Zentrum für Angewandte Mathematik und Physik
Zürcher Hochschule für Angewandte Wissenschaften

Bachelorthesis Frühlingssemester 2012

Entwicklung eines graphischen Editors zur Modellierung von Systemen mit dynamischer Modellstruktur

Andreas Bachmann
bachman0@students.zhaw.ch

Andreas Butti
buttiand@students.zhaw.ch

8. Juni 2012

Betreuer:
Prof. Dr. Stephan Scheidegger
School of Engineering
Technikumstrasse 9
8400 Winterthur
Telefon: 058 934 74 63
stephan.scheidegger@zhaw.ch

Betreuer:
Dr. Rudolf Marcel Füchslin
School of Engineering
Technikumstrasse 9
8400 Winterthur
Telefon: 058 934 75 92
rudolf.fuechslin@zhaw.ch

Abstract

Englische Version von “Zusammenfassung”

Zusammenfassung

Wird erst am Ende der Arbeit geschrieben

Inhaltsverzeichnis

1. Einleitung **2**
- 1.1. Bereits existierende Tools . . . 2
- 1.2. Vorgabe . . . 2
- 1.3. Problemstellung (oder Motivation) . . . 2

2. Theorie **3**
- 2.1. Analytisch oder Numerisch . . . 4
- 2.2. Analytische Lösungsverfahren . . . 4
  - 2.2.1. Gewöhnliche Differentialgleichungen . . . 4
  - 2.2.2. Partielle Differentialgleichungen . . . 5
- 2.3. Numerische Lösungsverfahren . . . 5
  - 2.3.1. Einschrittverfahren . . . 5
  - 2.3.2. Gradienten-Verfahren . . . 9

3. Methode **9**
- 3.1. Graphische Darstellung der Modelle . . . 9
- 3.2. Werkzeuge und Hilfsmittel . . . 10
  - 3.2.1. Programmiersprachen & Entwicklungsumgebung . . . 10
  - 3.2.2. Markup Language . . . 12
  - 3.2.3. Plugin Handling . . . 12
  - 3.2.4. Dateiformat . . . 13

4. Vorgehen **15**
- 4.1. Softwarearchitektur . . . 15
- 4.2. Technische Beschreibung Softwarekomponenten . . . 18
- 4.3. Tests und Validierung . . . 22

5. Resultate **22**
- 5.1. Numerische Lösungsverfahren . . . 22
  - 5.1.1. Diffusionsgleichung . . . 22

6. Diskussion und Ausblick **22**

7. Literaturverzeichnis **24**

I. Anhang **24**

8. Codeanmerkungen **24**

9. Glossar **25**

10. Weiterentwicklung **25**
- 10.1. Erstellen eines neuen Simulationsplugins . . . 25
- 10.2. Erstellen eines neuen GUI Elementes . . . 31
- 10.3. Ant Buildsystem . . . 31
- 10.4. Setup . . . 33

11. Benutzerhandbuch **33**
11.1. Installation 33
11.1.1. Portable Version 33
11.2. Übersicht / Begriffsdefinitionen 33
11.3. Erstellen eines einfachen Flowmodells 35
11.4. Tastenkombinationen 35
11.4.1. Grundlegend 35
11.4.2. Hauptfenster 35

Vorwort

(= persönliches)

In der Forschung aber auch zu Unterrichtszwecken werden viele Theorien vermittelt und Simuliert. Solche Simulationen basieren meistens auf einem strikt mathematischen Hintergrund, meinst Integrale erster Ordnung. Forschende Personen im Bereich Physik / Biologie verwenden dabei bevorzugt eine Modellierungssoftware, da es nicht ihr Fachgebiet ist selbst zu programmieren. Andreas Butti hat sich bei der Verwendung von Simulationstools über deren Plattformabhängigkeit und Benutzerunfreundlichkeit gestört. Als Informatiker kommt man da schnell in Versuchung selbst etwas besserer zu schreiben. Nach einem Gespräch mit dem Physikdozenten, Herr Scheidegger, wurde daraus dann diese BA, die jedoch nicht nur ein Benutzerfreundliches Simulationswertzeug sein soll, sondern auch bisher nicht vorhandene Möglichkeiten für die Simulationen von Biologischen Abläufen, wie das innere einer Zelle, darstellen soll. TODO: Danksagung

1. Einleitung

fusion->trennen von Meso (nicht implementiert)
FSM->nicht invivo

Es gibt bereits viele Simulationstools, wie z.B. Berekely Madonna um nur ein bekanntes Beispiel zu nennen. Diese Tools unterstützten die Modelierung von Differentialgleichungen als Modell, mit Containern und Flüssen.

Das gleiche Prinzip wird auch von Symlink verwendet, dieses verwendet als Symbole jedoch konsequent die Elektrotechnischen Abbildungen, es gibt Verbindung, Verstärker, Verzögerungen (x[t-1]) etc. Obwohl das Modell damit etwas anders aussieht ist die mathematische Abbildung schlussendlich die gleiche.

Da unsere Simulation keine Elektrotechnischen Hintergrund hat haben wir für die Darstellungen unserer Flussmodelle ebenfalls Container mit Flüssen (Integral) verwendet.

1.1. Bereits existierende Tools

1.2. Vorgabe

Bestehende graphische Modelleditoren erlauben eine effiziente Modellierung von kompartimentalen Systemen. Dabei unterstützt die graphische Oberfläche die Strukturierung des Modells bzw. des Systems. Dies kann gerade bei der Erfassung von komplexen Systemen den Zugang zu einer adäquaten Systembeschreibung erleichtern. Gerade aber Modelleditoren wie Berkeley-Madonna sind auf eine kompartimentale Struktur des Systems angewiesen. Räumlich strukturierte bzw. verteilte Systeme lassen sich nur schwer und in vereinfachter Form abbilden. Die Verwendung oder Kopplung verschiedener Simulationswerkzeuge kann für gewisse technische Systeme in Betracht gezogen werden (z.B. elektrische Schaltung mit Komponenten, bei denen die Wärmeabstrahlung und oder Wärmeleitung räumlich modelliert werden). Bei vielen Systemen lässt sich aber durch eine solche Kopplung das System nicht abbilden. Bei biologischen Systemen z.B. können sich Kompartimente bewegen (bei Zellen z.B. Chemo- und Haptotaxis). Zudem zeichnen sich biologische Systeme durch hierarchische Kompartimentstrukturen mit Unterkompartimenten aus. Ein weiterer Aspekt betrifft die Möglichkeit, dass Kompartimente in biologischen Systemen fusionieren oder sich teilen können.

Anforderungen: Das zu entwickelnde Modellierungswerkzeug soll an die intuitive graphische Oberfläche bestehender Modellierungswerkzeuge für kompartimentale Simulationen anknüpfen. Folgene Aspekte sollen konzeptuell untersucht und wenn möglich implemetiert werden:

- Hierarchische Kompartimente
- Räumliche Positionierung von Kompartimenten, welche die Wechselwirkung der Kompartimente auf gleicher Stufe beeinflussen kann und somit Einführung von Koordinaten (erster Schritt 2-Dim.) bzw. orthogonales Grid und beschreibung von Gradienten (z.B. für Änderung der räumlichen Position von Kompartimenten aufgrund von z.B. Gradienten)
- Ausgabe eines Codes in einer Markup Language (z.B. SBML), welcher von einem bestehenden Solver ausgeführt werden kann (z.B. Matlab)

1.3. Problemstellung (oder Motivation)

Motivation durch Problemstellung!
signalig chains (SimuLink), Biologie. Motivation->Problem->Insiliko

Mit unserer Simulation ist es zusätzlich möglich in einem XY Modell mehrere Meso Kompartmente abzubilden, ein Meso Kompartment ist das vorhin genannte Flussmodel. Diese Meso Kompartmente können sich wären der Simulation im XY-Raum bewegen. Es können Dichten angegeben werden, die über den XY Raum verteilt sind, und die Meso Kompartmente können an Ihrer aktuellen Position von der dichte Konsumieren oder dichte Produzieren, somit kann die Umgebung beeinflusst werden. Mit diesen Fähigkeiten ist es möglich das innenleben einer Zelle oder andere Biologische Prozesse einfach, grafisch abzubilden. Die Idee und Vorgabe dieser Simulationsmethode stammt von Herr Scheidegger, und wurde zusammen mit Herr Füchslin und uns ausgearbeitet.

2. Theorie

Um Biologische Prozesse zu verstehen, nachzubilden und zu entwickeln werden mathematische Systeme modelliert und danach mit rechnerunterstützten Solvern Näherungslösungen berechnet. Als Beispiel kann sich die Population $N(t)$ eines Bakterienvolkes in einer Petrischale mit der Zeit ändern, je nach dem welche äussere Einflüsse auf die Bakterien einwirken. Die Geburten- und Sterberate bilden die Änderung $\frac{dN}{dt}$ der Population $N(t)$. Dabei werden oftmals Differentialgleichung (DGL) verwendet. Im Falle von Bakterien kann von einem exponentiellen Wachstum ausgegangen werden.

$$\begin{aligned} \frac{dN}{dt} &= Geburtenrate - Sterberate \\ \frac{dN}{dt} &= \alpha N - \beta N, \quad N \geq 0 \end{aligned} \tag{2.1}$$

Doch das Wachstum ist Beschränk auf das Nährmedium in der Petrischale. Das Bakterienvolk kann sich nicht mehr Fortpflanzen, wenn keine Nahrung mehr zur Verfügung steht. Eine Nährstoffkonzentration $c(t)$ wird eingeführt. Je mehr Bakterien entstehen, desto schneller fällt die Konzentration. Die fallende Konzentration $c(t)$ wiederum wird als Hemmungsterm in die Wachstumsfunktion $\frac{dN}{dt}$ aufgenommen. Diese Populationsfunktion $N(t)$ gehört zu den Logistischen Funktionen, die wiederum Teil der Sigmoidfunktionen ist. Zu Beginn wächst die Funktion beinahe exponentiell. Ihren Höhepunkt erreicht sie am Wendepunkt, wenn die Änderung der Population am grössten ist. Nach endlicher Zeit tritt eine Sättigung ein. Die Herausforderung ist aber, welche Werte N_0, c_0 initial verwendet werden. Denn, wenn eine alternative Anfangsbedingung genommen wird, verändert sich auch die Lösung. Dies wird als Anfangswertproblem (AWP) bezeichnet [Sch11]

$$\begin{aligned} \frac{dc}{dt} &= Zuflüsse - Abflüsse \\ \frac{dc}{dt} &= -\gamma N, \quad c \geq 0 \\ \frac{dN}{dt} &= c(t) \cdot \alpha N - \beta N, \quad N \geq 0 \end{aligned} \tag{2.2}$$

In diesem Abschnitt wird eine Einführung in die numerischen Lösungsverfahren beschrieben, die ein Modelleditor ausführt um eine approximierte Lösung zu erhalten. In unserer Simulation haben wir verschiedene numerische Verfahren implementiert, die im Folgenden kurz vorgestellt werden.

Da sich diese Dokumentation an die Physik oder Biologie und nicht an die Mathematik richtet, wird keine mathematische Nomenklatur verwendet. So wird aus dem Zeitschritt h der Mathematik ein Δt der Physik und aus einer Ableitung y' der Mathematik ein $\dot{y}$ der Physik nach der Zeit t oder in ausgeschriebener Form $\frac{dy}{dt}$.

2.1. Analytisch oder Numerisch

In diesem Abschnitt wird der Nutzen aber auch die Grenzen des analytischen Verfahrens aufgezeigt. Auch wenn das numerische Verfahren schon früh bekannt war, wurden komplexe Modelle erst seit dem Siegeszug des Computers numerisch gerechnet. Zuvor war der numerische Weg zu Zeitaufwändig, da er von Hand berechnet werden musste. Komplizierte Geometrien wurden soweit vereinfacht, das sie analytisch oder gar nicht gelöst werden konnten.

In der allgemeinen Verwendung unterscheidet sich das numerische vom analytischen Verfahren mit der diskretisierung der Werte. Analytisch bleibt das Verfahren im Kontinuum. Es gibt keine Lücken zwischen zwei Werten. Bei der Numerik aber kann eine Zahl nur als Folge von Bits dargestellt werden. Wenn ein Bit ändert, ändert sich die Zahl und zwischendrin gibt es eine Lücke.

