**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](#_Toc355704642)

[1.1. Thematik 4](#_Toc355704643)

[1.2. Ziel der Arbeit 4](#_Toc355704644)

[1.3. Realisierungspunkte 4](#_Toc355704645)

[2. Populationsdynamik 6](#_Toc355704646)

[2.1. Was ist Populationsdyamik? 6](#_Toc355704647)

[2.2. Hefe-Zucker-System 7](#_Toc355704648)

[2.3. Räuber-Beute-System 8](#_Toc355704649)

[2.4. DS mit zweiter Ableitung **Fehler! Textmarke nicht definiert.**](#_Toc355704650)

[3. Projektplanung 10](#_Toc355704651)

[3.1. Agile Softwareentwicklung 10](#_Toc355704652)

[3.1.1. Iterationsplanung 10](#_Toc355704653)

[3.1.2. Burndown-Charts 10](#_Toc355704654)

[3.1.3. Taskliste 11](#_Toc355704655)

[3.2. Iteration 1 12](#_Toc355704656)

[3.2.1. Geplante Tasks 12](#_Toc355704657)

[3.2.2. Burndown-Chart 12](#_Toc355704658)

[3.2.3. Retrospektive 13](#_Toc355704659)

[3.3. Iteration 2 14](#_Toc355704660)

[3.3.1. Geplante Tasks 14](#_Toc355704661)

[3.3.2. Burndown-Chart 14](#_Toc355704662)

[3.3.3. Retrospektive 14](#_Toc355704663)

[3.4. Iteration 3 16](#_Toc355704664)

[3.4.1. Geplante Tasks 16](#_Toc355704665)

[3.4.2. Burndown-Chart 16](#_Toc355704666)

[3.4.3. Retrospektive 16](#_Toc355704667)

[4. Realisierung 17](#_Toc355704668)

[4.1. Anforderungen 17](#_Toc355704669)

[4.1.1. Eingabe von DS Konfigurationen 17](#_Toc355704670)

[4.1.2. Laden von vordefinierten DS Konfigurationen 17](#_Toc355704671)

[4.1.3. Speichern und Laden von eigenen DS Konfigurationen 17](#_Toc355704672)

[4.1.4. Grafische Darstellung der Simulation 17](#_Toc355704673)

[4.2. Tools, Technologien und Frameworks 17](#_Toc355704674)

[4.2.1. Java 1.7 und Swing 17](#_Toc355704675)

[4.2.2. Eclipse IDE 17](#_Toc355704676)

[4.2.3. Git und Github 17](#_Toc355704677)

[4.2.4. Maven 3.0.4 18](#_Toc355704678)

[4.2.5. JUnit 4.8.1 18](#_Toc355704679)

[4.2.6. JFreeChart 1.0.13 18](#_Toc355704680)

[4.2.7. Apache Common JEXL 18](#_Toc355704681)

[4.3. Projektstruktur 20](#_Toc355704682)

[4.4. Architektur / Klassendiagramm 21](#_Toc355704683)

[4.5. Expression Language 22](#_Toc355704684)

[4.5.1. JEXL – Java Expression Language 22](#_Toc355704685)

[4.5.2. Hilfsklasse: ch.zhaw.dynsys.el.utils.ExpressionUtil 22](#_Toc355704686)

[4.6. Grafische Darstellung der Simulation 24](#_Toc355704687)

# Einleitung

## 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.

## 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.

## 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

# Populationsdynamik

## Was ist Populationsdyamik?

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 mutiert. Dabei erhält man Rückschlüsse auf Idealbedingungen der Kulturen. Die gewonnene Erkenntnis wird anschliessend auf die Natur angewendet.

## 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:

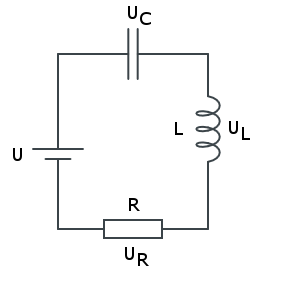
Mit und für die Population der Hefe bzw. des Zuckers und bzw. für das Hefewachstum und die Zuckerabnahme.

## 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:

Mit , und für die Population der Karotten, Hasen und Füchse und , und für die Karotten-, Hasen- und Fuchswachstumrate.

## 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 , einem Widerstand , einer Spule und einem Kondensator.

Es gilt:

Daraus folgen die zwei Differentialgleichungen:

# Projektplanung

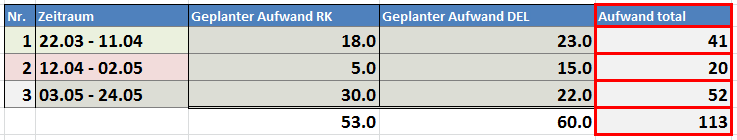
## Agile Softwareentwicklung

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.

