

Software Projekt 2

Dynamische Systeme

Projektarbeit

Zürcher Hochschule für Angewandte Wissenschaften

Roger Knecht, David Elsener

Klasse: 4Ib, Dozenten: Syrus Mozafar, Albert Heuberger

07 Mai 2013

Inhalt

1.	Einleitung	4
1.1.	Thematik	4
1.2.	Ziel der Arbeit	4
1.3.	Realisierungspunkte	4
2.	Populationsdynamik	6
2.1.	Was ist Populationsdynamik?	6
2.2.	Hefe-Zucker-System	7
2.3.	Räuber-Beute-System	8
2.4.	Elektromagnetische Schwingkreise	9
3.	Projektplanung	10
3.1.	Agile Softwareentwicklung	10
3.1.1.	Iterationsplanung	10
3.1.2.	Burndown-Charts	10
3.1.3.	Taskliste	11
3.2.	Iteration 1	12
3.2.1.	Geplante Tasks	12
3.2.2.	Burndown-Chart	12
3.2.3.	Retrospektive	13
3.3.	Iteration 2	14
3.3.1.	Geplante Tasks	14
3.3.2.	Burndown-Chart	14
3.3.3.	Retrospektive	14
3.4.	Iteration 3	16
3.4.1.	Geplante Tasks	16
3.4.2.	Burndown-Chart	16
3.4.3.	Retrospektive	16
4.	Realisierung	18
4.1.	Anforderungen	18
4.1.1.	Eingabe von DS Konfigurationen	18
4.1.2.	Laden von vordefinierten DS Konfigurationen	18
4.1.3.	Speichern und Laden von eigenen DS Konfigurationen	18
4.1.4.	Grafische Darstellung der Simulation	18
4.1.5.	Regelbarkeit der Populationswerte während der Simulation	18
4.2.	Tools, Technologien und Frameworks	18

4.2.1.	Java 1.7 und Swing	18
4.2.2.	Eclipse IDE	18
4.2.3.	Git und Github	19
4.2.4.	Maven 3.0.4	19
4.2.5.	JUnit 4.8.1	19
4.2.6.	JFreeChart 1.0.13	19
4.2.7.	Apache Common JEXL	19
4.3.	Projektstruktur	21
4.4.	Architektur / Klassendiagramme	22
4.4.1.	Simulations-Modell	22
4.4.2.	GUI-Klassen	23
4.5.	Expression Language	24
4.5.1.	JEXL – Java Expression Language	24
4.5.2.	Hilfsklasse: ch.zhaw.dynsys.el.utils.ExpressionUtil	24
4.6.	Grafische Benutzeroberfläche	26
4.7.	Darstellung der Simulation	26
4.7.1.	X-Achse	26
4.7.2.	Y-Achse	26
4.7.3.	Legende	27
4.7.4.	Konfiguration des Werte-Bereichs	27
4.7.5.	Konfiguration des Dynamischen Systems	27
4.7.6.	Vordefinierte Konfigurationen Laden	28
4.7.7.	Ganze Simulation betrachten	28
5.	Schlusswort	29
6.	Quellenverzeichnis	30
6.1.	Projektstruktur auf GitHub	30
6.2.	Theorie: Dynamische Systeme und Populationsdynamik	30

1. Einleitung

1.1. Thematik

Das Themenfeld *Dynamische Systeme (DS)* ist sehr gross und umfasst verschiedene Arten von DS. Das Grundprinzip ist jedoch immer gleich: Ein DS ist ein mathematisches Modell eines zeitabhängigen Prozesses. Es kann sich dabei um mathematische, physikalische oder biologische Prozesse handeln. Das DS erhält einen Startzeitpunkt, Startzustände- und Parameter und verändert seinen Zustand entweder pro festgelegte gleichbleibende Zeiteinheit (diskret) oder in unendlich kleinen Zeitabständen (kontinuierlich). Die Zustandsänderungen erfolgen aufgrund der Änderungen der Grössen (z.B. Population, Ressourcen) deren Änderungsverhalten wiederum durch Differentialgleichungen bestimmt wird. Die Differentialgleichungen entstehen durch die Definition des simulierten Prozesses als mathematisches Modell. Möchte man nämlich Naturgesetze als Funktionen mit Variablen definieren, entstehen sehr häufig Funktionen, welche die eigene Ableitung verwenden, und damit also Differentialgleichungen.

1.2. Ziel der Arbeit

Für unsere Projektarbeit mussten wir uns zunächst auf ein bestimmtes Gebiet einschränken. Da es ein geläufiges Beispiel, vom Verständnis her einfacher und sich gut für eine Visualisierung in einer Software eignet, haben wir uns für die Umsetzung eines biologischen Prozesses, der Populationsdynamik, entschieden als DS. Dabei wird von einer bestimmten Art eine Startpopulation genommen und mittels definierten Funktionen deren Entwicklung über die Zeit anhand des Umfelds und der Ressourcen simuliert.

Wir wollen dabei eine möglichst generische Applikation entwickeln. Das Basisprogramm soll es ermöglichen, beliebige DS anhand von Funktionen und Variablen zu definieren und in das Programm einzugeben. Die eingegebenen DS werden simuliert und der Prozess anhand mehrerer Grafen visualisiert. Um aber ein konkretes Beispiel vor Auge zu haben, werden wir ein konkretes DS für die Populationsdynamik von Hefe (Schlauchpilz, verwendet in Brot- und Bierprodukten) definieren und als Muster und Test DS in der Applikation verwenden.

1.3. Realisierungspunkte

Das Software-Projekt beinhaltet folgende Punkte, die realisiert werden müssen:

- Planung (und Schätzung) der User Stories und Tasks
- Definition des DS für die Populationsdynamik von Hefe
- Entwicklung eines Basisprogramms
 - Fütterung des Programms mit (möglichst) beliebigen DS
 - Generische Visualisierung dieser DS

- Programmierung des konkreten DS (Hefe)
- Dokumentation / Vorbereitung der Präsentation

2. Populationsdynamik

2.1. Was ist Populationsdynamik?

Populationsdynamik beschreibt ein System, welches in Abhängigkeit der Zeit die Wechselwirkungen von Prozessen beschreibt. Primär stehen hier biologische Abläufe im Zentrum. Mittels Differentialgleichungen werden die Umgebungseinflüsse von Kulturen wie auch von intra- und extrasystemischen Faktoren beschrieben. Die Komplexität eines solchen Systems nimmt mit jedem weiteren Faktor exponentiell zu. Deshalb werden solche Prozesse über Computersimulation berechnet um potenzielle Optimierungen in biologischen Abläufen zu finden.

Im Vordergrund steht am Anfang die Analyse des Prozesses. Dabei werden die empirischen Beobachtungen mathematisch formuliert. Im weiteren Verlauf wird die Entwicklung der Simulation und des natürlichen Prozesses verglichen. Falls die Simulation stark von der Realität abweicht, müssen die Faktoren justiert werden oder im schlimmsten Falle muss eine neue Formulierung gesucht werden.

Hat sich ein System über längere Zeit bewährt, so kann man mit der Optimierung beginnen. Dabei werden Startwerte und extrinsische Einwirkungen variiert. Dabei erhält man Rückschlüsse auf Idealbedingungen der Kulturen. Die gewonnene Erkenntnis wird anschließend auf die Natur angewendet.