Falls das analytische Verfahren löstbar ist, ist das Resultat exakt. Jedoch können nur einfache Geometrien berechnet werden. Komplexe Geometrien, wie nicht lineare Differentialgleichungen, können nur in Ausnahmefällen analytisch gelöst werden, worauf in heutiger Zeit fast ausschliesslich ein numerisches Verfahren verwendung findet. [Kos94]

2.2. Analytische Lösungsverfahren

2.2.1. Gewöhnliche Differentialgleichungen

Eine Gewöhnliche Differentialgleichung (engl. Ordinary Differential Equation ODE) ist eine mathematische Gleichung, die Ableitungen, die Funktion selbst sowie die unabhängige Variable enthalten kann. Ableitungen und die Funktion selbst treten nach genau einer unabhängigen Variable auf. Lösung $y\,(t)$ einer Differentialgleichung $y^{(n)}$ist eine Funktion, die mit ihren Ableitungen deckungsgleich mit der Differentialgleichung selbst ist.

$$y^{(n)} = f\left(t,\ y\,(t)\,,\ \dot{y}\,(t)\,,\ \ldots,\ y^{(n-1)}\,(t)\right) \tag{2.3}$$

Auf analytischem Weg kann eine Differentialgleichung durch unbestimmte Integration erfolgen.

$$\int y^{(n)} \cdot dt = y^{(n-1)} + C \tag{2.4}$$

Für die computerunterstützte Berechnung von Simulationen dieser Art können verschiedene Techniken angewandt werden. Dabei gibt es Grundsätzlich zwei verschiedene Vorgehen: analytisch/symbolische oder numerische. Unsere Simulation verwendet das numerische Verfahren, wie es auch Berkeley Madonna tut. Das Vorgehen beruht auf einer numerischen Approximation von Gewöhnlichen Differentialgleichungen. Diese Art kann nur Differentialgleichung 1. Ordnung berechnen. Eine Differentialgleichungen 2. Ordnung ist in Gleichung 2.5 zu sehen.

$$a = \ddot{s}\,(t) = -g \tag{2.5}$$

Eine Gewöhnliche Differentialgleichung n. Ordnung kann aber in n Differentialgleichungen 1. Ordnung umgeformt werden. In den Gleichungen 2.6 wird die Differentialgleichung 2. Ordnung in zwei Differentialgleichung1. Ordnung umgewandelt.

$$\begin{aligned} \dot{v}(t) &= -g \\ \dot{s}(t) &= v(t) \end{aligned} \tag{2.6}$$

2.2.2. Partielle Differentialgleichungen

Eine Partielle Differentialgleichung (engl. Partial Differential Equation PDE) ist eine mathematische Gleichung, die partielle Ableitungen enthalten, also Ableitungen von Funktionen mehrere Variablen. Eine Dichte, zum Beispiel Nährboden, überdeckt eine Fläche, zum Beispiel eine Petrischale, im $(x,\ y) = \mathbb{R}^2$ mit einer Funktion $f(x\ y,\ t)$. Ein Bakterienvolk $N(t)$ verändert diese Dichte über die Zeit t aber auch bei jeder Position $(x,\ y)$ unterschiedlich stark. Bei komplexen Systemen kann es zu analytisch unlösbaren Differentialgleichungen führen. Generell können nur Lineare Partielle Differentialgleichung analytisch gelöst werden. Hier hilft die Numerik weiter, die zwar nur näherungsweise Lösungen ausgibt, doch besser eine Näherung als keine Lösung. Zwei wichtige numerische Verfahren sind die Finite-Differenzen-Methode (FDM) und die Finite-Elemente-Methode (FEM). [Sch11]

Die Finite-Differenzen-Methode die für die Simulation benötigt werden sind implementiert worden das Gradienten-Verfahren und die Diffusiongleichung, die später erläutert werden.

2.3. Numerische Lösungsverfahren

2.3.1. Einschrittverfahren

Zum näherungsweise Lösen von Anfangswertproblemen kann ein Einschrittverfahren von einem Integrationschritt t_n zum Nächsten t_{n+1} nur auf den gerade eben berechneten Wert y_n zurückgreifen, wie in Abbildung 2.1 zu sehen ist, nicht aber auf weiter zurückliegende Werte wie y_{n-1}. Falls eine Reihe von zuvor berechneten Werte genutzt werden möchte, ist ein Mehrschrittverfahren zu wählen. Unterschieden wird ein Verfahren auch noch in zwei Unterformen. Ein **explizites Verfahren** berechnet die Näherungswerte y_{n+1} nur aus Funktionswerten $f(t_n,\ y_n)$, die einen Schritt zurück liegen. Ein **implizites Verfahren** bezieht auch Funktionswerte $f(t_{n+1},\ y_{n+1})$ im aktuellen Schritt mit ein. Ein Verfahren kann s **Stufen** aufweisen, die s Funktionswerte-Berechnungen ausführt um einen Näherungswert y_{n+1} zu erhalten. Wie genau ein Verfahren rechnet kann mit der **Ordnung** p (Konsistenzordnung) ausgesagt werden. Genauer heisst hier mit weniger Abweichung vom exakten Wert (Fehler) pro Schritt, die sich über die Zeit kumulieren. Je genauer ein Verfahren rechnet, also je grösser die Ordnung ist, desto mehr Aufwand muss betrieben werden um einen besseren Näherungswert zu erhalten.

Euler

Das Euler-Verfahren, von **Leonard Euler** 1768 in seinem Buch *Institutiones Calculi Integralis* präsentiert, ist ein einfaches numerisches Verfahren, dass näherungsweise Lösungen einer Integration rechnet. Ausgangslage ist eine Differentialgleichung.

$$\frac{dy}{dt} = f(t,\ y(t)) \tag{2.7}$$

Da die Numerik mit diskreten Werten rechnet, kann eine Ableitung umgeschrieben werden in einen Differenzenquotient .

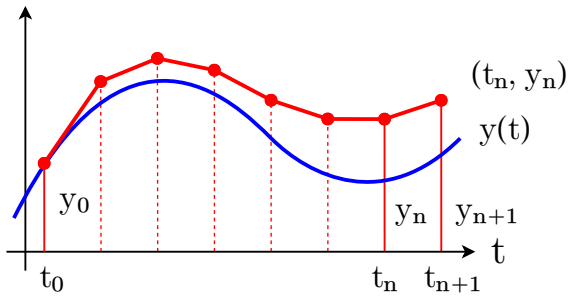

Abbildung 2.1: Diskrete Lösung einer DGL

$$\begin{aligned} \frac{\Delta y}{\Delta t} &= f\left(t,\ y\left(t\right)\right) \\ \Delta y &= \Delta t \cdot f\left(t,\ y\left(t\right)\right) \end{aligned} \tag{2.8}$$

Mit der Differentialgleichung und dem Anfangswert y_0 kann schrittweise bis zur gewünschten Endzeit numerisch Integriert werden.

$$y_{n+1} = y_n + \Delta y \tag{2.9}$$
$$y_{n+1} = y_n + \Delta t \cdot f\left(t_n,\ y_n\right) \tag{2.10}$$
$$t_{n+1} = t_n + \Delta t \tag{2.11}$$

Bei Differentialgleichungen 1. Ordnung und kleiner Schrittweite Δt erhält man ausreichende Genauigkeit. Ab einer höheren Ordnungen büst dieses Verfahren bei jedem Schritt an Genauigkeit ein. Es gibt bessere Verfahrung, die auch Ordnungen höheren Grades mit weniger Genauigkeitsverlust zulassen.

Butcher Tableau

Um weitere Verfahren besser zu Verstehen, wird zuerst eine Tabelle eingeführt. Die Tabelle wurde in den 1960er Jahren von **John Charles Butcher** entwickelt für den besseren Umgang mit dem nachfolgenden Runge-Kutta-Verfahren und nennt sich *Butcher tableau* in Gleichung 2.12. Die Tabelle beinhaltet einen Variations-Vektor c, eine Matrix $A = (a_{ij})$ und einen Gewichts-Vektor b wobei $i = 1..s,\ j = 1..s$ die Zeilen- und Spalten-Indizes und s die Dimension der Stufe ist. Verwendet wird nur ein Teil der Matrix A, nähmlich ein Dreieck von a_{21} schräg herunter zu $a_{s(s-1)}$.

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \rightarrow \begin{array}{c|cccc} 0 & & & & \\ c_2 & a_{21} & & & \\ \vdots & \vdots & \ddots & & \\ c_s & a_{s1} & \dots & a_{s(s-1)} & \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{2.12}$$

Wie beim Euler-Verfahren in Gleichung 2.9 ergibt sich ein neuer Wert y_{n+1} aus dem alten Wert y_n addiert mit einer festen Schrittweite Δt multiplitiert mit der Ableitung. Doch beim Runge-Kutta-Verfahren werden statt einer Ableitung verschieden gewichtete Ableitungen $b_i k_i$ aufsummiert, wobei das Gewicht b_i und die Ableitung k_i ist.

$$y_{n+1} = y_n + \Delta t \cdot \sum_{i=1}^{s} b_i k_i \tag{2.13}$$

Alle Gewichte b_i zusammen müssen 1 ergeben, sonst würde das Verfahren einen Teil der Ausgangswerte verlieren.

$$\sum_{i=1}^{s} b_i = 1 \tag{2.14}$$

Eine Ableitung, auch Zwischenschritt oder Stützstelle genannt, ist in Gleichung 2.15 erläutert. Je grösser die Dimension der Stufe s ist, desto mehr Zwischenschritte werden berechnet. Zu Beginn jedes Schrittes wird der Vektor k zurückgesetzt auf 0. Das heisst, beim ersten Zwischenschritt k_1 ist $c_1 = 0,\ k_i = 0,\ i = 1, ..., s$ und es ergibt sich $k_1 = f\left(t_n,\ y_n\right)$.

$$k_i = f\left(t_n + \Delta t \cdot c_i,\ y_n + \Delta t \cdot \sum_{j=1}^{s} a_{ij} k_j\right),\ i = 1, ..., s \tag{2.15}$$

Für den ersten Parameter der Funktion $f\left(t\ y\left(t\right)\right)$ in Gleichung 2.15 benötigen wir einen Koeffizienten c_i, der die unabhängige Variable t_n variert. Jedes $c_i,\ i = 1, ..., s$ bildet eine Summe aller Elemente eine Zeile der Matrix $A = (a_{ij})$.

$$c_i = \sum_{j=1}^{s} a_{ij} \tag{2.16}$$

Die Ordnungsbedingung, welche Ordnung p ein *Butcher tableau* besitzt, wird aus Zeitgründen aussenvorgelassen und auf die Referenz [Obe08] verwiesen. Jedoch kann die Ordnung p höchstens die Stufe s erreichen.

$$p \leq s \tag{2.17}$$

Klassisches Runge-Kutta

Um eine bessere Genauigkeit zu erhalten haben **Carl Runge** und **Martin Wilhelm Kutta** 1900, 60 Jahre vor der Präsentation des *Buchter Tableaus*, ein leistungsfähigeres Verfahren als Euler in Gleichung 2.9 entwickelt, die Differentialgleichungen numerisch lösen. Das heisst, sie kannten die Möglichkeit einer Tabelle noch nicht. Nachfolgen soll jedoch der Ansatz des *Buchter tableau* verwendet werden. Dabei rechnet das Klassische vier Zwischenschritte oder Stützstellen, arbeitet also mit Dimension der Stufe $s = 4$ und einer Ordnung $p = 4$. Die dazugehörigen Tabelle ist in Gleichung 2.18 zu finden.

