2}));
13
14        if($('#this').val() > parseInt($('#productionLimit0').text().replace
15          (/[\.]/g, '')))
16        {
17          alert("Die Produktionskapazitaeten reichen nicht aus!\nDas Limit
18            betraegt: "+$('#productionLimit0').text());
19          $('#this').attr('class', 'numInput');
20          $('#this').val('');
21        }
22        else
23        {
24          if(getMoney() >= ($('#this').val() * parseFloat($('#ekVal0').text().
25            replace(/[\.\u20ac]/g, ''))))
26          {
27            let prodCost = $('#this').val() * parseFloat($('#ekVal0').text().replace
28              (/[\.\u20ac]/g, '')); 
29            let difference = getMoney() - prodCost;
30            $('#money').text(difference.toLocaleString('de-DE', {
31              minimumFractionDigits: 2}));
32            $('#this').attr('class', 'numInput');
33            $('#this').addClass('d');
34            $('#this').addClass($('#this').val());
35          }
36        }
37      }
38    }
39  }
40)

```

```

28 } else
29 {
30     alert("Das Guthaben reicht nicht aus!");
31     $(this).val('');
32     $('#prodCost0').val('');
33 }
34 }
35 }
36 else
37 {
38 if($(this).val() > parseInt($('#productionLimit0').text().replace
39     (/[\.\]/g, '')))
40 {
41     alert("Die Produktionskapazitaeten reichen nicht aus!\nDas Limit
42         betraegt: "+$('#productionLimit0').text());
43     $(this).val('');
44     $('#prodCost0').val('');
45 }
46 else
47 {
48 if(getMoney() >= ($(this).val() * parseFloat($('#ekVal0').text().replace
49     (/[\.\u20ac]/g, '')))
50 {
51     let prodCost = $(this).val() * parseFloat($('#ekVal0').text().replace
52         (/[\.\u20ac]/g, ''));
53     let difference = getMoney() - prodCost;
54     $('#money').text(difference.toLocaleString('de-DE',{
55         minimumFractionDigits: 2}));
56     $(this).addClass('d');
57     $(this).addClass($(this).val());
58 }
59 }
60 }
61 }
62 }
63 else if(!$(this).val())
64 {
65 if($(this).hasClass('d'))
66 {
67     let undoProdCost = $(this).attr('class').replace('d', '').replace(
68         'numInput', '');
69     let undoSum = getMoney() + undoProdCost * parseFloat($('#ekVal0').
69     text().replace(/[\.\u20ac]/g, ''));
70     $('#money').text(undoSum.toLocaleString('de-DE',{
71         minimumFractionDigits: 2}));

```

```

70 || $(this).attr('class', 'numInput');
71 }
72 }
73 });

```

Listing 9.6: Eingabeüberprüfung Produktion

(EA) In der Abteilung Produktion lässt sich die Menge der zu produzierenden Uhren angeben. Die Menge ist jedoch durch die Produktionskapazität begrenzt. Insofern wird auf Eingaben, welche die Kapazität übersteigen durch eine Meldung hingewiesen und das Eingabefeld wird geleert.

Wenn die Produktionsmenge jedoch im Rahmen des Möglichen liegt, so wird überprüft ob das Guthaben zu dem Zeitpunkt für die geplante Menge ausreicht. Ansonsten erfolgt die Meldung, dass dies nicht der Fall ist.

Dadurch dass das Geld sofort nach der Eingabe abgezogen wird muss berücksichtigt werden, dass Eingaben wieder rückgängig gemacht werden können. Insofern wird erkannt dass bereits eine Eingabe erfolgt ist und bei Änderung werden die vorher abgezogenen Beträge erst wieder auf das Konto hinzugefügt um danach die neuen Produktionskosten abziehen.

Einkauf

(EA) Beim Einkauf gibt es die Möglichkeit Produktressourcen durch Verbesserungen billiger zu beziehen. Hierbei erfolgt auch die Prüfung ob das Guthaben ausreicht.

Vertrieb

(EA) Im Vertrieb wird lediglich überprüft der Verkaufspreis mit der geplanten Absatzmenge multipliziert. Wobei auch die Absatzmenge durch den Vorrat begrenzt ist.