### Iterationsplanung

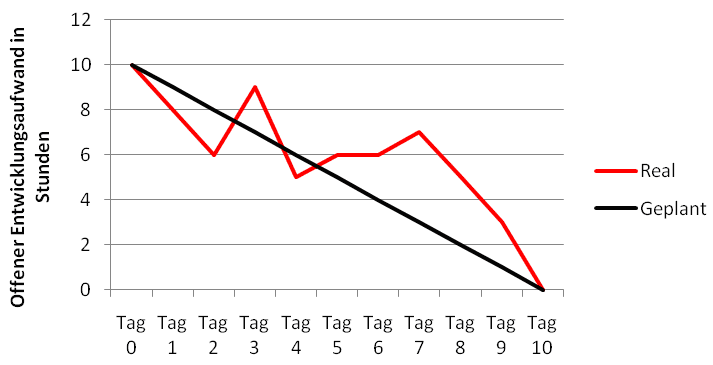
Um eine flexible Planung zu garantieren und so besser auf Unvorhergesehenes reagieren zu können, werden wir die gesamte Projektzeit in drei Iterationen aufteilen:



### 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.

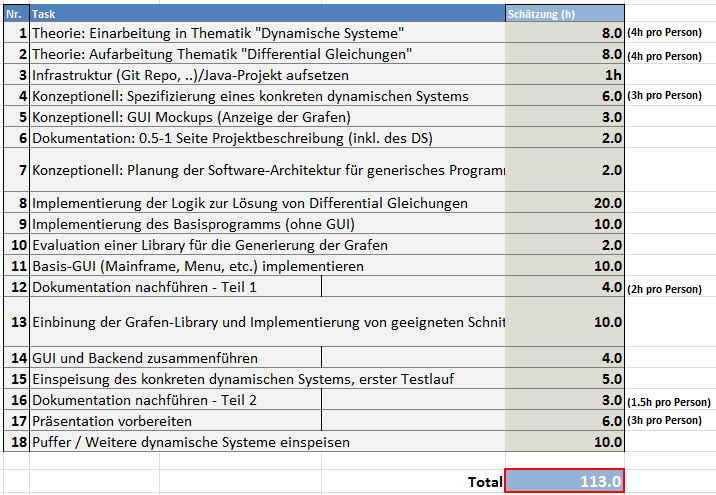
Ein exemplarisches Burndown-Chart (entspricht nicht unserem Verlauf):



### 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:



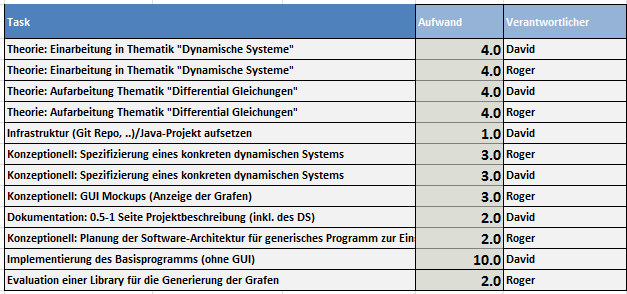
## Iteration 1

Zeitraum: 22.03 – 11.04

Ressourcen: 42 Stunden

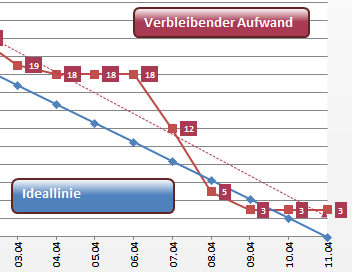
* Roger: 18 Stunden
* David: 24 Stunden

### Geplante Tasks



### Burndown-Chart

Mit folgendem Bild endete die erste Iteration (Ausschnitt):