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{2.18}$$

Mit $s = 4$ werden 4 Zwischenschritte berechnet, wobei wie in Gleichung 2.15 beschrieben t_n und y_n variert werden. Dabei beschränkt sich die Matrix $A = (a_{ij})$ auf das erwähnte Dreieck, da $a_{ij} = 0,\ j = i..s$.

$$\begin{aligned} k_1 &= f(t_n,\ y_n) \\ k_2 &= f(t_n + \Delta t \cdot c_2,\ y_n + \Delta t \cdot [a_{21} k_1]) \\ k_3 &= f(t_n + \Delta t \cdot c_3,\ y_n + \Delta t \cdot [a_{31} k_1 + a_{32} k_2]) \\ k_4 &= f(t_n + \Delta t \cdot c_4,\ y_n + \Delta t \cdot [a_{41} k_1 + a_{42} k_2 + a_{43} k_3]) \end{aligned} \tag{2.19}$$

Durch einsetzen der Werte aus der Matrix A in die Argumente kann nochmals gekürzt werden, da auch im Dreieck noch Null-Werte vorhanden sind.

$$\begin{aligned} k_1 &= f(t_n,\ y_n) \\ k_2 &= f(t_n + \Delta t \cdot \frac{1}{2},\ y_n + \Delta t \cdot \frac{1}{2} \cdot k_1) \\ k_3 &= f(t_n + \Delta t \cdot \frac{1}{2},\ y_n + \Delta t \cdot \frac{1}{2} \cdot k_2) \\ k_4 &= f(t_n + \Delta t \cdot 1,\ y_n + \Delta t \cdot 1 \cdot k_3) \end{aligned} \tag{2.20}$$

Schlussendlich wird eine Summe gewichteter Zwischenschritte berechnet, die wie beim Euler-Verfahren die Funktion $f(t,\ y(t))$ in Gleichung 2.9 übernimmt. Welche Gewichte genommen werden wird in Gleichung 2.13 erläutert.

$$\begin{aligned} y_{n+1} &= y_n + \Delta t \cdot \left(\frac{1}{6} \cdot k_1 + \frac{1}{3} \cdot k_2 + \frac{1}{3} \cdot k_3 + \frac{1}{6} \cdot k_4 \right) \\ t_{n+1} &= t_n + \Delta t \end{aligned} \tag{2.21}$$

Dormand-Prince Eine noch genauere Methode wurde in den 1980er von **John R. Dormand** und **Peter J. Prince** entwickelt, die eine höhere Ordnung aufweise wie das Klassische. Diese Methode gehört zu den eingebetteten Verfahren. Eingebetted darum, weil es noch einen zweiten Gewichts-Vektor b_2 neben dem ursprünglichen Gewichts-Vektor b_1 besitzt. Die Gewichts-Vektoren sind von unterschiedlicher Ordnung. Das Butcher Tableu ist in Gleichung 2.22 zu finden.

$$
\begin{array}{c|c}
\mathbf{c} & A \\
\hline
 & \mathbf{b_1^T} \\
 & \mathbf{b_2^T}
\end{array}
=
\begin{array}{c|ccccccc}
0 & & & & & & & \\
\frac{1}{5} & \frac{1}{5} & & & & & & \\
\frac{3}{10} & \frac{3}{40} & \frac{4}{5} & & & & & \\
\frac{4}{5} & \frac{44}{45} & -\frac{4}{5} & \frac{4}{5} & & & & \\
\frac{8}{9} & \frac{19372}{6561} & -\frac{4}{5} & \frac{4}{5} & \frac{4}{5} & & & \\
1 & \frac{9017}{3168} & -\frac{4}{5} & \frac{4}{5} & \frac{4}{5} & -\frac{4}{5} & & \\
1 & \frac{35}{384} & \frac{4}{5} & \frac{4}{5} & \frac{4}{5} & -\frac{4}{5} & \frac{4}{5} & \\
\hline
p=4 & \frac{35}{384} & 0 & \frac{35}{384} & \frac{35}{384} & -\frac{35}{384} & \frac{35}{384} & \frac{35}{384} \\
p=5 & \frac{35}{384} & 0 & \frac{35}{384} & \frac{35}{384} & -\frac{35}{384} & \frac{35}{384} & 0
\end{array}
\tag{2.22}
$$

Zuerst rechnet man wie beim klassischen Runge-Kutta alle k_i, $i = 1, ..., s$ wie in Gleichung 2.15 aus. Die Differenz $|z_{n+1} - y_{n+1}|$ zwischen dem Näherungswert 5. Ordnung z_{n+1} und dem Näherungswert 4. Ordnung y_{n+1} ergibt die Fehlertoleranz. Wenn die Toleranz zu gross wird, wird so lange die Schrittweite Δt erniedrigt, bis der Fehler wieder in der Toleranz liegt. Dieses Verfahren hat demzufolge eine Schrittweitensteuerung. [HB06, Dan09]

Die (AB)² Simulation besitzt zwar im Flow-Modell-Editor eine Möglichkeit die Simulation mit der Dormand-Prince-Methode auszuführen, ist aber noch nicht ausreichen getestet worden und darum noch im Alpha-Stadium.

2.3.2. Gradienten-Verfahren

3. Methode

3.1. Graphische Darstellung der Modelle

Diffusionsgleichung, Meso-> stoff konsumieren / ausscheiden

Text: für die graphise Darstellung...

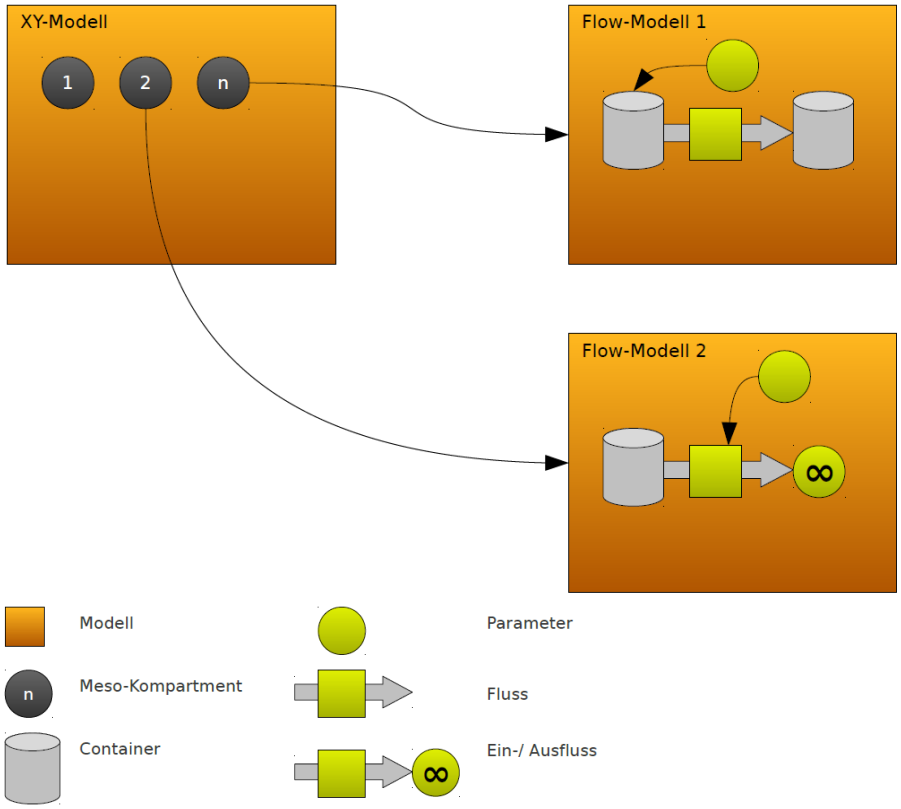

Abbildung 3.1: bla

Es kann entweder ein Herkömmliches «Flow-Modell» erstellt werden, das bereits bekannt ist da es vom Konzept her identisch bereits von vielen Simulationstools angeboten wird. Oder es kann ein «XY-Modell» erstellt werden, das dann Meso Kopartments beinhaltet, diese befinden sich an einer Position (X / Y) und verweisen auf ein Modell («Flow-Model-X»). Im «XY-Modell» könne sich «Dichten» befinden, das sind Stoffe die eine gewisse Konzentration an einer gewissen Stelle aufweisen. Ein Meso Kompartment kann sich wären der Simulation im «XY-Modell» bewegen, es kann Dichten konsumieren oder produzieren.

3.2. Werkzeuge und Hilfsmittel

Wie ist Lösung und warum diese Lösung?

Problem beschreiben: herausfinden was Problem ist / wo Problem ist und Lösungen finden

3.2.1. Programmiersprachen & Entwicklungsumgebung

Als Vorgabe haben wir erhalten das wir eine GUI Applikation schreiben sollen die Plattform unabhängig funktionieren sollte. Daher haben wir uns für Java entschieden. Java ist dank der Java Virtual Machine komplett Plattform unabhängig, und bietet mit Swing ein komplettes Framework für GUI Applikationen.

Die eigentliche Simulation wird mit Plugins realisiert, daher beschränkt sich die Auswahl der Programmiersprache nur auf den Editor, das Plugin kann in einer beliebigen Sprache geschrieben werden, es muss lediglich eine Java Klasse geschrieben werden, die dieses Plugin lädt, dies ist z.B. mit JNI (Java Native Interface) möglich. Die Argumente für / gegen eine Sprache gelten aber

grundsätzlich auch für die Simulation, abgesehen vom GUI Framework, das für die Simulation nicht benötigt wird.

Alternative Plattform unabhängige Programmiersprachen mit Vor- und Nachteilen:

- C / C++ (Weiter C++ genannt)
  - Positiv
    - Da C++ eine Hardwarenahe Sprache ist könnten Hardwarebeschleunigungen wie z.B. MMX bei Intel Prozessoren genutzt werden. Dies muss jedoch von Hand vorgenommen werden, und es ist zu beachten das diese Optimierungen für jede Prozessorarchitektur (z.B. Intel 32 bit, Intel 64 bit, ARM) separat vorgenommen werden müsste.
  - Negativ
    - C++ Beinhaltet keine GUI Bibliotheken, je nach Plattform kommen verschiedene Frameworks wie MFC (Windows), Cocoa (Mac OS X), GTK / QT (Linux) zum Einsatz. Die Framerworks GTK und QT sind jedoch auch Plattformunabhängig einsetzbar, und lassen sich auch unter Mac OS X / Windows einsetzen, dies ist jedoch mit einem höheren Aufwand verbunden als z.B. bei Java.
    - Die Applikation müsste für jedes Betriebssystem / jede CPU Architektur neu übersetzt werden. Unterschiede wie z.B. Little- / Big-Endian müssen bei der Entwicklung berücksichtigt werden, ansonsten können Fehler auftreten.
    - Das Programmieren ist aufwändiger, da in C++ Prototypen in Headerfiles deklariert werden müssen, diese müssen in korrekter Reihenfolge inkludiert werden etc.
- .NET
  - Positiv
    - Die .NET Umgebung von Microsoft läuft wie Java ebenfalls in einer VM, beinhaltet eine GUI Library und ist vom Syntax Java ähnlich
  - Negativ
    - Microsoft bietet nur eine VM für Windows an, es existiert zwar Mono als VM für Linux, Mac OS X ist jedoch schlecht unterstützt. Mono unterstützt nicht alle GUI Elemente die von Microsoft unterstützt werden.
    - Die Kompatibilität verschiedener .NET Versionen (z.B. 2.0 und 4.0) ist wesentlich schlechter als bei Java. Bei Java lässt sich auch Code der für eine ältere Version geschrieben wurde kompilieren, was bei .NET oftmals nicht der Fall ist.
- Skriptsprachen wie Python, Ruby etc.
  - Positiv
    - Einfache & Flexible Entwicklung
    - Kein Kompilieren notwendig
  - Negativ

* Grundsätzlich langsamer als Java (Python / Ruby können auch zu Java Bytecode kompiliert werden, dann fällt dieser Punkt weg, wir gehen aber von der Standard Runtime aus)
* Wie bei C++ keine Standard GUI Library, es muss ebenfalls auf GTK / QT zurückgegriffen werden
* Das Fehlerhandling ist schwerer, da bei Skriptsprachen oftmals Fehler erst zur Laufzeit festgestellt werden.

Java ist für den Benutzer die einfachste Lösung, da er nur die JVM installiert haben muss, es gibt keine weiteren Abhängigkeiten wie z.B. bei Skriptsprachen wie Python, welches zusätzlich noch GTK benötigen würde.

Java wird zudem beim Start automatisch auf die vorhandene Hardware optimiert, was im allgemeinen eine bessere Performance zur Folge hat als wenn mit C++ versucht wird zu optimieren, da der kaum für alle möglichen Architekturen optimiert wird, sondern nur für eine Spezifische Architektur. Als grosser Nachteil von Java ist dafür der lange Start zu nennen, wären dieser Zeit werden dann unter anderem die Optimierungen vorgenommen. Zudem brauchen Java Applikationen aufgrund des Overhead der VM mehr Arbeitspeicher. Dafür kriegt man bei Java bei jedem Absturz / Fehler einen Stacktracke, während man sich bei C++ aufgrund eines fehlenden Speicherschutzes unter Umständen sogar den Stack zerstören kann, was eine sehr schwirige Fehlesuche zur Folge hat, da der Fehler grundsätzlich nach dem Absturz nicht lokalisiert werden kann, und daher mühsam Stückweise getestet / Analysiert werden muss. Festplattenspeicher wird nicht berücksichtigt, da in den heutigen Zeiten dieser sowieso im Überfluss vorhanden ist, und ein Vergleich schwierig ist, da Libraries / Runtime auch mit eingerechnet werden müssten.

Als Entwicklungsumgebung kamen Sowohl Eclipse als auch IntelyJ zum Einsatz, für beide IDEs sind Projektfiles vorhanden. Es kam JDK 1.6 (Java 6) zum Einsatz, JDK 1.7 (Java 7) ist noch nicht soweit verbreitet, und würde der Applikationen keine Vorteil bringen.

3.2.2. Markup Language

Was ist eine Markup Language?

Warum dieser Weg?