```

1  $('#quantitySupplied0').blur(function()
2 {
3   if( !$(this).val() == '' )
4   {
5     if(!$('#offerPrice0').val() == '')
6     {
7       if($(this).val() > parseInt($('#stock0').text().replace(/[\.\,]/g, '')))
8       {
9         alert('Der Vorrat reicht nicht aus! \nVorrat: '+$('#stock0').text());
10        $(this).val('');
11      }
12      else if($(this).val() > 0 && $(this).val() <= parseInt($('#stock0').
13          text().replace(/[\.\,]/g, '')));

```

```
13 || {
14   alert($(this).val().toLocaleString('de-DE',{minimumFractionDigits:
15     2}));
16   alert($('#offerPrice0').val());
17   let verkauf = parseFloat($(this).val().toLocaleString('de-DE',{
18     minimumFractionDigits: 2})) * parseFloat($('#offerPrice0').val());
19   $('#verkauf0').text(parseFloat(verkauf)+ '\u20ac');
20 }
21 }
22 else
23 {
24   alert('Sie m\u00fcssen zuerst einen Verkaufspreis eingeben!');
25   $(this).val('');
26 }
```

Listing 9.7: Eingabe\u00fcberpr\u00fcfung Vetrieb

10 Entwicklungsumgebung

10.1 Eclipse JAVA EE IDE

(NF) Eclipse ist eine Programmierumgebung als Entwicklungswerkzeug von Software und Programmen aller Art. Es gibt eine Vielzahl von kommerziellen und kostenlosen Erweiterungen. Unter Eclipse ist es möglich nahezu jede Programmiersprache zu entwickeln. Für das Projekt wurde lediglich die Programmiersprache Java für die Logik der Simulation verwendet. Als integrierte Entwicklungsumgebung umfasst Eclipse nicht nur einen Texteditor zum erstellen des Quellcodes, sondern viele weitere für die Softwareentwicklung benötigten oder hilfreichen Programme bzw. Funktionen. Diese lassen sich dank des modularen Aufbaus von Eclipse in beliebiger Kombination hinzufügen, oder man verwendet eine der für bestimmte Anwendungsfälle bereits vorkonfigurierten Versionen. Aufgrund der Verwendung einer Server-Client-Architektur unter Verwendung von Java Servlets, bietet es sich an, die entsprechende Version Eclipse JAVA EE IDE zu verwenden. Diese Version beinhaltet gleich die benötigten Komponenten, um den Tomcat-Server in die IDE einbinden zu können. Das Testen einzelner Funktionen kann somit innerhalb der Entwicklungsumgebung erfolgen und eine separate Verwaltung eines Webservers zu Testzwecken ist somit nicht erforderlich.

10.2 Git

(NF) Bei Git handelt es sich um eine freie Software zur Versionsverwaltung. Git ist Open Source und lediglich per Kommandozeile verfügbar. Dennoch gibt es bereits unzählige grafische UIs, die Git vereinfachen und seine Befehle grafisch darstellen (beispielhaft dafür: Sourcetree, GitHub Desktop oder TortoiseGit). Git wird auch von zahlreichen Großprojekten verwendet, darunter Android, LibreOffice oder jQuery.

10.3 GitHub

(NF) GitHub ist der webbasierte Online-Dienst, welcher Entwicklungsprojekte auf einem Server bereitstellt und somit Filehosting betreibt. Grundsätzlich ist es nicht

anderes als Git, eben nur nicht lokal, zudem bringt GitHub seinen eigenen grafischen UI Client namens GitHub Desktop mit. Ein grafischer Client wurde in diesem Projekt nicht mitgliederübergreifend eingesetzt. Weiterhin bieten viele Entwicklungsumgebungen wie auch Eclipse eine integrierte Möglichkeit auf die gehosteten Softwarerepositories zuzugreifen. Auch diese Lösung kam im Team zum Teil in Einsatz.

10.4 LaTeX

(NF) Für die Erstellung der Dokumentation wurde das Softwarepaket LaTeX genutzt. Als Editor wurden überwiegend TeXstudio oder der zur Softwareentwicklung gedachte Texteditor Atom eingesetzt.

LaTeX ist ein Softwarepaket, welches die Benutzung des Textsatzsystems TeX mit Hilfe von Makros vereinfacht. LaTeX ist die populärste Methode TeX zu verwenden. TeXstudio ist ein plattformunabhängiger Editor zur Erstellung von genannten LaTeX Dokumenten.

Beide sind Open-Source und kostenlos. Neben TeXstudio gibt es noch eine Vielzahl an weiteren Editoren, speziell für LaTeX Dokumente.

11 Qualitätssicherung mittels JUnit-Tests

(RH,LK) Zunächst wurde die Funktionalität der Methoden der Unternehmenssimulation anhand einfacher Konsolenausgaben durch die Entwickler unseres Teams getestet. Diese Art der Tests kann als Entwicklertest eingestuft werden. Die Entwicklertests werden in der Regel vom Entwickler selbst durchgeführt und dienen zur Überprüfung von Zwischenergebnissen und einzelnen Codezeilen. In dem untenstehendem Beispiel für einen Entwicklertest ist zu sehen, wie die Methoden `getAktuellerSpieler()` und `naechsterSpieler()` durch einfache Konsolenausgaben überprüft wurden.