### 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.

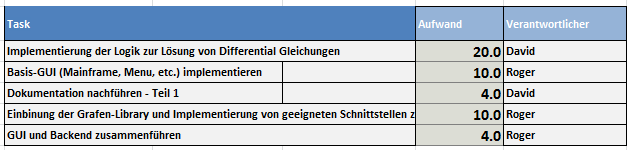
## Iteration 2

Zeitraum: 12.04 – 02.05

Ressourcen: 48 Stunden

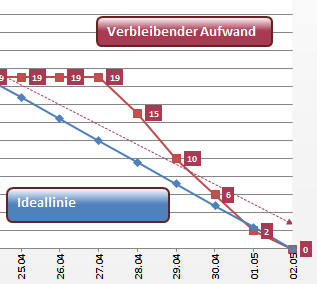
* Roger: 24 Stunden
* David: 24 Stunden

### Geplante Tasks



### Burndown-Chart

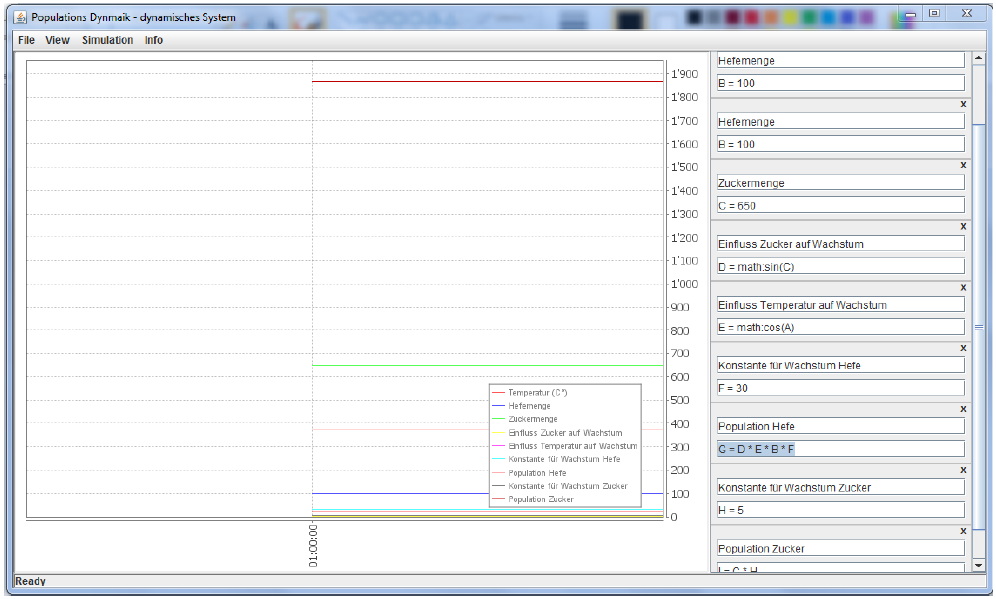
Mit folgendem Bild endete die zweite Iteration (Ausschnitt):



### 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:



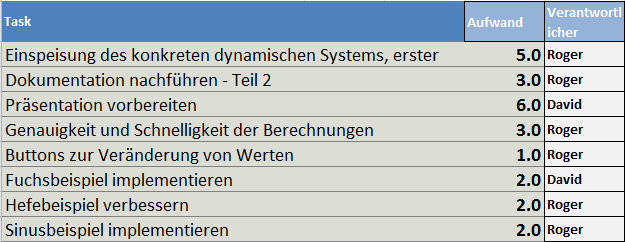
## Iteration 3

Zeitraum: 03.05 – 24.05

Ressourcen: 52 Stunden

* Roger: 30 Stunden
* David: 22 Stunden

### Geplante Tasks



### Burndown-Chart

Mit folgendem Bild endete die zweite Iteration (Ausschnitt):

### Retrospektive

# Realisierung

## Anforderungen

Nachfolgend werden kurz die wichtigsten Anforderungen an die Applikation beschrieben.

### 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.

### 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.

### 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.

### 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.

## Tools, Technologien und Frameworks

### 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.

### Eclipse IDE

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

### 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.

### 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>

### 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.

### 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](http://www.object-refinery.com/).

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>

### 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, habe 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)))

];

## 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: 
    - Beschreibung: Beinhaltet alle Klassen, welche zur Darstellung der grafischen Oberfläche gebraucht werden
  + **Logik/Simulations-Komponenten**:
    - Package: 
    - 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: 
    - Beschreibung: Um mit der JEXL Library arbeiten zu können, brauchen wir Hilfsklassen (Utility/Helper). Diese befinden sich in diesem separaten Package.
  + **Persistenz-Klassen**:
    - Package: 
    - Beschreibung: Alle Klassen, welche für die Clientseitige Persistierung (Serialisierung und Files) benötigt werden, befinden sich in diesem Package.

## Architektur / Klassendiagramm

## 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.

### 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>

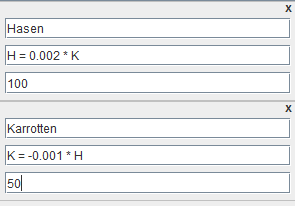
Die wichtigste Klasse dieser Library für uns aus Benutzersicht ist die **JexlEngine**. Die Engine bietet die Methoden **createScript(String input)** und **createExpression(String input)**, welche eine mathematische Eingabe interpretiert und auswertet.

### Hilfsklasse: ch.zhaw.dynsys.el.utils.ExpressionUtil

Wie im Klassendiagramm ersichtlich ist, werden die einzelnen Kulturen/Popoulationen 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.

#### Konkretes Beispiel:

* Populationen:  
    
  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.

## Grafische Darstellung der Simulation