Als Markup Language für die Simulation kam der Matlab Syntax zum Einsatz, der ebenfalls vom Open Source Tool «octave» verarbeitet werden kann.

3.2.3. Plugin Handling

Die Applikation hat einige Punkte, die sinvollerweise durch Plugins erweitert werden können. Es gibt bereits fertige Frameworks die Pluginhandlings ermöglichen, eines der bekannten ist Eclipse mit dem OSGi. Dieses basiert auf Extension Points und Extensions, welche wider Extensio Pointes bereitstellen können. Die komplette Applikation wird dann als Plugin aufgebaut, Abhängigkeiten können spezifiziert werden und Seiteneffekte können vermieden werden durch die Abschottung einzelner Plugins gegeneinander. Sogar das mehrfache Laden einer Library in verschiedenen Versionen ist ohne Probleme möglich.

Für unsere Applikation ist soviel Flexibilität aber nicht nötige, zudem würde dies auch einen hohen mehraufwand bei der Implementation bedeutet.

Unsere Pluginimplementation funktioniert daher viel einfacher:

1. Es wird ein Interface für ein Plugin definiert, dies geschieht in der Hauptapplikation

2. Ein Plugin implementiert diese Schnittstelle. Zudem wird ein XML File erstellt, indem steht
    a) Der Name des Plugins
    b) Eine Beschreibung
    c) Der Autor
    d) Die Klasse die geladen werden soll beim Start des Plugins
3. Der Javacode und das XML File werden in ein .jar gepackt und in einem vordefinierten Ordner abgelegt (meist konfigurierbar in config.properties)
4. Die Applikation durchsucht beim Start diesen Ordner, öffnet alle .jar Dateien und lädt das plugin.xml File. Hat dies alles geklappt wird das Jarfile in den aktuellen Kontext geladen und die Klasse ausgeführt
5. Es gibt keine Querabhängigkeiten, das Plugin wird direkt in den Applikationkontext geladen, und hat vollen Zugriff auf die Applikation

Es sind momentan zwei Pluginschnittstellen vorhanden

1. Laden von Fremdformate: Plugins für Berkeley Madonna und Dynasys sind vorhanden, siehe 3.2.4
2. Simulation: Plugins für Interne Simulation und Matlab Codegeneration: TODO Querverweis

3.2.4. Dateiformat

Als Dateiformat wird ein eigens entwickeltes, auf XML basierendes Dateiformat verwendet. Dieses Dateiformat spiegelt die Internen Strukturen ziemlich genau ab. Zudem gibt es Plugin Schnittstelle um Fremdformate zu importieren, es ist somit möglich, beliebige Fremdformate zu importieren. Es muss dafür lediglich ein Plugin erstellt werden und im richtigen Verzeichnis abgelegt werden.

Fremdformat importe Es wurden zwei Importplugins erstellt, eins für Berkeley Madonna, uns eins für Dynasys. Um einen sinnvollen Test der Pluginschnittstelle vorzunehmen waren zwei Plugins notwendig.

Dynasys ist seit 2009 unter der GPL freigegeben, somit war ein Einblick in den Pascal Code möglich. Die Pascal Objekte wurden binär serialisiert, und da es sich bei Dynasys noch um eine 16bit Applikation handelt werden nicht alle Daten korrekt eingelesen. Trotz Einschränkungen ist es somit möglich kleinere Dateien komplett, und grössere Dateien teilweise einzulesen.

Bei Berkeley Madonna war kein Quellcode verfügbar, jedoch basiert der grafische Modellierungsteil von Berkeley Madonna auf Java. Die Datei an sich ist binär, enthält jedoch die Magic Bytes 0xACED, welches von Java als Start für eine Object Stream verwendet werden. Der Stream enthält nur primitive Datentypen, und Objekte aus der Java Runtime, es müssen also keine spezifischen Klassen vorhanden sein. Mit Beispieldateien war es möglich das Format zu reverse engeneeren. Es wird nicht alles korrekt importiert, es konnten jedoch die aus der Physikvorlesung vorhandenen Dateien korrekt importiert werden.

Die Nächste Version von Berkeley Madonna, Version 9, verwendet nun aber ein neues, XML basierendes Format, welches nicht implementiert wurde. Es könnte jedoch auch dieses Format mit einem zusätzlichen Plugin eingelesen werden.

TODO => http://docs.oracle.com/javase/6/docs/platform/serialization/spec/protocol.html

Internes Dateiformat Die von (AB)² Simulation erstellen Dateien enden auf .simz, es handelt sich dabei um ein standard ZIP-File, welches 4 Files enthält:

- configuration.xml
- mimetype
- simulation.xml
- version

Version ist ein Java Property File, mit zwei Einträgen:

version=1

compatible=1

«version» ist ein Ganzzahlwert, die Version die Datei, «compatible» ist ebenfalls eine Ganzzahl, die angibt mit welcher Version die Datei immer noch eingelesen werden kann, ohne das Probleme auftreten. Kleinere Unstimmigkeiten werden in Kauf genommen, aber das Modell kann immer noch eingelesen werden, somit ist die Rückwärtskompatibilität genau geregelt.

Wird eine Inkompatible Änderung am Dateiformat vorgenommen, so müssen beide Versionsnummern um eins hochgezählt werden, wird eine kompatible Erweiterung am Dateisystem vorgenommen so wird lediglich «version» hochgezählt.

mimetype ist ein Textfile mit einem einzigen String, «application/zhaw.simulation.project». Diese Datei wird oft verwendet um den Typ eines auf Zipbasierenden Dateiformats zu erkennen. Zum Beispiel verwendet .odt als mimetype «application/vnd.oasis.opendocument.text», dank solchen Kennungen kann Zipfile eindeutig als gültiges Simulationsfile erkennen.

configuration.xml enthält die Konfiguration der Plugins, z.B.

Listing 1: configuration.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <simconfig>
3     <double name="simulation.dt" value="0.1"/>
4     <double name="simulation.end" value="5.0"/>
5     <double name="simulation.start" value="0.0"/>
6 </simconfig>
```

Diese Einstellungen können von den Plugins beliebig festgelegt werden, hier sind nur Einstellungen gespiechert, und keine Business Daten.

simulation.xml Enthält das Modell, oder bei einem XY-Model auch mehrere Modelle. Diese Datei sieht folgendermassen aus (gekürzt, kommentiert):

Listing 2: simulation.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <simulation>
3     <model grid="20" height="400" type="xy" width="400" zerox="200" zeroy="200">
4         <meso derivative="FIRST_DERIVATIVE" directionx="grad(&amp;quot;d1&amp;quot;, &
              amp;quot;x&amp;quot;)" directiony="1" name="m0" value="model3" x="151" y="89"
              />
5         <meso derivative="FIRST_DERIVATIVE" directionx="sin(1/2)" directiony="cos
              (1/2)" name="m1" value="model3" x="10" y="88"/>
```

```
6          <meso derivative="FIRST_DERIVATIVE" directionx="1" directiony="1" name="
               m5" value="model2" x="256" y="281"/>
7          <global name="g0" value="5" x="252" y="37"/>
8
9          <density name="d1" text="" value="10*sin(x/20)+10*cos(y/20)"/>
10         <density name="d2" text="" value="x"/>
11
12        <model name="model2" type="flow">
13             <container name="Q" value="0" x="262" y="109"/>
14             <parameter name="UC" value="Q/C" x="220" y="238"/>
15
16            <!-- Fluss, <helperpoint> sind die Bezier Hilfspunkte -->
17            <flowConnector name="I" value="UR/R">
18                 <source>
19                     <helperpoint x="116" y="147"/>
20                     <infinite x="34" y="116"/>
21                 </source>
22                 <target to="Q">
23                     <helperpoint x="230" y="145"/>
24                 </target>
25                 <valve x="140" y="118"/>
26             </flowConnector>
27
28             <!-- Parameter Verbindung -->
29             <connector from="Q" to="UC">
30                 <helperpoint x="256" y="207"/>
31             </connector>
32         </model>
33         <model name="model3" type="flow">
34             <!-- Anderes Submodel... -->
35         </model>
36     </model>
37 </simulation>
```

Das Dateiformat ist weitgehend selbsterklärend. Es wurde erstellt um das Modell unserer Applikation möglichst genau abzubilden. Es wurde nicht angestrebt einen Standard zu erschaffen / mit anderen Applikationen über dieses Format zu interagieren.

Grundsätzlich gilt: jeder XML Tag entspricht einem Objekt, jedes Attribut eines Tags entspricht einem Attribut eines Objekts (einfaches Attribut wie ein String). Jeder Tag innerhalb eines anderen Tags entspricht einem Attribut eines Objekts (komplexes Attribut, z.B. Punkt mit zwei Koordinaten). Somit entspricht das Format nicht ganz den XML Richtlinien, welches alle relevanten Informationen als Tags und alle Metainformationen als Attribute abspeichert, ist aber dem Code sehr nahe und auch einfach zu verstehen.

4. Vorgehen

4.1. Softwarearchitektur

Projektunterteilung

Die Applikation besteht aus mehreren Javaprojekten, die hier in einer Übersicht kurz beschrieben werden

1. **AppDefinition:** Dieses Projekt beinhaltet Interfaces der Applikation, die nicht im Projekt

«Simulation» untergebracht werden konnten, da ansonsten eine Zyklische Abhängigkeit entstanden wäre

2. **BJavalibs:** Generelle Java GUI Libraries von Andreas Butti, dieser Code ist grösstenteils vor der BA entstanden
3. **Editor:** Abstrakter Editor für XY- und Flow Modelle. Beinhaltet alle basis Klassen, Globale Parameter, Clipboard, Undo / Redo, Toolbar / Menubar und den Formeleditor
4. **Editor.Flow:** Der Editor für Flussdiagramme, basierend auf «Editor»
5. **Editor.XY:** Der Editor für XY-Diagramme, basierend auf «Editor»
6. **ExternLibraries:** Dies ist nur ein Pseudoprojekt, und beinhaltet kein Code. Hier sind unsere externen Libraries untergebracht.
   a) JFreeChart: Diagramm Library
   b) JCommon: Wird von JFreeChart benötigt
   c) JXLayer: Wird verwendet um beim Simulieren den Editor Unscharf darzustellen und ein Statusdialog direkt im Editor darzustellen
   d) SwingX: Erweiterte Swing GUI-Komponenten, werden an diversen Stellen benötigt
7. **ImageExport:** Der «Speichern als Bild» Dialog
8. **ImportFilter:** Definition für Importplugins
9. **ImportFilter.Dynasys:** Dynsys Import Plugin
10. **ImportFilter.Madonna:** Berkeley Madonna Import Plugin
11. **Model:** Das «Domänenmodell», dies ist die interne Abbildung aller Daten die modelliert werden können.
12. **NetbeansDirchooser:** Dieser Code entstammt dem Netbeans Projekt, es ermöglicht die komfortable auswahl eines Ordners, die mit Swing Boardmittel wesentlich weniger konfortabel wäre: Es können die Ordner wie gewohnt als Baum ausgeklappt werden.
13. **OnscreenKeyboard:** Hierbei handelt es sich um die Tastatur für die Spezialzeichen, die bei Feldern im Diagramm zur Verfügung steht.
14. **Plugin:** Dies ist die Implementation um Plugins zu laden. Diese 285 Zeilen Code stammen aus einem vorgängigen Projekt von Andreas Butti
15. **Simulation:** Dies ist der Einsprungpunkt der Applikation. Hier befinden sich auch die globalen Komponenten
    a) About Dialog
    b) Einstellungen
    c) Speichern / Laden
16. **SimulationBuild:** Dies ist ein Pseudoprojekt das nur für den Build der Applikation benötigt wird, es enthält kein Code

17. **SimulationDiagram:** Dieses Projekt enthält die Darstellung eines Diagramms, obwohl das eigentliche Diagramm von JFreeChart dargestellt wird gibt es einiges um das Diagramm, wie die Legende oder die Konfiguration, welche in diesem Projekt zu finden sind

18. **SimulationJepLib-2.4.1:** JEP ist ein Open Source Parser, welcher für die Interne Simulation und das Prüfen von Formeln eingesetzt wird. Da das Projekt offiziell nicht mehr supported wird wurde der Sourcecode komplett kopiert um kleine Fehlerkorrekturen vorzunehmen.

19. **SimulationPlugin:** Die Definition des Simulationplugins

20. **SimulationPlugin.Intern:** Interne Simulation, die Formeln werden mit JEP berechnet. Das Plugin unterstützt nur das Flow-Modell, das XY-Modell ist nicht implementiert. Es wird nur Runge-Kutta und Euler unterstützt.