2.2. Hefe-Zucker-System

Das erste System simuliert das Wachstumsverhalten von Hefe. Die Hefe vermehrt sich wenn Zucker (oder andere Kohlenhydrate) in unmittelbarer Nähe ist. Bis zu einem gewissen Grad gilt: Je mehr Zucker, desto schneller vermehrt sich die Hefe. Allerdings ist die Hefe irgendwann gesättigt und kann pro Zeiteinheit nicht mehr Zucker aufnehmen. Der Zucker nimmt proportional zur Wachstumsrate der Hefe ab. Als dritte Komponente kommt die Temperatur hinzu. Die Hefe vermehrt sich mit sinkender Temperatur langsamer, bis sie schliesslich zum Stillstand kommt. Bei steigender Temperatur nimmt die Wachstumsrate entsprechend zu, bis zu einem gewissen Maximum und anschliessen nimmt diese wieder ab. Bei sehr hoher Temperatur sterben die Hefezellen.

Als Referenz für unsere Simulation wird die Kinetik nach Monod verwendet:

$$\begin{aligned}a &= 1 \\b &= 2 \\c &= -1 \\d &= -0.2\end{aligned}$$

$$\begin{aligned}H' &= a \cdot H \cdot \frac{Z}{b + Z} \\Z' &= \begin{cases} c \cdot H + d \cdot H', & Z > 0 \\ 0, & \text{sonst} \end{cases}\end{aligned}$$

Mit H und Z für die Population der Hefe bzw. des Zuckers und H' bzw. Z' für das Hefewachstum und die Zuckerabnahme.

2.3. Räuber-Beute-System

Eine etwas kompliziertere Simulation stellt das Räuber-Beute-Schema dar. Hierbei wird der Fortpflanzungserfolg eines Jäger bzw. Räuber und eines Opfer bzw. der Beute beobachtet. Zur Veranschaulichung wurde repräsentativ für den Räuber der Fuchs gewählt und anstelle des Opfers kommt der Hase ins Spiel. Hasen vermehren sich fortlaufend und ungebremst. Je mehr Hasen in der Simulation vorhanden sind umso grösser die Wachstumsrate. Der Fuchs ernährt sich von Hasen und je mehr Hasen es gibt desto besser kann er sich fortpflanzen. Zusätzlich werden noch die Karotten, wovon sich die Hasen ernähren, berücksichtigt. Mathematisch kann man dies folgendermassen definieren:

$$\begin{aligned}K_{Wachstum} &= 20, & K_H &= -0.2 \\H_{Wachstum} &= -1, & H_F &= -1, & H_K &= 1 \\F_{Wachstum} &= -100, & F_H &= 1\end{aligned}$$

$$\begin{aligned}K' &= K_{Wachstum} + K_H \cdot H \\H' &= H_{Wachstum} + H_F \cdot F + H_K \cdot K \\F' &= F_{Wachstum} + F_H \cdot H\end{aligned}$$

Mit K , H und F für die Population der Karotten, Hasen und Füchse und K' , H' und F' für die Karotten-, Hasen- und Fuchswachstumsrate.

2.4. Elektromagnetische Schwingkreise

Die Simulation kann neben biologischen Prozessen auch für viele weitere Aspekte gebraucht werden. Mit einem Schaltkreis (siehe Bild rechts) wird das Verhältnis von Ladung zum elektromagnetischen Feld beschrieben, ein sogenannter elektromagnetischer Schwingkreis. Der Schaltkreis besteht aus einer Batterie U , einem Widerstand R , einer Spule L und einem Kondensator.

Es gilt:

$$C = \frac{Q}{U}$$

$$U = L \cdot i'$$

$$U_R = R \cdot i$$

$$U_C = \frac{Q}{C}$$

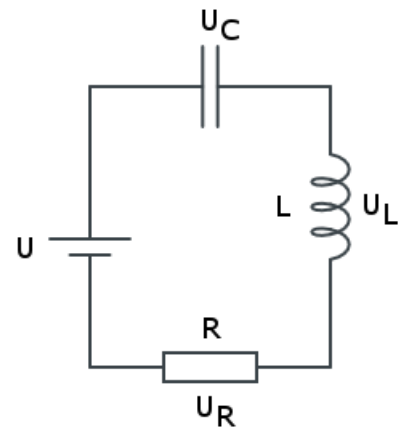
$$U + U_R + U_L + U_C = 0$$

$$U + R \cdot Q' + \frac{Q}{C} + L \cdot Q'' = 0$$

Daraus folgen die zwei Differentialgleichungen:

$$i' = Q'' = -\frac{1}{L} \cdot \left(U + R \cdot Q' + \frac{1}{2} \cdot Q \right)$$

$$Q' = i$$



3. Projektplanung

3.1. Agile Softwareentwicklung

Im Modul *Methoden der Programmierung* haben wir die aktuell populärste Vorgehensmethodik in der Softwareentwicklung behandelt – die Agile Softwareentwicklung, die dem sogenannten „*Agile Manifesto*“ folgt:

- Menschen und Interaktionen sind wichtiger als Prozesse und Werkzeuge.
- Funktionierende Software ist wichtiger als umfassende Dokumentation.
- Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlungen.
- Eingehen auf Veränderungen ist wichtiger als Festhalten an einem Plan.

3.1.1. Iterationsplanung

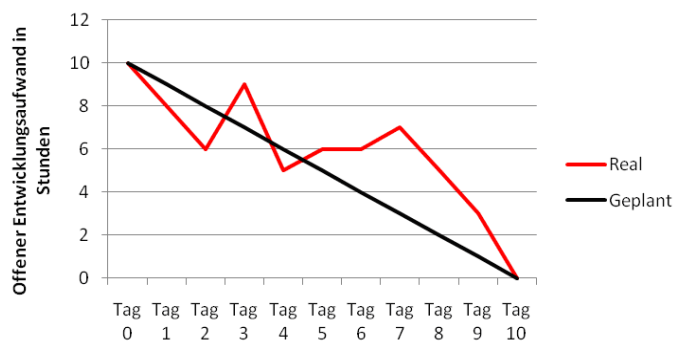
Um eine flexible Planung zu garantieren und so besser auf Unvorhergesehenes reagieren zu können, werden wir die gesamte Projektzeit in drei Iterationen aufteilen:

Nr.	Zeitraum	Geplanter Aufwand RK	Geplanter Aufwand DEL	Aufwand total
1	22.03 - 11.04	18.0	23.0	41
2	12.04 - 02.05	5.0	15.0	20
3	03.05 - 24.05	30.0	22.0	52
		53.0	60.0	113

3.1.2. Burndown-Charts

Um unsere Fortschritte stets mitverfolgen und überprüfen zu können, werden wir pro Iteration jeweils einen Burndown-Chart führen. Dieser zeigt jeweils über die Iteration hinweg welche Aufwände geleistet und welche noch verbleibend sind. Anhand der daraus entstehenden Kurven können wir unter Umständen auch Tendenzen feststellen und Gegenmassnahmen planen.

Ein exemplarisches Burndown-Chart (entspricht nicht unserem Verlauf):



3.1.3. Taskliste

Aus den Anforderungen, die wir zu Beginn definiert haben, ist eine schätzbare und gut portionierte Taskliste entstanden, anhand derer wir vorgehen werden. Es wird jeweils zu Beginn einer Iteration bestimmt, welche Tasks mit den zur Verfügung stehenden Ressourcen umgesetzt werden. So können wir auch kurzfristig – spätestens vor Start der nächsten Iteration – Unvorhergesehenes in die Taskliste aufnehmen.