```
1 // Nächster Spieler ist dran
2 System.out.println("Spieler " + spiel.getAktuellerSpieler() +
3     " am Zug");
4 System.out.println("Nächster Spieler!");
5spiel.naechsterSpieler()
6 System.out.println("Spieler " + spiel.getAktuellerSpieler() +
7     " am Zug")    System.out.println();
```

Listing 11.1: Einfache Überprüfung durch Konsolenausgabe

Nachdem die Funktionalität der Methoden überprüft und gegebenenfalls angepasst wurde, konnte mit dem Schreiben von JUnit-Tests begonnen werden. Unit-Tests, auch Modultests genannt, überprüfen einzelne isolierte Komponenten wie beispielsweise Methoden. Dadurch ist es möglich, Fehler schnell und einfach zu erkennen. Eine Änderung muss oft nur an einer Stelle vorgenommen werden, da die Methoden unabhängig von einander sein sollen. Da in unserem Projekt viele Methoden aufeinander aufbauen, war es nicht möglich, alle Methoden isoliert zu testen. Es existiert jedoch immer ein Test, der zunächst die Grundlage testet, zum Beispiel einen Spieler zu erstellen. Diese Methode wird dann in nachfolgenden Tests erneut aufgerufen, um eine logische Einheit abilden zu können. Das Testen einer sinnvollen Einheit ist uns wichtiger, als ein isoliertes Detail zu testen, welches aber in dieser Form kein Mehrwert bringt.

```
1 public void erstelleSpielerTest() {
2     Spielbrett spielbrett = new Spielbrett(10, 10000,
3         0.1);
4     Unternehmen[] spieler;
```