21. **SimulationPlugin.MatlabOctave:** Dieses Plugin generiert den Matlab Markup. Ohne dieses Plugin ist die Simulation von XY-Modellen nicht möglich.

22. **SimulationSidebar:** Hier befinden sich Definitionen für die Sidebar der Applikation, diese mussten in ein eigenes Projekt ausgelagert werden um keine Zyklischen Abhängigkeiten zu Produzieren, gehören jedoch eigentlich zum Editor.

23. **Sysintegration:** Hier werden die Plattform / Systemabhängigen Komponenten abgelegt, dies beinhaltet unter anderem

    a) Alle Icons (bei der aktuellen Implementation wird jedoch nicht nach Plattform unterschieden)

    b) Bookmark Implementationen (z.B. die Auswahl «Desktop» beim Speichern eines Bildes stammt von hier)

    c) Toolbar Implementation: Die von Swing bereitgestellte Toolbar sieht auf Mac OS X komisch aus, daher hier eine angepasste Version. Zudem enthält diese Implementation einige für die Simulation zugeschnittene Anpassungen.

24. **VectorExport:** Hier enthalten ist die Library Freehep und einige Helferklassen. Mithilfe dieses Projektes werden beim Bilderexport SVG, EMF und EPS Dateien erzeugt.

25. **XYResultViewer:** Bei diesem Projekt handelt es sich um das «Diagramm» von XY-Simulationen. Es kann die Bewegungen und die Dichten darstellen. Im Gegensatz zur Darstellung von Flow-Simulationen müssen hier 3 Dimensionale Daten dargestellt werden. Die dritte Dimension ist hier die Zeit, und wird durch einen Slider dargestellt.

Die Aufteilung in 25 Projekte erscheint auf den ersten Blick etwas unübersichtlich, auf den zweiten Blick entsprechen die meisten Projekte aber einzelnen Komponenten der Applikation die auch in der Applikation entsprechend aufgeteilt sind, und machen daher Sinn. Lediglich 2 Projekte wurden aus technischen Gründen (zyklische Abhängigkeiten) erstellt, und entsprechen keinen eigenen Komponenten.

Design Pattern

Die Applikation wurde nach MVC (Model View Control) bzw. Domänenmodell aufgebaut. Das Bedeutet das die Daten, die Logik und der View nur lose gekoppelt sind, und somit auch der View ersetzt werden könnte, ohne das der Rest angepasst werden müsste. Diese beiden Design Pattern beschreiben ein ziemlich ähnliches vorgehen, wobei MVC sich hauptsächlich auf einen GUI Komponenten beziehen lässt, wie z.B. ein TreeView, während das Domänenmodell sich auf eine komplette Applikation anwenden lässt. Beides sind Vorgehensmuster, und keine exakten Vorgaben.

Das (Domänen)Modell befindet sich im Projekt «Model» und ist entsprechend von allem anderen entkoppelt. Die Simulation ist nur vom Model abhängig, und es besteht keine direkte Verbindung zur GUI. Die GUI selbst ist ein relativ grosser Teil der Applikation, und ist auch nicht überall klar abgegrenzt.

Komponeten wie Undo / Redo sind bei uns komplett in der GUI untergebracht, da diese auch komplett von dieser abhängig sind. Designtechnisch ist die Einordnung solcher Komponenten nicht eindeutig, denn z.B. Undo / Redo enthält ein Modell, nämlich die letzten Änderungen. Es enthält zudem Logik, es kann Änderungen Rückgängig machen / Wiederherstellen. Es ist aber ein GUI Komponent, denn es ist 100% von der GUI abhängig. Wenn wir jedoch definieren das es sich bei den Undo / Redo Daten nicht um Business Daten, sondern um Metadaten handelt sollte es auch nach den Pattern kein Problem sein wenn wir es Innerhalb der GUI unterbringen. Zudem war dies die einzig technisch sinvolle Möglichkeit.

4.2. Technische Beschreibung Softwarekomponenten

Model

Das Datenmodell ist wie folgendermassen aufgebaut: Der Einsprungpunkt ist das «Simulation-Document», welches entweder ein «SimulationFlowModel» oder ein «SimulationXYModel» beinhaltet. Sowohl das Flow als auch das XY Modell erben von , «AbstractSimulationModel», dieses beinhaltet «AbstractSimulationData», welches ebenfalls eine Abstrakte Klasse für alle Simulationskomponenten darstellt, wie

- Container
- Parameter
- Global
- Meso Kompartment.

«SimulationFlowModel» enthält zusätzlich noch Verbindungen, welche entweder ein Fluss oder eine Parameterverbidung darstellen. Abgebildet werden diese als «AbstractConnectorData<?>». Wobei der Fluss «FlowConnectorData» und der «ParameterConnectorData» die beiden möglichen Implementationen darstellen.

«SimulationXYModel» enthält zusätzlich zur Basisklasse noch Dichten, welche von der Klasse «DensityData» sind, hier besteht keine Abstraktion, es gibt nur eine Implementation.

Zudem kann ein XY Model beliebig viele Flow Modelle enthalten, welche über «SubModelList» gehandelt werden.

Die Modelle enthalten jede Menge an Methoden zur Manipulation der Daten, welche hier nicht aufgeführt werden, diese können der Codedokumentation oder dem Code entnommen werden.

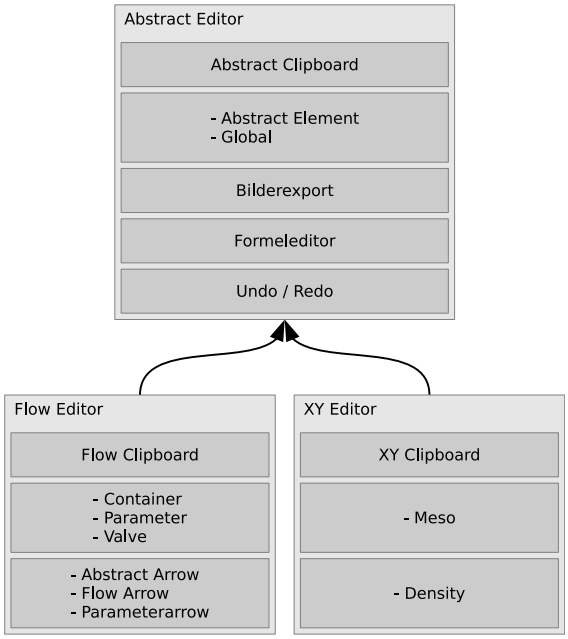

Abbildung 4.1: Übersicht Editoraufbau

Editor

Der Editor ist einer der komplexeren Elemente dieser Applikation.

Der Editor besteht aus mehreren Komponenten, grob aufgelistet in der Grafik.

Der eigentliche Editor Component ist ein eine Ableitung von «AbstractEditorView», entweder «FlowEditorView» oder «XYEditorView». Diese Komponenten haben ein paar ungewöhnliche Eigenheiten. Die Methode paint ist überschrieben, und beim Zeichen werden alle Aktionen selbst koordiniert. Dies ist notwendig um Pfeile, die selbst kein JComponent sind korrekt abzubilden. Pfeile können nicht als JComponent abgebildet werden, da ein JComponent immer rechteckig ist, und unsere Pfeile dies im Normalfall nicht sind. Bei der Darstellung ist dies noch kein Problem, aber der Pfeil würde dann auf der gesamten Fläche auch alle Mouse-Events konsumieren, diese müssten dann weitergeleitet werden wenn nötig etc. Darum haben wir für die Pfeile eine Architektur gewählt die Swing Architektur bricht, da wir eine Darstellung gewählt haben die unüblich ist und daher von Swing so nicht unterstützt wird.

Auch die Selektion wird nach dem Zeichen aller Komponenten einfach überzeichnet. Es gibt zudem die Möglichkeit das Komponenten einen Schatten zeichnen, dies wird angewendet um Abhängigkeiten von / zu Globalen darzustellen. Wir eine Globale angewählt werden alle abhängigen Elemente mit einem Schatten versehen.

Als Layoutmanager kommt «SimulationLayout» zum Einsatz, welches alle «AbstractDataView<?>» korrekt platziert. Alle anderen Komponenten werden nicht umplatziert, dies bedeutet wenn etwas anderes angezeigt wird muss dieses manuell mit «setBounds» platziert werden, was auch verwendet wird z.B. bei der Pfeil-erstell-UI (TODO Name?).

Die Clipboard implementation ist Architekturbedingt in mehrere Klassen aufgeteilt, unterstützt unter anderem das Kopieren als Rastergrafik. Das Exportieren als Vektorgrafik konnte nicht erfolgreich implementiert werden, da dies je nach Betriebsystem anders gehandhabt wird, und damit es z.B. unter Windows zuverlässig funktioniert muss eine Vektorgrafik (EMF / WMF) eingebettet in einem RTF exportiert werden, während unter Linux eher ein SVG direkt in die

Zwischenablage kopiert wird. Alle Simulationsinternen Datenstrukturen werden in die Datenstruktur «TransferData» abgefüllt. Zudem werden alle Objekte mit IDs versehen, damit nach dem einfügen die Pfeile wider verbunden werden können.

Der Bilderexport gestaltet sich relativ einfach, dank der verwendeten «freehep» Library wird einfach mit dem Graphics von «freehep» «paint» aufgerufen, den eigentlichen Export übernimmt die Library.

Undo / Redo war und ist einer der umständlichen teile der Implementation, denn es muss für jede Aktion die der User vornimmt manuell eine Implementation vorgenommen werden, die diese Aktion auch wider Rückgängig machen kann. Dies ist auch beim erweitern der Applikation zu bedenken, denn wenn dies vergessen geht ist die Konsistenz nicht mehr gewährleistet, was zu Fehlern führt. Für das Debugging des Undo / Redo Mechanismus musste daher eine kleine Hilfe implementiert werden. Wir die Applikation mit dem Parameter "–debug-undo" aufgerufen erscheint ein Dialog der alle Aktionen des aktuellen Undo / Redo Handlers live auflistete, und je nach Status einfärbt. Zu bedenken ist, dass nach dem erstellen einer neuen Datei ein neuer Undo / Redo Handler erstellt wird, und der Dialog daher nicht mehr funktioniert.

Errorhandling / Logging

Bei Serverapplikationen ist es üblich das alle Aktionen geloggt werden, bei GUI Applikationen ist es weniger üblich, daher haben wir uns entschieden nur die Fehler zu loggen. Andere Ausgaben werden direkt auf STDOUT geschrieben, und sind somit nur lesbar wenn dieser entweder umgeleitet oder $(AB)^2$ Simulation direkt auf einer Konsole gestartet wird. Dies ist jedoch nur bei der Entwicklung sinnvoll, alle Meldungen die für den Benutzer relevant sind werden als Popups ausgegeben.

Ein wichtiger Punkt, der oft vergessen geht, ist, dass die Exceptions im Eventloop auch abgefangen werden müssen. Dies geschieht mit "Errorhandler.registerAwtErrorHandler();" somit wird unserer eigener Errohandler als Eventloop errorhandler registriert, und kann somit alle Fehler die nicht gecatcht sind und im Eventloop auftreten abfange, dem User anzeigen und loggen.

Fehler die zu erwarten sind werden in unserer Applikation abgefangen, und dem Benutzer eine Entsprechende Meldung angezeigt. Zu erwartende Fehler sind z.B. Ungültige Benutzereingaben, nicht genügend Schreibrechte oder Fehler bei Formeln wie log(-1). Unerwartete Fehler sind z.B. IOException aufgrund eines kaputten Datenträgers, manipulierte Simulationsdateien etc.

Alle zu erwartenden Fehler werden den Benutzer mit «Messagebox.showError» dargestellt, während unerwartete Fehler mit «Errorhandler.showError(e, <optionale Beschreibung>);» dargestellt und automatisch geloggt werden. Der Pfad für das Logging wird in der Konfigurationsdatei festgelegt.

GUI Konsistenz

Es wurde viel Wert gelegt das die GUI konsistent erscheint. Dabei wurde das Benutzerhandling an Linux / OS X angelehnt.

- Konfigurationsdialoge wie die Einstellungen / Formeldialog / Diagrammeinstellungen enthalten keine "OK", "Übernehmen", "Abbrechen" Buttons, sondern alle Änderungen werden sofort übernommen. (Teilweise werden die Änderungen auch erst beim Schliessen des Dialoges gespeichert, was aber für den User irrelevant ist)

- Jegliche Messageboxen werden von der Klasse «Messagebox» dargestellt, welche den Defaultbutton grösser, fett und immer ganz rechts darstellt.