Nachfolgend die Taskliste mit den jeweiligen Schätzungen und dem Total Aufwand:

Nr.	Task	Schätzung (h)	
1	Theorie: Einarbeitung in Thematik "Dynamische Systeme"	8.0	(4h pro Person)
2	Theorie: Aufarbeitung Thematik "Differential Gleichungen"	8.0	(4h pro Person)
3	Infrastruktur (Git Repo, ..)/Java-Projekt aufsetzen	1h	
4	Konzeptionell: Spezifizierung eines konkreten dynamischen Systems	6.0	(3h pro Person)
5	Konzeptionell: GUI Mockups (Anzeige der Grafen)	3.0	
6	Dokumentation: 0.5-1 Seite Projektbeschreibung (inkl. des DS)	2.0	
7	Konzeptionell: Planung der Software-Architektur für generisches Programm	2.0	
8	Implementierung der Logik zur Lösung von Differential Gleichungen	20.0	
9	Implementierung des Basisprogramms (ohne GUI)	10.0	
10	Evaluation einer Library für die Generierung der Grafen	2.0	
11	Basis-GUI (Mainframe, Menu, etc.) implementieren	10.0	
12	Dokumentation nachführen - Teil 1	4.0	(2h pro Person)
13	Einbinung der Grafen-Library und Implementierung von geeigneten Schnittstellen	10.0	
14	GUI und Backend zusammenführen	4.0	
15	Einspeisung des konkreten dynamischen Systems, erster Testlauf	5.0	
16	Dokumentation nachführen - Teil 2	3.0	(1.5h pro Person)
17	Präsentation vorbereiten	6.0	(3h pro Person)
18	Puffer / Weitere dynamische Systeme einspeisen	10.0	
Total		113.0	

3.2. Iteration 1

Zeitraum: 22.03 – 11.04

Ressourcen: 42 Stunden

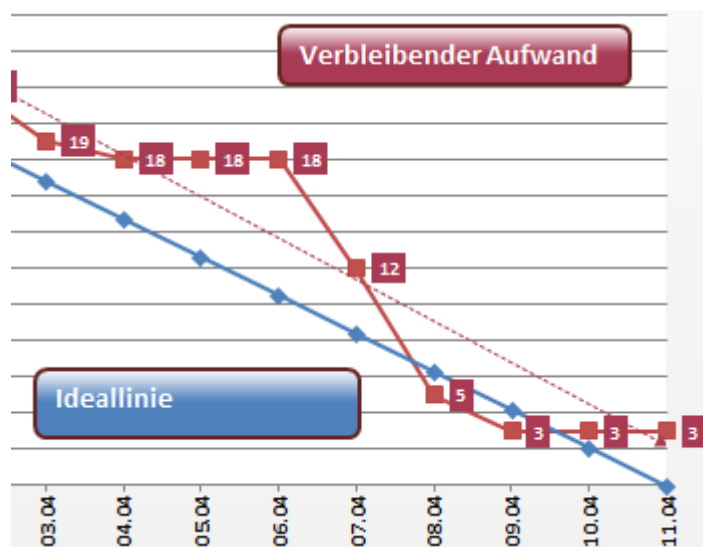
- Roger: 18 Stunden
- David: 24 Stunden

3.2.1. Geplante Tasks

Task	Aufwand	Verantwortlicher
Theorie: Einarbeitung in Thematik "Dynamische Systeme"	4.0	David
Theorie: Einarbeitung in Thematik "Dynamische Systeme"	4.0	Roger
Theorie: Aufarbeitung Thematik "Differential Gleichungen"	4.0	David
Theorie: Aufarbeitung Thematik "Differential Gleichungen"	4.0	Roger
Infrastruktur (Git Repo, ..)/Java-Projekt aufsetzen	1.0	David
Konzeptionell: Spezifizierung eines konkreten dynamischen Systems	3.0	Roger
Konzeptionell: Spezifizierung eines konkreten dynamischen Systems	3.0	David
Konzeptionell: GUI Mockups (Anzeige der Grafen)	3.0	Roger
Dokumentation: 0.5-1 Seite Projektbeschreibung (inkl. des DS)	2.0	David
Konzeptionell: Planung der Software-Architektur für generisches Programm zur Ein	2.0	Roger
Implementierung des Basisprogramms (ohne GUI)	10.0	David
Evaluation einer Library für die Generierung der Grafen	2.0	Roger

3.2.2. Burndown-Chart

Mit folgendem Bild endete die erste Iteration (Ausschnitt):



3.2.3. Retrospektive

Die Iteration endete mit einem Restaufwand von 3 Stunden. Bei einer Schätzung von Tasks von insgesamt 42 Stunden ist dies kein schlechtes Ergebnis. Der verbleibende Aufwand betraf den Task 9 (Implementierung des Basisprogramms), wobei dies aber nicht schlimm ist, die Arbeit hier kann gut auf ein paar Tasks der nächsten Iteration verteilt werden.

Wir sind beide zufrieden mit der Iteration. Der Start mit der ersten Sitzung mit Herrn Heuberger hat das Projekt konkret werden lassen und mit den ersten umgesetzten Tasks hat die Applikation zwar noch nicht gross Form angenommen, aber die Vorbereitung ist gelungen.

3.3. Iteration 2

Zeitraum: 12.04 – 02.05

Ressourcen: 48 Stunden

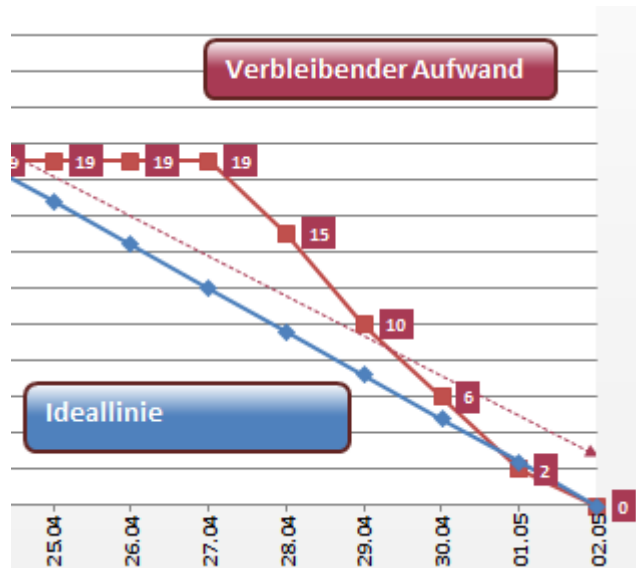
- Roger: 24 Stunden
- David: 24 Stunden

3.3.1. Geplante Tasks

Task	Aufwand	Verantwortlicher
Implementierung der Logik zur Lösung von Differential Gleichungen	20.0	David
Basis-GUI (Mainframe, Menu, etc.) implementieren	10.0	Roger
Dokumentation nachführen - Teil 1	4.0	David
Einbinung der Grafen-Library und Implementierung von geeigneten Schnittstellen z	10.0	Roger
GUI und Backend zusammenführen	4.0	Roger