```
5     spielbrett.erstelleSpieler(2);
6     spieler = spielbrett.getSpieler();
7
8     assertEquals("Spieler wurde nicht korrekt erstellt",
9                  2, spieler.length);
}
```

Listing 11.2: Test einer Spielererzeugung

In dem obigen Beispiel ist ein JUnit-Test unserer Unternehmenssimulation zu sehen. Dieser Test überprüft die Erstellung der Spieler.

Um einen Spielablauf simulieren zu können, wurde ein Szenariotest geschrieben. Anders als beim Unit-Test prüft dieser das Zusammenspiel der einzelnen Methoden, wodurch ein Programmablauf nachgeahmt werden kann. Der Szenariotest überprüft somit auch die Eingaben, die ein Nutzer auf der Benutzeroberfläche vornehmen kann. Der Szenariotest besteht aus drei Runden, damit ausreichend Funktionalität abgedeckt werden kann.

In der ersten Runde werden drei Spieler in den jeweils drei unterschiedlichen Marktsegmenten erstellt. Jeder Spieler erforscht in seinem Segment eine Uhr, die dann produziert wird.

In der zweiten Runde können die produzierten Uhren am Markt angeboten werden. Jeder Spieler entwickelt zusätzlich in seinem Segment einen Bestandteil der Uhr (Armband, Gehäuse und Uhrwerk). Des Weiteren wird der Einkauf erweitert, wodurch die Selbstkosten gesenkt werden können. Anschließend wird nochmals produziert. In der dritten Runde werden erneut die produzierten Uhren angeboten. Die Produktion wird erweitert, wodurch das Produktionslimit gesteigert wird und somit mehr Uhren produziert werden können. Die letzten Funktionen, die der Szenariotest abbildet, sind die beiden Marketingstrategien. Diese setzen sich aus dem Uhrenmarketing, welches ein einzelnes Uhrenmodell bewirbt und dem Unternehmensmarketing, welches das Unternehmen als Ganzes betrifft, zusammen. Abgeschlossen wird der Test durch die Ermittlung des Siegers mittels einer assertEquals-Abfrage. Die genaue Platzierung der restlichen Spieler bleibt hierbei jedoch unberücksichtigt, da die Methode, welche die Platzierung der Gewinner ermittelt, privat ist, damit die Gewinnermittlung nicht von außen beeinflussbar ist.

Dem Screenshot ist zu entnehmen, dass der Szenariotest „Spielablauftest“ unsere Unternehmenssimulation zu 42,7% abdeckt.

Die JUnit-Tests unserer Unternehmenssimulation wurden in drei Testklassen aufgeteilt, um die Tests übersichtlicher zu gestalten. Somit testest z.B. der Markttest Methoden des Markts, welche jedoch teilweise Methoden des Unternehmens voraussetzen. Daher wurden manche Methoden sowohl im Unternehmenstest als auch im Markttest

Element	Coverage	Covered Instru...	Missed Instructio...	Total Instructio...
TestDynWebApp	42,7 %	3.813	5.125	8.938
src	42,7 %	3.813	5.125	8.938
tests	16,3 %	238	1.226	1.464
UnternehmenTest.java	0,0 %	0	989	989
Spielablauftest.java	100,0 %	238	0	238
Markttest.java	0,0 %	0	237	237
servlets	0,0 %	0	3.192	3.192
func	83,5 %	3.575	707	4.282

Abbildung 11.1: Coverage des Szenariotests

durchgeführt. Die drei Testklassen erreichen zusammen eine Codecoverage von 61,1%. In dem Screenshot ist zu erkennen, dass alle Klassen des Pakets „func“ zu mindestens 80% getestet wurden. Die Codecoverage wurde durch das Auslassen des Tests für das Servlet verringert.

Element	Coverage	Covered Instru...	Missed Instructio...	Total Instructio...
TestDynWebApp	61,1 %	5.462	3.476	8.938
src	61,1 %	5.462	3.476	8.938
tests	100,0 %	1.464	0	1.464
UnternehmenTest.java	100,0 %	989	0	989
Spielablauftest.java	100,0 %	238	0	238
Markttest.java	100,0 %	237	0	237
servlets	0,0 %	0	3.192	3.192
func	93,4 %	3.998	284	4.282
Unternehmen.java	90,2 %	1.671	182	1.853
Uhrmodell.java	95,7 %	66	3	69
Teilmarkt.java	100,0 %	171	0	171
Spielbrett.java	85,9 %	183	30	213
PremiumUhr.java	96,7 %	204	7	211
OekoUhr.java	96,7 %	204	7	211
Info.java	97,8 %	925	21	946
Gesamtmarkt.java	92,4 %	367	30	397
BilligUhr.java	98,1 %	207	4	211

Abbildung 11.2: Gesamte Coverage

Beim Ausgrenzen des Servlets von der Codecoverage wird die Unternehmenssimulation zu 95,1% durch Tests abgedeckt.

Element	Coverage	Covered Instru...	Missed Instructio...	Total Instructio...