- Alle Layouts werden mit Layoutmanager gelayoutet, dies bedeutet wenn ein Benutzer z.B. eine grössere Schrift hat ist dies kein Problem, es können alle Dialoge vergrössert werden und der Inhalt wird auch vergrössert. Einzige Ausnahme ist der Editor: Das Simulationdokument wird nicht "gezoomt" wenn der Editor vergrössert wird.
- Es wird nur die nötige Konfiguration angeboten: Nur die Einstellungen die auf jedem System verschieden sind / Von externen Parametern abhängig sind werden angeboten. Globale sind grün und Parameter gelb. Es macht keinen Sinn solche Dinge konfigurierbar zu machen, das verwirrt die Benutzer höchstens.
- Es werden keine "Sind Sie sicher?" Meldungen angezeigt. Es können aber alle Aktionen die das Dokument verändern Rückgängig gemacht werden (solange der Editor offen bleibt) somit sind solche Nachfragen nicht nötig / sinnvoll.
- Fehler bei der Datenvalidierung werden grafisch hervorgehoben, oder es ist erst gar nicht möglich ungültige Werte einzugeben.

Konfiguration / Einstellungen (Backend)

Beim Start der Applikation wird eine Datei ./config/config.properties eingelesen, diese Datei muss existieren, ansonsten werde Fehler auftreten. In dieser Datei sind alle Konfigurierbaren Einstellungen gespeichert, die der Benutzer aber nicht umstellen soll / muss / kann. Diese sind:

- portable=[true | false]: Wenn portable = true ist die Applikation Portabel, es werden alle Einstellungen relativ zur Applikation gespeichert. Ist die Einstellung false werden alle Einstellungen im Betriebsystemspezifischen Ordner abgelegt (es gibt keine Registry Einträge unter Windows)
  - Linux: ~/.config
  - Mac ~/Library
  - Windows "C:\Users" or "C:\Document and Settings"
- errorlogPath=errorlog/: Der Pfad wo das Fehlermeldungen abgelegt werden (ggf. relativ zur, je nach «portable»)
- settingsPath=config/: Der Pfad für die Einstellungen (ggf. relativ zur, je nach «portable»)
- importPluginFolder=plugin/import/: Immer relativ zur App, wo die .jar's oder .xml's gesucht werden um die Plugins zu laden, hier die Importplugins für Fremdformate.
- simulationPluginFolder=plugin/simulation/: Siehe oben; Hier die Plugins zur Simulation
- predefinedDash: Die Liniendefinitionen für das Diagram. Getrennt mit Semikolon (;). Durchgezogene Linie: "" (leerer String, wobei in der Konfiguration keine Anführungszeichen angegeben werden.), 5 Punkte gestrichelte Linie: "5 5"
- defaultsFonts: Die Fonts die verwendet werden für das Diagram, die erste Schrift, die installiert ist, der mit Semikolon (;) separierter Liste wird verwendet
- keyboard...: Definition des Sonderzeichen On-Screen-Keyboard, wird verwendet um bei den Diagrammen in den Namen Sonderzeichen einzugeben. Parameter definieren das Aussehen

der Tastatur und werden hier nicht alle Im Detail erläutert. Die Zeichen die Angezeigt werden sind hier definiert, man beachte die Unicodeschreibweise (wird z.B. von Eclipse automatisch escaped): keyboard.keys=\u03B1;\u03B2;\u03B3;...

Im Settingsordner werden mehre Dateien abgelegt, alle Einstellungen die von der Applikation und den Plugins definiert werden sind in der Datei "settings.ini" abgelegt. Es ist nicht vorgesehen das die Datei von Hand editiert wird. Sollte Probleme mit den Einstellungen auftreten kann die Datei einfach gelöscht werden, es werden dann die Defaulteinstellungen verwendet.

Die Dateien endend auf ".windowPos" Speichern die Position der Fenster, um nach dem nächsten Start der Applikation alle Fenster wider an der gleichen Position anzuzeigen.

Dies kann zu probleme führen, z.B. wenn die Bildschirmkonfiguration verändert wurde / Beamer an / abgesteckt wurde. Um solche Probleme zu lösen, können einfach alle Dateien endend auf ".windowPos" gelöscht werden, dann werden beim nächsten Start alle Fenster auf dem Hauptbildschirm zentriert angezeigt.

Numerische Lösungsverfahren

4.3. Tests und Validierung

Test der Matheengine, Vergleich mit Berkeley-Madonna

5. Resultate

Ergebnisse

5.1. Numerische Lösungsverfahren

5.1.1. Diffusionsgleichung

Um die Implementierung der numerischen Diffusionsgleichung zu überprüfen
Beschreibung, Screenshots, Beispielsimulationen mit Ergebniss-Diagramm

6. Diskussion und Ausblick

Markup Language

Wir haben uns die Matlab Markup Language entschieden, da wir auf viele vordefinierten Funktionen ("Toolbox") zurückgreifen konnten, und uns somit den Mathematischen Teil teilweise vereinfachen.

Es haben sich jedoch immer wieder Probleme mit den in Matlab vordefinierten Funktionen ergeben, grundsätzlich erfüllen die Funktionen unsere Anforderungen. Dies haben wir auch vorgängig abgeklärt, jedoch sind bei der Verwendung Probleme aufgetreten, die die Verwendung der Toolboxfunktionen verunmöglichten:

- Numerische Integration: Es kann zwar ein Integral numerisch integriert werden, jedoch müssen wir mehrere Integrale Simultan lösen, was von der Matlab Toolbox nicht unterstützt wird

- Gradienten berechnen mit "grad": Wird ein Gradienten im diskreten Raum berechnet, so tritt immer das Problem auf das die Zahlen in einem Gitter abgelegt werden müssen, was mathematisch natürlich falsch ist. Matlab versucht den Mathematischen Fehler gleichmässig auf alle Richtungen zu verteilen, dies bedeutet das wenn eine Spitze abgeleitet wird, so entsteht ein Plateu. Diese Plateus sind für unsere Simulation extrem ungünstig. Daher haben wir jetzt einen Gradienten implementiert der einfach um 0.5 Einheiten im Raster verschoben ist. Dies bewirkt zwar ein Abdriften in eine Richtung, dies stört aber die Simulation weniger als ein Plateu. Ein Plateu kann bewirken das sich ein Meso Kompartment, welches sich zur nidrigsten Konzentration bewegen soll, nicht mehr bewegt.

Wir erstellen nun den kompletten Matlab Code selbst, es werden keine Toolboxen verwendet. Nach dem erstellen eines Temporären Files wird ein neuer Matlab / Octave Prozess erstellt, und dieser führt die Simulation aus.

Leider hat sich das Fehlerhandling viel schwieriger gestaltet als erwartet, denn z.B. Matlab beendet sich nicht nach einem Fehler mit einem entsprechenden Exit Code, sondern geht in den Interaktiven Modus über.

Zudem ist die Plattformunabhängigkeit nicht gewährleistet, denn es muss für jede Plattform separat getestet werden, was wir auch gemacht haben und festgestellt haben das es nicht auf allen Plattformen funktioniert.

NICHT DER FALSCHE WEG aber schwierig

Aufgrund aller dieser Tatsachen haben wir entschieden, dass der Weg mit einer Markup Language der falsche war:

- Fehlerhandling schwierig
- Plattformunabhängigkeit funktioniert nur bedingt
- Keine Programmiertechnischen Vorteile
- Geschwindigkeitseinbussen (bei kleinen Simulationen fällt insbesondere der Start einer externen Applikation stark ins Gewicht)

Wir haben daher entschieden das eine zukünftige Entwicklung nicht auf dieser Basis, sondern der Code komplett intern ausgeführt wird. Daher haben wir auch keine Zeit mehr ins Fehlerhandling / Plattformunabhängigkeit etc. investiert. Eine Interne Simulation bringt zusätzlich folgende Vorteile:

- Eine Formel, die bei der Eingabe vom Parser validiert wurde ist auch zu 100%ig sicher ausführbar, wenn eine Formel mit dem Internen Parser validiert wurde und dann als Matlab Code ausgegeben wird ist dies nicht sicher, denn die Parser unterscheiden sich in Details, die unter Umständen nicht alle korrekt gehändelt werden. Zudem wird jede Formel beim internen Parser in einem separaten Kontext ausgeführt, was bei Matlab nicht der Fall ist, somit können unter ungünstigen Umständen Seiteneffekte auftreten, welche beim Internen Parser praktisch ausgeschlossen sind.
- Fehler sind immer gehändelt, und können normalerweise Ihren Ursprungsobjekt zugeordnet werden (z.B. einem Container), wenn im Matlab Code ein Fehler auftritt kann dieser nur einer Zeile zugeordnet werden, es ist jedoch für den Anwender nicht offensichtlich wo der Ursprung des Fehlers liegt.

- Die Simulation kann einfach abgebrochen werden: Wenn die externe Simulation abgebrochen werden muss ist dies wesentlich komplizierter, und funktioniert ggf. nicht unter allen Umständen.
- Parallelisierung: Heutzutage besitzt jeder moderne PC mehr als einen Rechenkern. Mit Matlab / Octave ist es jedoch nicht möglich dies zu nutzen. Bei einer Internen Simulation wäre dies möglich, wenn auch aufwendig. Somit könnte die Simulation grösserer Modelle um Faktoren beschleunigt werden. (Voraussetzung: mehr als ein Core, und das Modell muss entsprechend unabhängige Teile aufweisen, ein Aufteilen eines einzelnen Integrals ist theoretisch zwar möglich, praktisch ist es jedoch langsamer als die Simulation auf einem CPU, da die Synchronisation sehr viel Leistung benötigt) => MPC

TODO

Was Fehlt noch, was muss noch gemacht werden? Interpretation und Validierung der Resultate Rückblick auf Aufgabenstellung, erreicht bzw. nicht erreicht Legt dar, wie an die Resultate (konkret vom Industriepartner oder weiteren Forschungsarbeiten; allgemein) angeschlossen werden kann; legt dar, welche Chancen die Resultate bieten 18.

Container können nur positiv und negativ werden, nicht nur positiv, zumindest nicht im Matlab Plugin.

7. Literaturverzeichnis

[Dan09] Dankert, Jürgen: *Numerische Integration von Anfangswertproblemen.* 2009

[HB06] Hanke-Bourgeois, Martin: *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens.* Bd. 2. Auflage. Teubner, 2006. – 551–628 S.

[Kos94] Kost, Arnulf: *Numerische Methoden in Der Berechnung Elektromagnetischer Felder.* Springer-Lehrbuch, 1994. – 20–38 S.

[Obe08] Oberle, Hans J.: *Numerik gewöhnlicher Differentialgleichungen.* 2008

[Sch11] Scheidegger, Stephan: *Modelle in der medizinischen Biophysik.* 2011

Teil I.
Anhang

8. Codeanmerkungen

Tabwidth: 4

Encoding: UTF8: Wichtig! Ansonsten funktionieren Spezialzeichen wie x-Punkt und x-Punt-Punkt nicht.

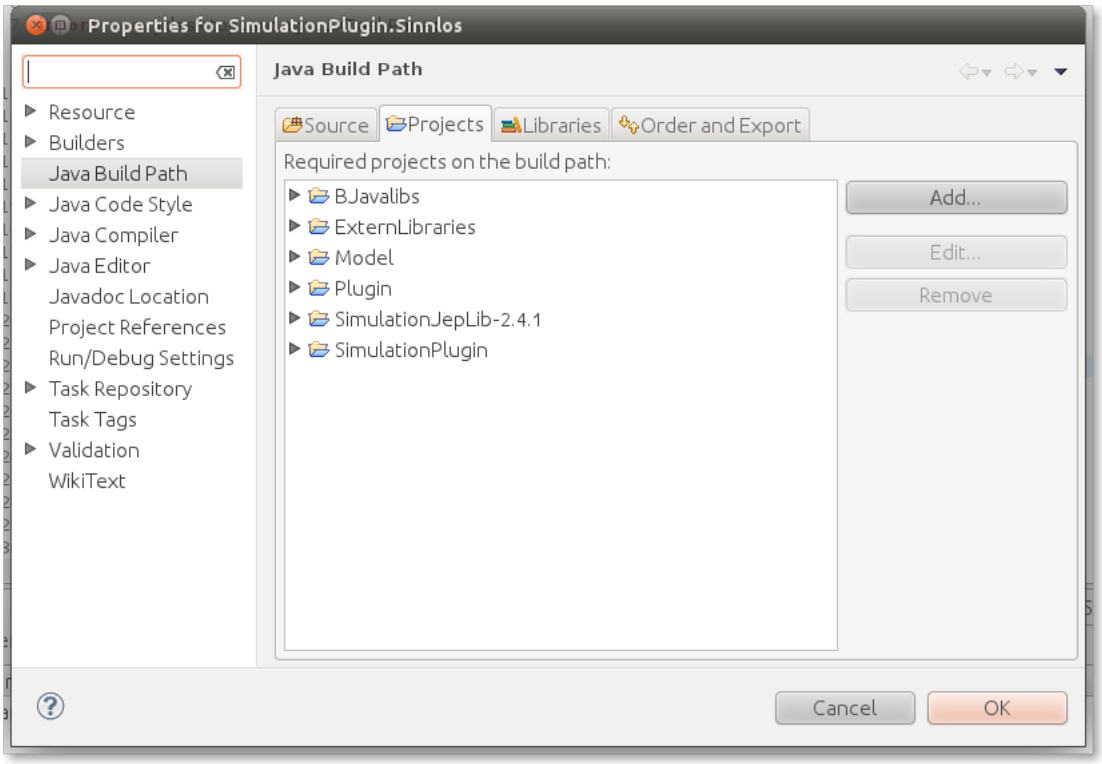

Abbildung 10.1: Java Buildpath

9. Glossar

Meso-Kompartment: Ein Teil des XY-Simulationsmodells
Swing: Java GUI Framework

10. Weiterentwicklung

Hier wird anhand von Beispielen beschrieben wie die Applikation weiterentwickelt werden kann. Die Beispiele an sich sind erfüllen nur den Zweck der Illustration. Die Anleitung bezieht sich auf Eclipse unter Linux, es ist jedoch nicht relevant welche IDE das verwendet wird. Java Kenntnisse sind erforderlich, es wird davon ausgegangen das alle Projekte der Simulation bereits importiert wurden und fehlerfrei kompilieren (Innerhalb der IDE, Ant ist für diese Beispiele nicht erforderlich, da Eclipse, InteliJ, Netbeans und alle modernen IDEs automatisch den Javacode kompilieren).