3.3.2. Burndown-Chart

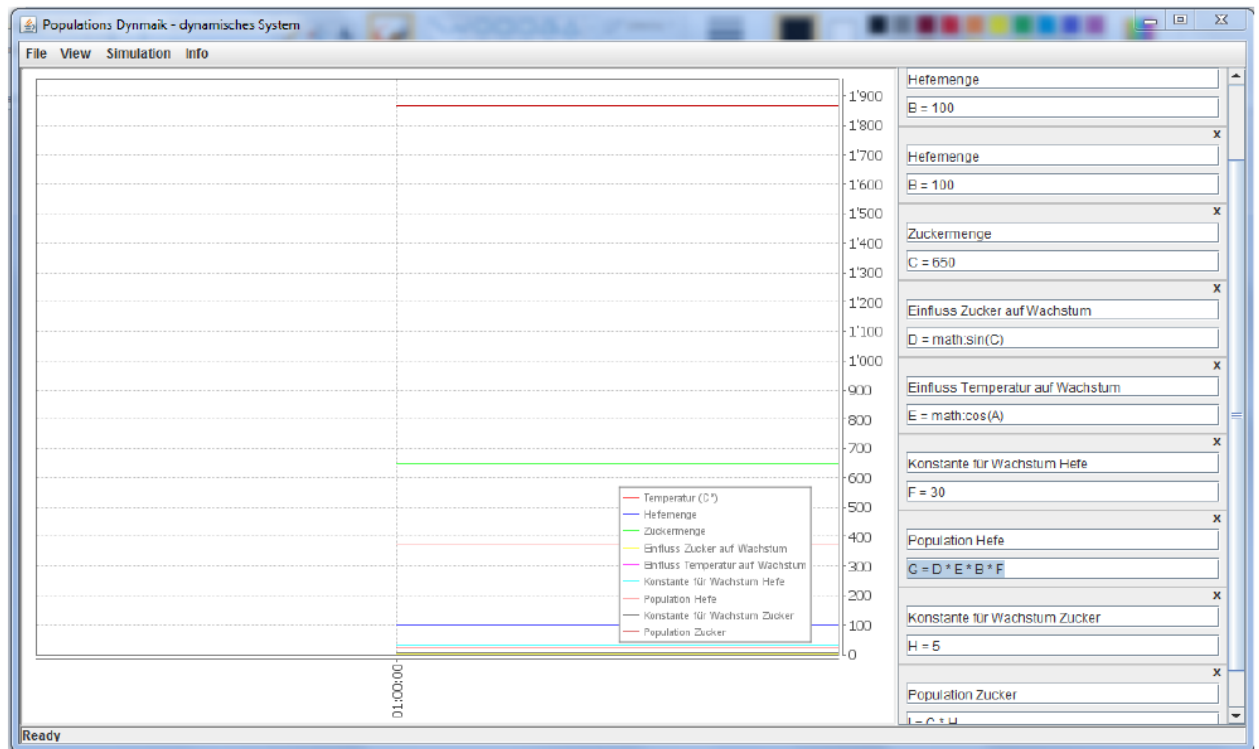
Mit folgendem Bild endete die zweite Iteration (Ausschnitt):



3.3.3. Retrospektive

In dieser Iteration konnten wir alle geplanten Tasks umsetzen. Wir haben aber etwas mehr Zeit in die Implementierung als in die Dokumentation investiert. Das Basis GUI steht bereits und auch können Funktionen Dank der eingebundenen Expression-Language Bibliothek relativ generisch eingegeben werden. Jetzt müssen das GUI und das Backend mit den interpretierten Funktionen noch miteinander verbunden werden um die Simulation eines dynamischen Systems durchspielen zu können.

Nachfolgend ein Screenshot der aktuellen Applikation:



3.4. Iteration 3

Zeitraum: 03.05 – 24.05

Ressourcen: 35 Stunden

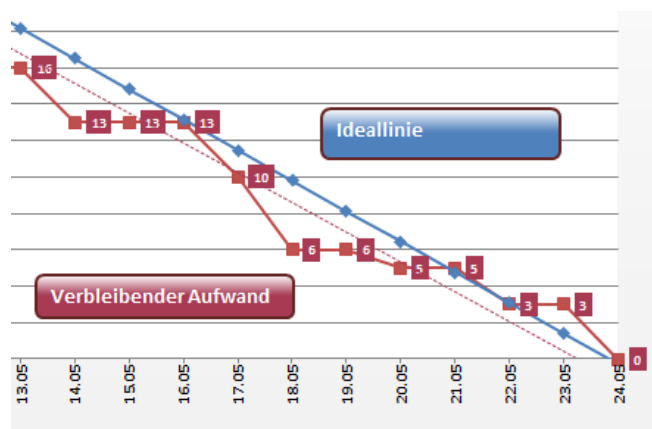
- Roger: 19 Stunden
- David: 16 Stunden

3.4.1. Geplante Tasks

Task	Aufwand	Verantwortlicher	Aufgewendete Zeit
Einspeisung des konkreten dynamischen Systems, erster	5.0	Roger	5
Dokumentation nachführen - Teil 2	6.0	Roger	6
Präsentation vorbereiten	6.0	David	6
Genauigkeit und Schnelligkeit der Berechnungen	3.0	Roger	3
Buttons zur Veränderung von Werten	1.0	Roger	1
Fuchsbeispiel implementieren	2.0	David	2
Hefebeispiel verbessern	2.0	Roger	2
Sinusbeispiel implementieren	2.0	Roger	2
Tests implementieren und Dokumentieren	2.0	David	2
Dokumentation nachführen - Teil 3	6.0	David	6

3.4.2. Burndown-Chart

Mit folgendem Bild endete die zweite Iteration (Ausschnitt):

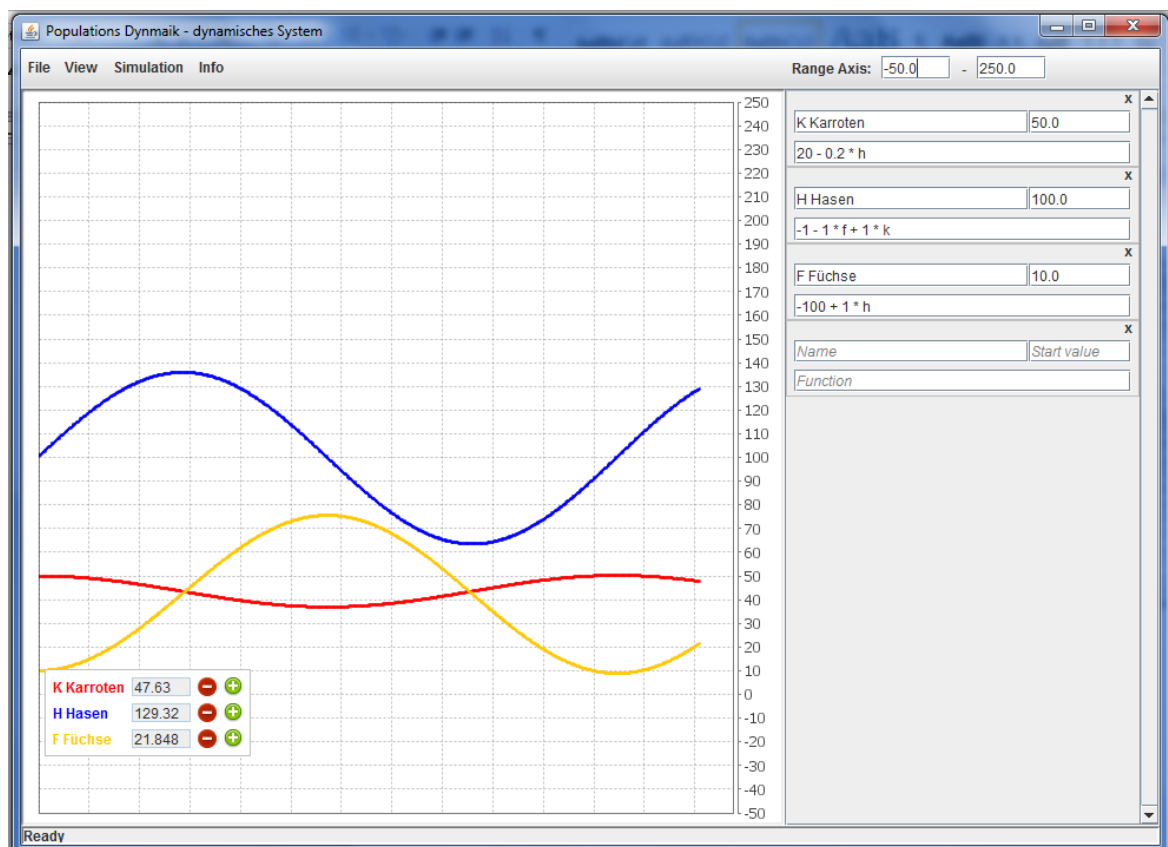


3.4.3. Retrospektive

Die Iteration 3 war mit 35 Stunden eigentlich mit sehr wenig Aufwand geplant. Alle geplanten Aufgaben konnten wir auch dementsprechend umsetzen. Jedoch hatten wir aufgrund der Besprechungen mit Herrn Heuberger und aufgrund von einigen kleinen

und grösseren Fehler in der Applikation doch mehr Programmieraufwand als erwartet. Kleine Änderungen an der Konfigurierbarkeit der Simulation benötigten teilweise doch grössere Anpassungen am Code. Inzwischen ist aber die Code-Basis gut geworden und die Applikation stabil. Auch konnten wir die Erweiterungsidee von Herrn Heuberger bezüglich der Regelbarkeit der aktuellen Populationswerte während der Simulation durch den Benutzer ohne Probleme miteinbauen. Mit diesem Stand können wir eine Präsentation unseres Themas und der Simulationen wagen.