TestDynWebApp	95,1 %	5.462	284	5.746
src	95,1 %	5.462	284	5.746
tests	100,0 %	1.464	0	1.464
func	93,4 %	3.998	284	4.282

Abbildung 11.3: Gesamte Coverage (ohne Servlets)

12 Projektorganisation

12.1 Aufgabenverteilung

(NF) Wie bereits in 1.2 erläutert wurde, war das Projektteam sehr gut ausbalanciert und brachte ein breites Spektrum an technischen und wirtschaftlichen Verständnis mit. Um die konkrete Aufgabenteilung darzustellen, sind nachfolgende alle Aufgaben und deren Bearbeiter beschrieben.

1. Coding
 - Miriam Wolf, Erik Schmidt, Ewald Anselm
2. JUnit-Tests: Erstellung, Realisierung und Implementierung
 - Rebekka Henn, Luisa Karl
3. Coding UI: Erstellung, Realisierung und Implementierung
 - Ewald Anselm, Erik Schmidt
4. Markt Algorithmen: Erstellung, Realisierung und Implementierung
 - Tilman Heß
5. Dokumentation und Präsentation
 - Nico Feil

12.2 Meilensteine



Abbildung 12.1: Meilenstein-Diagramm

13 Verbesserungs-/Erweiterungsmöglichkeiten

(NF) Während der Durchführungsphase zu diesem Projekt sind einige Verbesserungsmöglichkeiten entstanden, welche aufgrund der Zeit aber nicht mehr realisiert werden konnten. Nachfolgenden werden alle Verbesserungsmöglichkeiten aufgezählt und eine kurze Begründung abgegeben.

1. Erweiterung des Unternehmens um einen HR-Bereich
 - Wir haben uns schon recht früh gegen einen HR-Bereich entschieden. Gründe hierfür sind zum einen der geringe Bezug zu unserer eigentlichen Aufgabe und zum anderen die mangelnden Funktionen, welche der Bereich für den Spieler mit sich bringen würde.
2. Kreditaufnahme/-tilgung im Einkauf-Bereich
 - Die Idee war schlichtweg zeitlich nicht mehr möglich.
3. Speicherung des Spielstands
 - Da wir uns zu Beginn für ein Hot-Seat Verfahren entschieden haben und das Spiel insgesamt eine kurze Spieldauer hat, sahen wir dies nicht als notwendig an.
4. Anfälligkeit/Defekte der Produktionsstraßen
 - Die Idee war schlichtweg zeitlich nicht mehr möglich.
5. Zufallsereignisse (Krisen, Defekte,...)
 - Die Idee war schlichtweg zeitlich nicht mehr möglich.
6. Statistik Chart (Diagramm)
 - Die Idee war schlichtweg zeitlich nicht mehr möglich.
7. Marketing Chart (Diagramm)
 - Die Idee war schlichtweg zeitlich nicht mehr möglich.
8. Runden-Timer

- Die Idee war schlichtweg zeitlich nicht mehr möglich.

14 Fazit/Ausblick

(NF) Grundsätzlich ist die Aufgabe eine nette undfordernde Abwechslung zum normalen DHBW-Alltag. Das Planen und Entwickeln der verschiedenen Bestandteile hat Spaß gemacht und jeden in seiner Entwicklung und auf seine Weise weitergebracht. Wir konnten uns recht schnell auf unsere Industrieinigen und hatten so mehr Zeit für die Planung und Entwicklung. Die meiste Zeit wurde für die Konzeption und Entwicklung der einzelnen Bestandteile verwendet. Es gab einige Diskussionen rund um die einzelnen Unternehmensabteilungen und deren Methoden. Grundsätzlich kamen wir aber immer auf einen Nenner im Sinne der Aufgabe.

Negativ war leider die kurze Zeit, die wir insgesamt hatten. Einige Ideen konnten so nur teilweise oder gar nicht umgesetzt werden. Auch das Schreiben einer recht ausführlichen Dokumentation hat uns viel Zeit gekostet. Aus unserer Sicht ist eine solch ausführliche Dokumentation einfach nicht relevant, zielführend und schmälert zumal noch die Arbeitszeit, die eigentlich in die Realisierung investiert werden könnte. Zudem stehen auch noch die Klausuren für das Semester an.

Zusammenfassend kann man sagen, dass die Fallstudie ein guter Ansatz für einen realen Bezug ist, aber der enge Zeitraum einiges verhindert und somit rückblickend eher negativ bewertet werden muss.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich:

- dass ich die vorliegende Arbeit mit dem Titel *Watch Tycoon 2017* selbstständig verfasst und
- keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.
- Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum

Luisa Karl

Ort, Datum

Rebekka Henn

Ort, Datum

Miriam Wolf

Ort, Datum

Erik Schmitt

Ort, Datum

Tillmann Heß

Ort, Datum

Ewald Anselm

Ort, Datum

Nico Feil