Screenshot der Applikation nach dieser Iteration:



4. Realisierung

4.1. Anforderungen

Nachfolgend werden kurz die wichtigsten Anforderungen an die Applikation beschrieben.

4.1.1. Eingabe von DS Konfigurationen

Da ein DS mathematisch durch Funktionen definiert wird, muss der Benutzer die Möglichkeit haben, mittels einer der Mathematik ähnlichen Sprache solch ein DS konfigurieren zu können.

4.1.2. Laden von vordefinierten DS Konfigurationen

Wie bereits aus dem Projektbeschrieb zu entnehmen ist, verwenden wir das Hefe-Zucker-System als Beispiel-DS und später werden wir noch zwei weitere DS definieren. Diese DS sollen möglichst komfortable vom Benutzer geladen und die Simulation gestartet werden können.

4.1.3. Speichern und Laden von eigenen DS Konfigurationen

Hat der Benutzer Funktionen und Variablen eingegeben, muss er die Möglichkeit haben, seine Konfiguration zu speichern um sie später wiederverwenden zu können.

4.1.4. Grafische Darstellung der Simulation

Die Simulation soll möglichst einfach aber auch klar und übersichtlich dargestellt werden. Dies soll mittels Grafen realisiert werden, wobei es pro Kultur/Population eine Kurve im Grafen geben wird.

4.1.5. Regelbarkeit der Populationswerte während der Simulation

Um die Parameter einer Simulation gut zu wählen, erfordert es viele Versuche und Experimente. Deshalb kann es auch ganz interessant für den Benutzer sein, während einer Simulation die Möglichkeit zu haben, die aktuellen Populationswerte zu übersteuern und zu regeln.

4.2. Tools, Technologien und Frameworks

4.2.1. Java 1.7 und Swing

Um in kurzer Zeit eine Applikation mit grafischer Oberfläche zu entwickeln, eignet sich Java mit Swing als Framework für das GUI für uns sehr gut, da wir – zumindest was die Programmiersprache betrifft - schon einige Jahre Erfahrung haben.

4.2.2. Eclipse IDE

Wir benützen beide die Eclipse IDE als bewährte Entwicklungsumgebung für die Java Applikation.

4.2.3. Git und Github

Git wird an der ZHAW in Vorlesungen als Beispiel für ein Versionskontrollsystem verwendet. Mit Github erhalten wir gleichzeitig gratis ein Repository auf einem Server, weshalb sich also Git gut für unser Projekt eignet.

Das Repository des Projekts kann unter folgender URL erreicht werden:

<https://github.com/delsener/ch.zhaw.softwareprj2.git>

Wir verwenden die Versionskontrolle nicht nur für den Source Code, sondern auch für alle anderen Artefakte die aus der Projektplanung oder der Dokumentation entstehen.

4.2.4. Maven 3.0.4

Als Buildsystem verwenden wir auch den heutigen Standard – nämlich Maven. Die Identifikation unseres Maven Projektes sieht wie folgt aus:

```
<groupId>ch.zhaw.softwareprj2</groupId>
<artifactId>dynsys</artifactId>
<version>1</version>
```

4.2.5. JUnit 4.8.1

Das Standard Testing-Framework für Java ist JUnit. Wir verwenden dies ebenfalls und kombinieren es, falls es sich anbieten sollte, mit Mockito.

4.2.6. JFreeChart 1.0.13

Die Visualisierung der DS soll in der Applikation mittels Grafen realisiert werden. Die der Wahl einer Bibliothek, welche das Zeichnen dieser Grafen unterstützt, fiel auf JFreeChart, entwickelt von **Object Refinery Ltd.**

Folgende Punkte zeichnet die Library für uns aus:

- In Java geschrieben
- Open Source
- Unterstützt 2D und 3D Grafen
- Swing Integration
- Grafen können exportiert und in einem Bildformat (PNG, JPG, SVG, etc.) gespeichert werden

Anwendungsbeispiele können unter folgender URL gefunden werden:

<http://www.jfree.org/jfreechart/samples.html>

4.2.7. Apache Common JEXL

Die Applikation soll es erlauben, vom Benutzer mit Variablen / Funktionen / Konstanten gefüttert zu werden, welche zusammen ein dynamischen System bilden. Die Benutzereingaben müssen dabei also als mathematische Formeln interpretiert werden – dies ist keine triviale Sache. Für Java gibt es zwei bekannte Libraries welche eine

Sprache unterstützen, die der mathematischen Syntax mit Erweiterungen entspricht. Die Expression Language von Spring – vermutlich der Standard – und jene von Apache Commons. Da wir über Basiskenntnisse über die Commons Library (JEXL - Java Expression Language) verfügen, haben wir uns für sie entschieden.

Die Projektwebseite findet sich unter folgender URL:

<http://commons.apache.org/proper/commons-jexl/>


Nachfolgend ein Beispiel eines Inputs, welcher die vom Benutzer eingegebenen Funktionen beinhaltet, den wir mit JEXL interpretieren:

```
{
var t = -20.0
var t_diff = 0.0
var H = 100.0
var H_diff = 0.0
var Z = 650.0
var Z_diff = 0.0
}
return [
    new("java.lang.Double",5),
    new("java.lang.Double",math:max(-H, 0.0001*H*Z*(20-math:abs(10- t)))),
    new("java.lang.Double",math:min(0, -math:min(Z, H_diff)))
];
```


4.3. Projektstruktur

In unserem Java-Projekt können wir mittels Packages die verschiedenen Klassen in eine übersichtliche Struktur bringen. Das Projekt ist folgendermassen gegliedert:


- **GUI-Komponenten:**

- Package: ▶  `ch.zhaw.dynsys.gui`
- Beschreibung: Beinhaltet alle Klassen, welche zur Darstellung der grafischen Oberfläche gebraucht werden


- **Logik/Simulations-Komponenten:**

- Package: ▶  `ch.zhaw.dynsys.simulation`
- Beschreibung: Beinhaltet alle Klassen, welche für die Simulation benötigt werden. Natürlich gibt es hier Berührungspunkte zu GUI-Komponenten (Siehe Klassendiagramm).

- **Hilfsklassen für die Expression Language:**

- Package: ▶  `ch.zhaw.dynsys.el.utils`
- Beschreibung: Um mit der JEXL Library arbeiten zu können, brauchen wir Hilfsklassen (Utility/Helper). Diese befinden sich in diesem separaten Package.

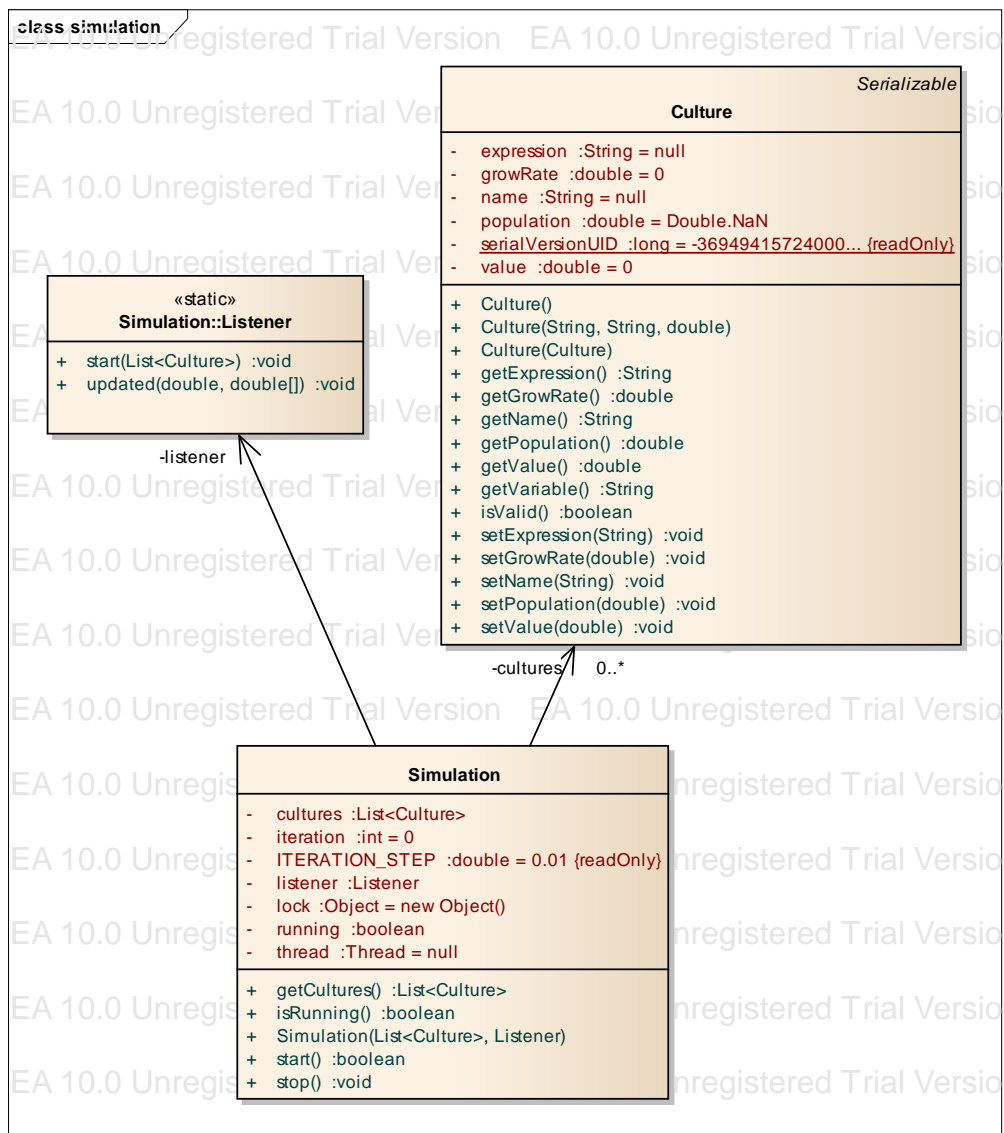
- **Persistenz-Klassen:**

- Package: ▶  `ch.zhaw.dynsys.persistence`
- Beschreibung: Alle Klassen, welche für die Clientseitige Persistierung (Serialisierung und Files) benötigt werden, befinden sich in diesem Package.

4.4. Architektur / Klassendiagramme

4.4.1. Simulations-Modell

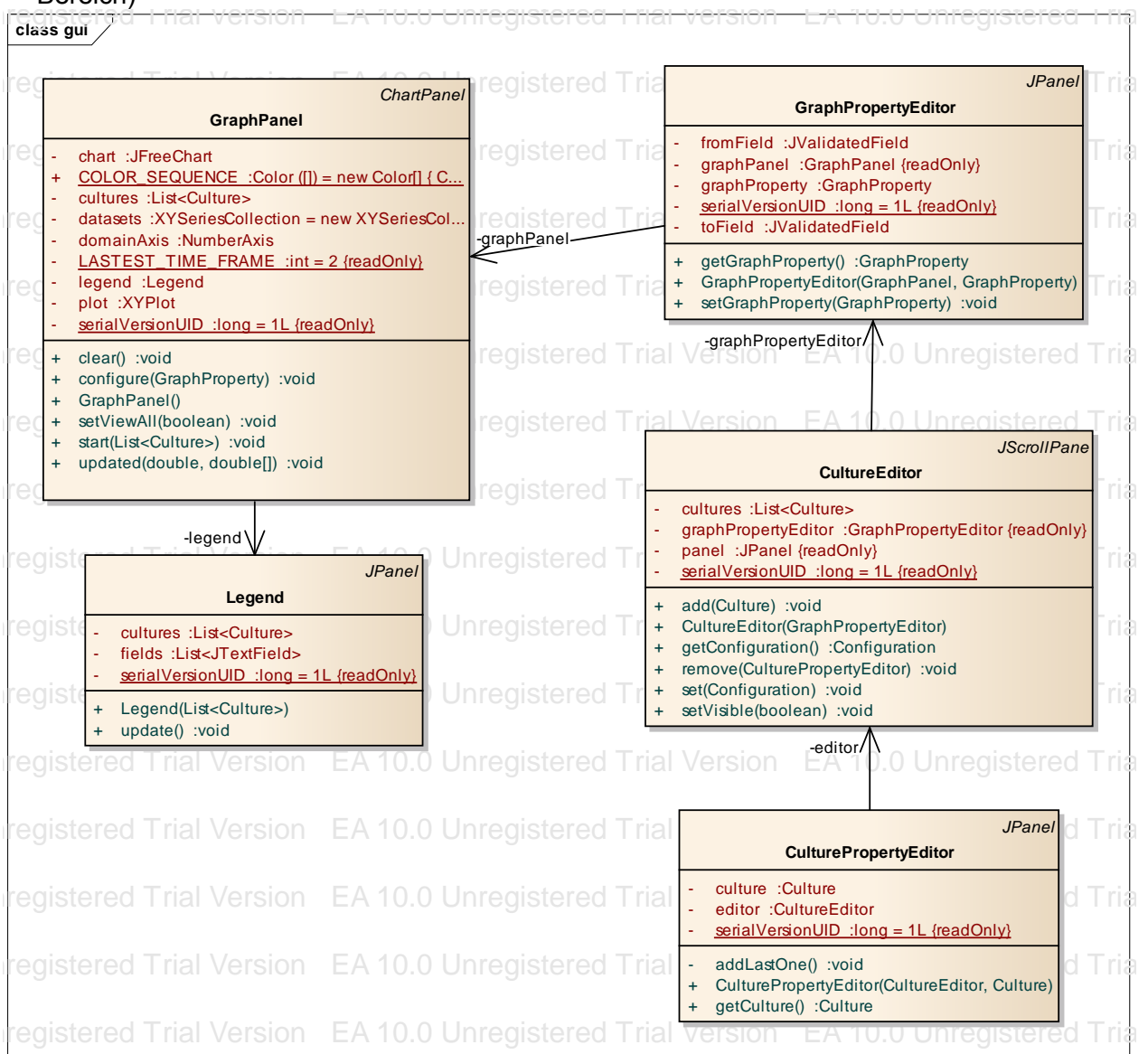
Das Modell für die Simulation ist sehr einfach. Es gibt ein *Simulations*-Objekt, das die ganze Simulation steuert und die Berechnungen durchführt. Eine *Culture* repräsentiert eine Population mit dem Namen, Anfangswert und der Wachstumsfunktion (*expression*). Das GUI oder auch andere Komponenten können einen *Simulation-Listener* beim Simulations-Objekt registrieren, um bei neuen Berechnungen notifiziert zu werden.



4.4.2. GUI-Klassen

Das nachfolgende Klassendiagramm zeigt natürlich nicht alle GUI-Klassen. Aber die wichtigsten Komponenten sind darauf ersichtlich:

- *GraphPanel*: Zeichnet den Grafen, die Legende, Simulations-Listener
- *CultureEditor/CulturePropertyEditor*: Editoren um die Konfigurationen des Dynamischen Systems vorzunehmen
- *GraphPropertyEditor*: Editor um den Grafen zu konfigurieren (aktuell nur den Wertebereich)



4.5. Expression Language

Eine sehr wichtige Anforderung und auch eine der Herausforderungen in diesem Projekt ist die Unterstützung der Eingabe von mathematischen Ausdrücken. Das Interpretieren solch eines Inputs selber zu programmieren wäre sehr aufwendig und würde den Rahmen dieser Projektarbeit sprengen.

4.5.1. JEXL – Java Expression Language

Folgende Library von Apache Commons haben wir in unser Projekt eingebunden:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-jexl</artifactId>
  <version>2.1.1</version>
</dependency>
```

Die wichtigste Klasse dieser Library für uns aus Benutzersicht ist die **JexlEngine**. Die Engine bietet die Methoden **createScript(String input)** und **createExpression(String input)**, welche eine mathematische Eingabe interpretiert und auswertet.

4.5.2. Hilfsklasse: **ch.zhaw.dynsys.el.utils.ExpressionUtil**

Wie im Klassendiagramm ersichtlich ist, werden die einzelnen Kulturen/Populationen mit der Klasse **ch.zhaw.dynsys.simulation.Culture** abgebildet. Diese Kultur-Objekte wiederum beinhalten die auszuwertenden mathematischen Ausdrücke.

Die Hilfsklasse muss nun diese Kultur-Objekte entgegen nehmen, alle Ausdrücke auslesen und sie zu einem sinnvollen validen JEXL Script zusammenfügen.

4.5.2.1. Konkretes Beispiel:

- Populationen:

Hasen
$H = 0.002 * K$
100

Karotten
$K = -0.001 * H$
50

Wir haben also zwei Wachstumsfunktionen (H und K), jeweils abhängig voneinander, und die dazugehörigen Startwerte für die Populationen (100 und 50).

- Generiertes JEXL Script:


```

{
var K = 50.0
var K_diff = 0.0
var H = 100.0
var H_diff = 0.0
}
return [
    new("java.lang.Double",-0.001 * H),
    new("java.lang.Double",0.002 * K)
];

```

Dieses Script wird nun der JexlEngine übergeben. Da wir bloss einen Rückgabewert haben dürfen, geben wir ein Array zurück, in welchem die ausgewerteten Funktionen in derselben Reihenfolge drin sind, wie sie als Kulturen übergeben wurden.

Die Variablen "K_diff" und „H_diff“ werden automatisch dazu generiert – sie entsprechen jeweils der Wachstumsrate (also der Ableitung) der dazugehörigen Kultur und können vom Benutzer selbst auch verwendet werden.

- Auswertung des Scripts nach dem ersten Schritt:

Wenn das Script von der JexlEngine ausgeführt wird, erhalten wir den oben definierten Rückgabewert, das heisst, wir erhalten ein Array von Doubles:

```
double[] results = (double[]) script.execute(context);
```

In unserem konkreten Fall erhalten wir das nachfolgende Array:

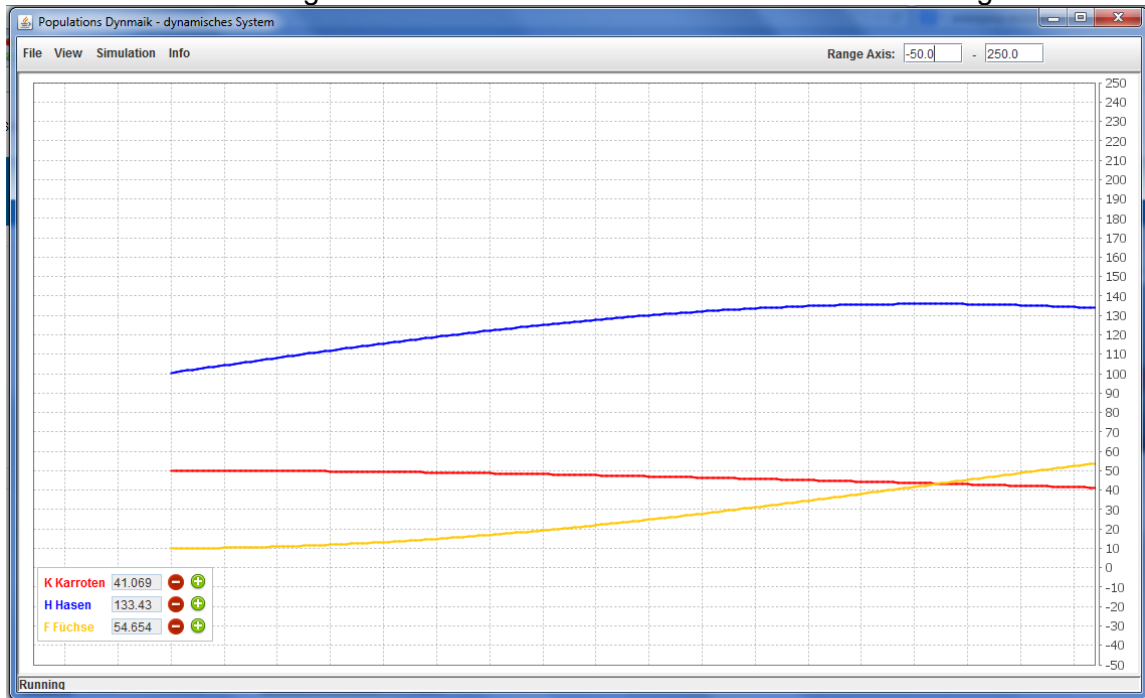
```
[-0.1, 0.1]
```

Diese beiden Werte entsprechen dem Resultat der Funktionen mit den Startwerten die gegeben wurden.

4.6. Grafische Benutzeroberfläche

4.7. Darstellung der Simulation

Der Verlauf jeder Population über die Zeit wird mit einer Kurve in einer eigenen Farbe gezeichnet. Dabei ist die Reihenfolge der Farben immer festvorgegeben, damit es nie zu einer Verwechslung kommt wenn eine Simulation mehrmals durchgeführt wird.



4.7.1. X-Achse

Die X-Achse ist in diesem Grafen nicht beschriftet. Dies ist absichtlich so gemacht, da eine sinnvolle Beschreibung nur sehr schwierig umsetzbar ist. Tatsächlich würde es im Prinzip dem Index der aktuellen Iteration entsprechen, also der Anzahl Iterationsschritten, die wir bei der Berechnung der Integrale machen. Dies ist aber bei solch einer Simulation keine interessante Information.

In einem ersten Anlauf hatten wir versucht, den aktuellen Index auf die vergangene Zeit umzurechnen. Dies hat aber später zu Problemen geführt, weshalb wir dies wieder ausgebaut haben. Für den Benutzer jedenfalls ist durch die Verlaufsrichtung der Kurven intuitiv klar, dass es sich um die „Zeit-Achse“ handelt.

4.7.2. Y-Achse

Die Werte auf der Y-Achse sind die berechneten Werte. Im Prinzip setzen sie sich zusammen aus der zu Beginn definierten Startpopulation und dem berechneten Wachstum über die Iterationen.

4.7.3. Legende

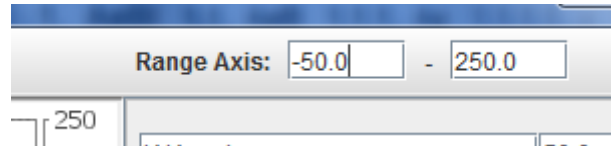


Zu Beginn haben wir die Standard-Legende verwendet, die man mit JFreeChart erzeugen kann. Als später die Anforderung bezüglich der Konfigurierbarkeit der aktuellen Populationswerte dazukam, entschieden wir uns aus Platzgründen die Legende mit einer eigenen

Implementation zu ersetzen, welche gleichzeitig auch die aktuellen Werte anzeigt und Buttons zum Regeln der Werte hat.

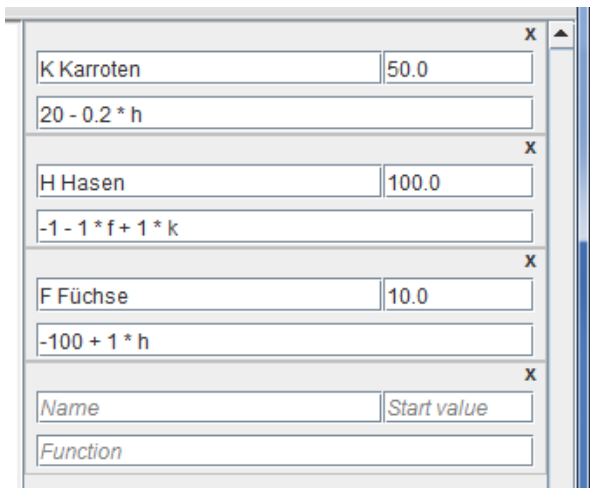
4.7.4. Konfiguration des Werte-Bereichs

Der Y-Achsenabschnitt für die Simulation muss fest vorgegeben werden. Technisch ist es kein Problem, den Werte-Bereich grafisch dynamisch während der Simulation anzupassen. Aber wir mussten feststellen, dass dies für den Benutzer störend ist und das Bild unruhig wirken lässt.



Konfiguriert werden kann der Werte-Bereich sowohl vor dem Simulationsstart wie auch während dem Verlauf.

4.7.5. Konfiguration des Dynamischen Systems



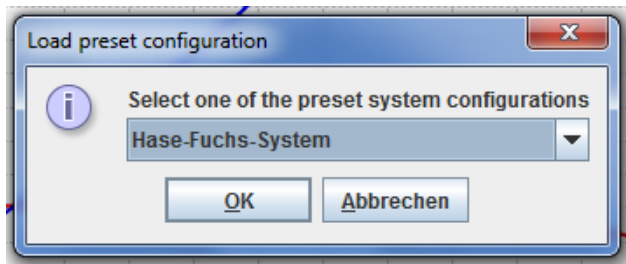
In der rechten Leiste konfiguriert man die Kulturen des Dynamischen Systems. Jede Kultur besteht aus folgenden Angaben:

- Variable + Name/Beschreibung
- Startgrösse der Population
- Wachstumsfunktion der Population

Die Konfigurationsleiste wird dynamisch beim Start der Simulation ausgeblendet und beim Stoppen wieder eingeblendet. So hat man während der Simulation immer den vollen Fensterbereich um den Grafen zu zeichnen.

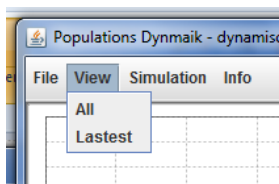
4.7.6. Vordefinierte Konfigurationen Laden

Über einen kleinen Selektions-Dialog kann eine der vordefinierten Konfigurationen geladen werden. Es werden automatisch alle aktuellen Kulturen gelöscht und die gewählte Konfiguration importiert.



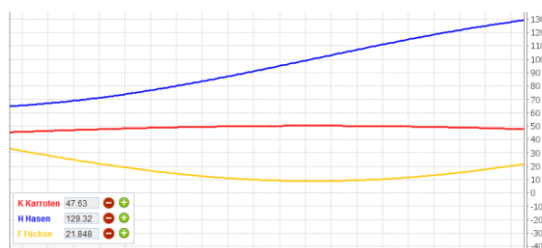
4.7.7. Ganze Simulation betrachten

Während der Simulation sieht man normalerweise immer nur gerade einen bestimmten Zeitausschnitt. Die Grafen laufen im Prinzip hinten einfach raus. Damit man sich aber auch das Gesamtbild betrachten kann, stehen im Hauptmenü unter *View* zwei Modi zur Verfügung:

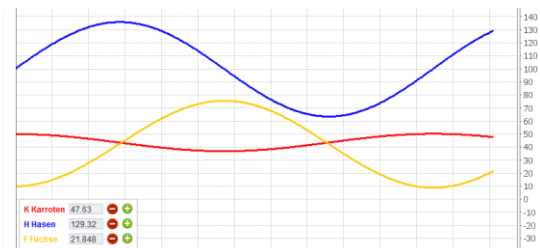


So kann schon während oder auch erst nach der Simulation betrachtet werden, wie die Kurven insgesamt verlaufen sind:

Letzter Abschnitt:



Gesamthaft:



5. Schlusswort

Bei der Wahl des Projektthemas war die Versuchung gross, ein Thema aus dem Algorithmen & Datenstrukturen Bereich zu nehmen. Da wir beide beruflich als Software-Entwickler tätig sind und wir im letzten Semester das Modul Algorithmen & Datenstrukturen besucht hatten, wäre der Einstieg in solch ein Thema und vermutlich auch die Umsetzung etwas leichter gewesen.

Eine kleine Applikation zu entwickeln, die ein physikalisches oder numerisches Problem löst, klang für uns aber interessanter. Es gab uns die Möglichkeit, etwas Neues kennen zu lernen und einmal etwas völlig anderes zu programmieren. Wir brauchten zwar einigen Aufwand um dann in die Themen Dynamische Systeme und Populationsdynamiken einzusteigen, aber es hat auch Spass gemacht daraus eine Applikation zu entwickeln. Die fachliche Projekt-Betreuung durch Herrn Heuberger war für uns ideal, da er sich in diesem Gebiet schon sehr gut auskannte und uns wichtige Informationen und Tipps geben konnte.

Das Programm ist schlussendlich mehr oder weniger so herausgekommen, wie wir es zu Beginn definiert hatten. Es ist sehr schlicht gehalten und hat eine einfache grafische Benutzeroberfläche, die aber dennoch vieles abdeckt. Man kann jetzt beliebige Simulationen konstruieren und mit Hilfe der Applikation durchspielen, man kann aber auch lediglich einfache Differenzialgleichungen darstellen. Erweiterungspotenzial hat das Programm wie es in der Software-Entwicklung üblich ist natürlich auch – aber wir konnten alle Anforderungen, die wir zu Beginn gestellt haben, damit erfüllen.

6. Quellenverzeichnis

6.1. Projektstruktur auf GitHub

Repository: <https://github.com/delsener/ch.zhaw.softwareprj2.git>

- [/documentation](#) : Vorliegende Projektdokumentation und Präsentation
- [/implementation](#) : Implementierung der Applikation (Source Code, Configs, ...)
- [/prjplanning](#) : Artefakte aus der Projektplanung
- [/dynsys.jar](#) : Ausführbares Programm (aktuellster Stand)

6.2. Theorie: Dynamische Systeme und Populationsdynamik

- Podcast zum Thema Populationsdynamik:
http://pod.drs.ch/mp3/kontext/kontext_201302271002_10258989.mp3
05.03.2013
- Dynamische Systeme und gewöhnliche Differentialgleichungen:
<http://www.math.uni-hamburg.de/home/lautebach/scripts/dsode09/dsode.pdf>
10.03.2013
- Populationsdynamik:
<http://de.wikipedia.org/wiki/Populationsdynamik>
10.03.2013
- Einführung in (gewöhnliche) Differentialgleichungen:
http://www3.mathematik.tu-darmstadt.de/fileadmin/home/users/186/Skripte_Roch/gdgl_gesamt.pdf
11.03.2013
- Kinetik nach Monod:
http://www.buetzer.info/fileadmin/pb/pdf-Dateien/Monod_Kinetik.pdf
15.04.2013