10.1. Erstellen eines neuen Simulationsplugins

Es wird ein neues Javaprojekt erstellt, mit den Name «SimulationPlugin.Sinnlos». In den Projekteinstellungen wird der «Java Build Path» konfiguriert, es werden folgende Projekte referenziert:

- BJavalibs: Nötig für die Klasse Settings, welche die Einstellungen der Applikation zur Verfügung stellt. Alle Einstellungen die gelesen / geschrieben werden erhalten zusätzlich

ein prefix beim Namen. Wird also z.B. der Key “hallo” gelesen wird intern “prefix.hallo” gelesen, wobei der Prefix in diesem Falle “simplugin.<name des Plugins>” ist.

- ExternLibraries: wird benötigt für «JXTaskPane», welches von SwingX stammt. Dies ist ein Element in der Sidebar der Simulationsapplikation.
- Model: Das Model wird benötigt da es alle Daten enthält
- Plugin: Die generelle definition des Plugins, wird benötigt weil wir ein Plugin implementieren
- SimulationJepLib-2.4.1: Wird benötigt aufgrund von «SimulationModelException» (und ggf. den davon erbenden Klassen, die alle auch zur Verfügung stehen)
- SimulationPlugin: Die definition des Simulationplugins

Es wird eine neue Package erstellt, hier «ch.zhaw.simulation.sim.sinnlos» die Package muss unique sein, es dürfen nicht zwei Plugins die gleichen Package Namen verwenden, ansonsten kann es zu Problemen kommen!

Es wird eine neue Klasse erstellt: «SimulationSinnlosPlugin», welche «SimulationPlugin» implementiert. Alle erforderlichen Methoden werden in die Klasse eingefügt: Bei Eclipse «Add unimplemented methods», dies erstellt 10 Methoden:

- public boolean load() throws Exception: Die Initialisierung des Plugins beim laden, wird im Normalfall nicht benötigt.
- public void unload(): Diese Methode wird vor dem Beenden der Applikation aufgerufen. Grundsätzlich ist es möglich Plugins wärend der Laufzeit zu entladen / neu zu laden, wird jedoch hier von der Applikation nicht unterstütz.
- public JPanel getSettingsPanel(): Wird hier ein Wert != null zurückgegeben wird dieser als eigener Tab in den Einstellungen angezeigt.
- public JXTaskPane getConfigurationSidebar(SimulationType type): Wird hier ein Wert != null zurückgegeben wird das Panel in der Sidebar angezeigt. «type» gibt an ob die Konfiguration für ein XY oder für ein Flow model benötigt wird.
- public void init(Settings settings, SimulationConfiguration config, PluginDataProvider provider): Hier wird das Plugin normalerweise initialisiert. «settings» sind die globalen Einstellungen, die auf Applikationsebene gespeichert werden, wie z.B. Defaultwerte. «config» sind die Einstellungen des aktuellen Dokuments, werden gespeichert aber sind immer vom aktuellen Dokument abhängig, und nicht global! «provider» stellt Objekte zur Verfügung, die z.B. für die GUI benötigt werden:
  - public JFrame getParent(): Wenn ein Dialog angezeigt werden soll kann dies als parent verwendet werden, und der Dialog kann mit setLocationRelativeTo(parent); zentriert auf dem parent angezeigt werden.
  - public ExecutionListener getExecutionListener(): Hier werden alle Statusmeldungen übermittelt (Fehler / Progress etc.)
  - public SimulationType getSimulationType(): Hier erhalten wir nochmals den Simulationstyp (FLOW / XY)

- public void checkDocument(SimulationDocument doc) throws SimulationModelException: Prüft ob eine Simulation möglich ist. Wenn das Plugin keine Dichten unterstützt kann dies hier abgefangen werden.
- public void executeSimulation(SimulationDocument doc) throws Exception: Führt die Simulation aus
- public SimulationCollection getSimulationResults(SimulationDocument doc): Gibt die Resultate der Flow Simulation zurück, wird immer verwendet (da auch die XY Simulation Flow Simulationen enthält)
- public Vector<XYDensityRaw> getXYResults(SimulationDocument doc): Gibt die Resultate der XY Simulation zurück
- public void cancelSimulation(): Wird aufgerufen wenn die Simulation abgebrochen werden soll (von extern)

Es ist davon auszugehen das alle Daten die dem Simulationsplugin übergeben werden bereits validiert sind, also Formeln etc., daher ist keine Validierung im Plugin erforderlich. Wenn ungültige Daten übergeben werden ist dies nicht im Plugin abzufangen, sondern in der Applikation. Wir vervollständigen nun den Code mit folgendem Beispiel:

Listing 3: SimulationSinnlosPlugin.java

```
1  package ch.zhaw.simulation.sim.sinnlos;
2
3  import java.util.Vector;
4
5  import javax.swing.JPanel;
6
7  import org.jdesktop.swingx.JXTaskPane;
8
9  import butti.javalibs.config.Settings;
10
11 import ch.zhaw.simulation.math.exception.SimulationModelException;
12 import ch.zhaw.simulation.model.SimulationDocument;
13 import ch.zhaw.simulation.model.SimulationType;
14 import ch.zhaw.simulation.model.flow.SimulationFlowModel;
15 import ch.zhaw.simulation.model.flow.element.SimulationContainerData;
16 import ch.zhaw.simulation.model.simulation.SimulationConfiguration;
17 import ch.zhaw.simulation.plugin.ExecutionListener;
18 import ch.zhaw.simulation.plugin.ExecutionListener.FinishState;
19 import ch.zhaw.simulation.plugin.PluginDataProvider;
20 import ch.zhaw.simulation.plugin.SimulationPlugin;
21 import ch.zhaw.simulation.plugin.data.SimulationCollection;
22 import ch.zhaw.simulation.plugin.data.XYDensityRaw;
23
24 public class SimulationSinnlosPlugin implements SimulationPlugin {
25
26     private ExecutionListener execListener;
27
28     @Override
29     public boolean load() throws Exception {
30         // erfolgreich geladen, bei return false kann das Plugin nicht verwendet
31         // werden!
32         return true;
```

```
33      }
34
35      @Override
36      public void unload() {
37      }
38
39      @Override
40      public JPanel getSettingsPanel() {
41          // keine Einstellungen (Einstellungsidalog)
42          return null;
43      }
44
45      @Override
46      public JXTaskPane getConfigurationSidebar(SimulationType type) {
47          // keine Einstellungen (Sidebar)
48          return null;
49      }
50
51      @Override
52      public void init(Settings settings, SimulationConfiguration config,
            PluginDataProvider provider) {
53          this.execListener = provider.getExecutionListener();
54      }
55
56      @Override
57      public void checkDocument(SimulationDocument doc) throws
            SimulationModelException {
58          // Validierung immer erfolgreich
59      }
60
61      @Override
62      public void executeSimulation(final SimulationDocument doc) throws Exception
            {
63          if (doc.getType() == SimulationType.FLOW_SIMULATION) {
64              new Thread(new Runnable() {
65
66                  @Override
67                  public void run() {
68                      SimulationSinnlosPlugin.this.execListener.executionStarted("
                            Simulation gestartet");
69                      try {
70                          Thread.sleep(1000);
71                      } catch (InterruptedException e) {
72                      }
73
74                      SimulationSinnlosPlugin.this.execListener.setState(50);
75                      SimulationSinnlosPlugin.this.execListener.setExecutionMessage
                            ("Fast fertig!");
76                      try {
77                          Thread.sleep(1000);
78                      } catch (InterruptedException e) {
79                      }
80
81                      SimulationFlowModel flowModel = doc.getFlowModel();
82                      for (SimulationContainerData c : flowModel.
                            getSimulationContainer()) {
83                          System.out.println("-> " + c.getName() + " = " + c.
                                getFormula());
```

```
84                      }
85
86                      SimulationSinnlosPlugin.this.execListener.executionFinished("
                            Keine Implementation", FinishState.ERROR);
87                  }
88              }).start();
89          } else {
90              // Unerwarteter Fehler, wird von der Applikation gehändelt
91              throw new Exception("Der Programmierer hatte Feierabend!");
92          }
93      }
94
95      @Override
96      public SimulationCollection getSimulationResults(SimulationDocument doc) {
97          // keine Daten...
98          return null;
99      }
100
101     @Override
102     public Vector<XYDensityRaw> getXYResults(SimulationDocument doc) {
103         // keine Daten...
104         return null;
105     }
106
107     @Override
108     public void cancelSimulation() {
109         // Kann nicht abgebrochen werden...
110         this.execListener.setExecutionMessage("Nicht möglich!");
111     }
112 }
```

Unser Plugin ist somit fertig, aber es wird noch nicht geladen. Um es laden zu können muss zuerst ein XML File mit den Daten des Plugins erstellt werden. Dies wird vom Plugin Loader benötigt, dies geschieht am einfachsten indem ein bereits vorhandenes XML File eines anderen Plugins kopiert und angepasst wird. Das File «plugin.xml» wird direkt im Root des Projektes abgelegt, und zwar mit folgendem Inhalt:

Listing 4: simulation.xml

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <plugin>
3     <class>ch.zhaw.simulation.sim.sinnlos.SimulationSinnlosPlugin</class>
4     <name>Sinnlos</name>
5     <description>Beispiel Plugin</description>
6     <author>Hans Muster</author>
7 </plugin>
```

Um das Plugin nun zu kompilieren muss noch ein Antfile erstellt werden, auch dieses kann von einem bestehenden Plugin kopiert werden. Jedoch muss dann bei jeder Änderung das Antfile erneut ausgeführt werden, was ziemlich mühsam ist, daher gehen wir einen einfacheren Weg: wir kopieren das plugin.xml im Projekt «Simulation» in den Ordner «plugin/simulation» unter dem Name «Sinnlos.xml»

Danach wird die Applikation gestartet, dies geschieht durch ausführen der Klasse «ch.zhaw.simulation.startu
aus dem Projekt «Simulation».

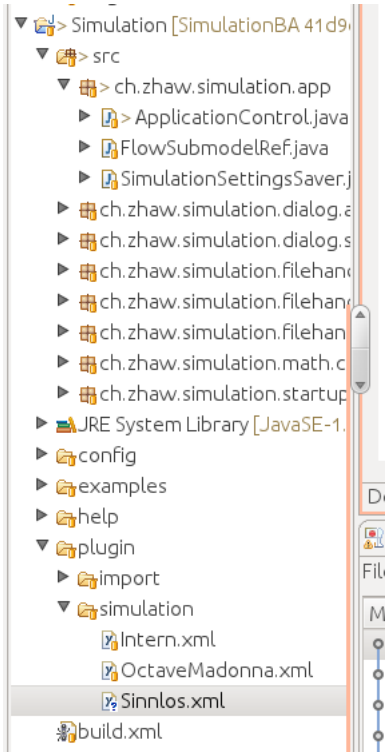

Abbildung 10.2: Sinnlos.xml im korrekten Verzeichnis

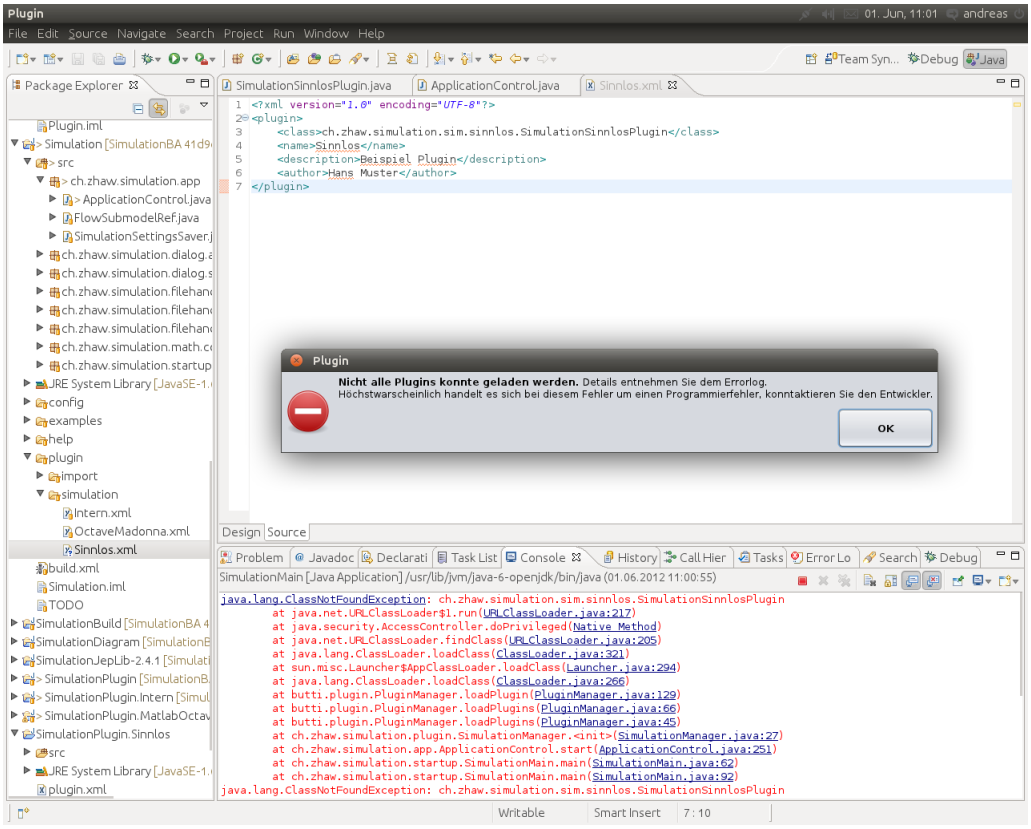

Abbildung 10.3: Fehler beim laden des Plugins

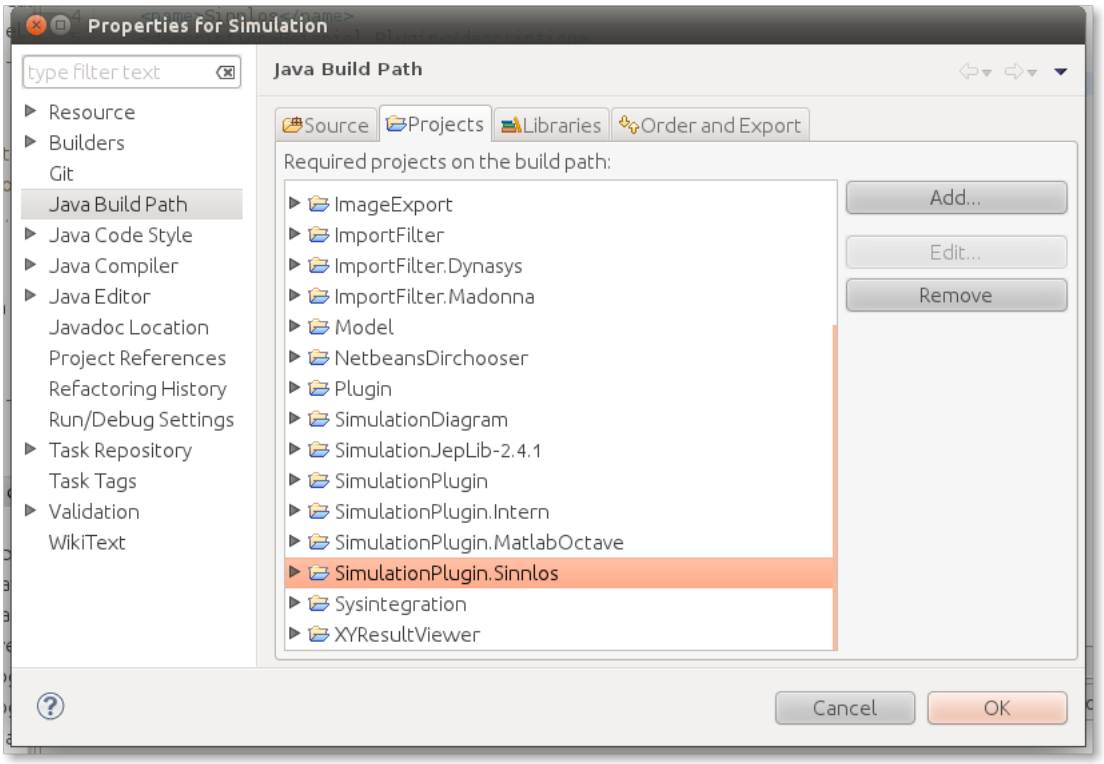

Abbildung 10.4: Projekt «Simulation» Einstellungen

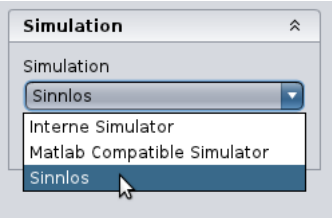

Abbildung 10.5: Wählen des neuen Plugins

Nun erscheint ein Fehler, dass das Plugin nicht geladen werden konnte. Der Fehler wird geloggt, jedoch wird die Exception incl. allen Texten (in diesem Fall sind keine Zusatzlichen Texte vorhanden) auf der Konsole ausgegeben. In diesem Fall konnte die Klasse nicht geladen werden, dies kann einfach gelöst werden, indem beim Projekt «Simulation» das Projekt «SimulationPlugin.Sinnlos» in den Referenzierten Projekte (Java Build Path) hinzugefügt wird.

Danach sollte das Plugin korrekt funktionieren, es muss noch in der Sidebar ausgewählt werden.

Das Testen / Weiterentwickeln ist jetzt dem Leser überlassen!

10.2. Erstellen eines neuen GUI Elementes

asdf

10.3. Ant Buildsystem

Ant wird von uns verwendet um die Simulationsapplikation zu Builden

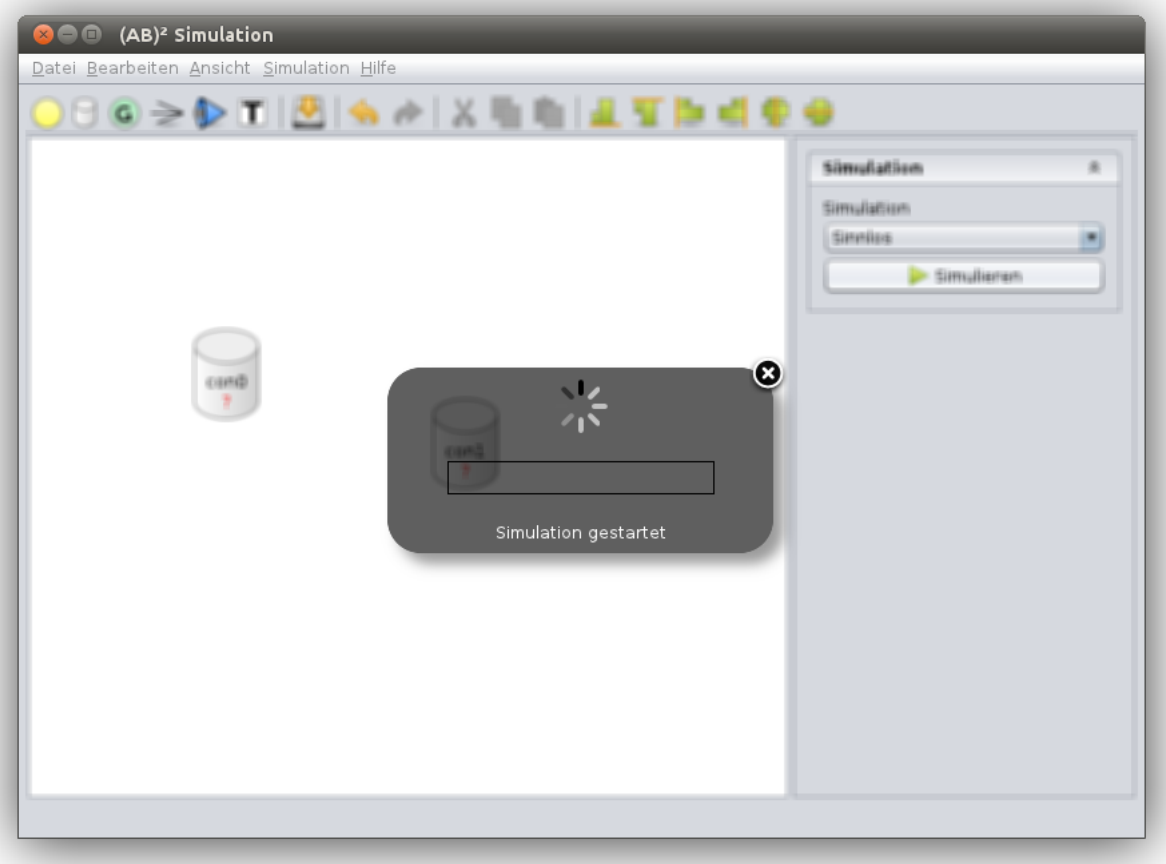

Abbildung 10.6: Start der Simulation

10.4. Setup

asdf

11. Benutzerhandbuch

Das Benutzerhandbuch soll die Grundlegenden Funktionen der Applikation beschreiben. Es wird davon ausgegangen das die grundlegende Funktionsweise von Simulationstools bereits bekannt ist, wie z.B. von Berkeley Madonna / Dynasys, die ähnlich funktionieren.

11.1. Installation

asdf

11.1.1. Portable Version

Die Portable Version benötigt keine Installation. Die Applikation wird als .zip zur Verfügung gestellt, und wird einfach entpackt. Ist die Applikation entpackt kann der komplette Ordner einfach kopiert werden, z.B. auf einen USB Stick, und kann dann an jedem beliebigen PC der über eine Java Runtime verfügt gestartet werden.

Unter Linux / Unix / Mac OS X wird die Applikation gestartet indem das Shellscript Simulation.sh gestartet wird.

Unter Windows wird die Applikation gestartet indem Simulation.exe gestartet wird.

TODO Screenshots!

11.2. Übersicht / Begriffsdefinitionen

Das Bedienkonzept der Applikation ist relativ einfach. Auf den Einsatz eines Kontextmenüs wurde verzichtet. (Je nach Betriebsystem kann ein Kontextmenü angeboten werden, alle diese Aktionen sind jedoch auch mit Tastenkombinationen oder im Menü / der Toolbar zu finden).

Alle Wichtigen Operationen werden in der Toolbar dargestellt, diese kann auch über ein Drop-Down Menü verfügen, dies wird mit einem kleinen Pfeil nach unten signalisiert. Für alle gebräuchlichen Aktionen wird eine Tastenkombination angeboten, siehe 11.4.

Editierschritte können Rückgängig gemacht werden (bis max. 100 Schritte), dafür wird auf "Sind Sie sicher..." Nachfragen verzichtet. Trotzdem liegt es in der Verantwortung des Benutzers regelmässig zu speichern, und ggf. Backups anzulegen.

Beim Starten der Applikation wird das **Hauptfenster** angezeigt. Dieses gibt es in zwei Varianten, eine für die XY Simulation und eine für die Flow Simulation, diese sind abgesehen von einer leicht verschiedene Funktionalität jedoch identisch.

1. Die **Toolbar**, oben die **Menübar**. Links in der Toolbar sind die Elemente die eingefügt werden können. Weiter rechts sind die Speichern / Rückgängig / Widerherstellen und Layout Buttons
2. Die **Zeichnungsfläche**, hier wird das Model gezeichnet.
3. Die **Sidebar**, hier werden die Einstellungen für die Simulation und für die aktuell Selektierten Elemente vorgenommen.
4. Die **Statusleiste**. Hier wird der Benutzer darüber Informiert was er als nächstes machen kann.

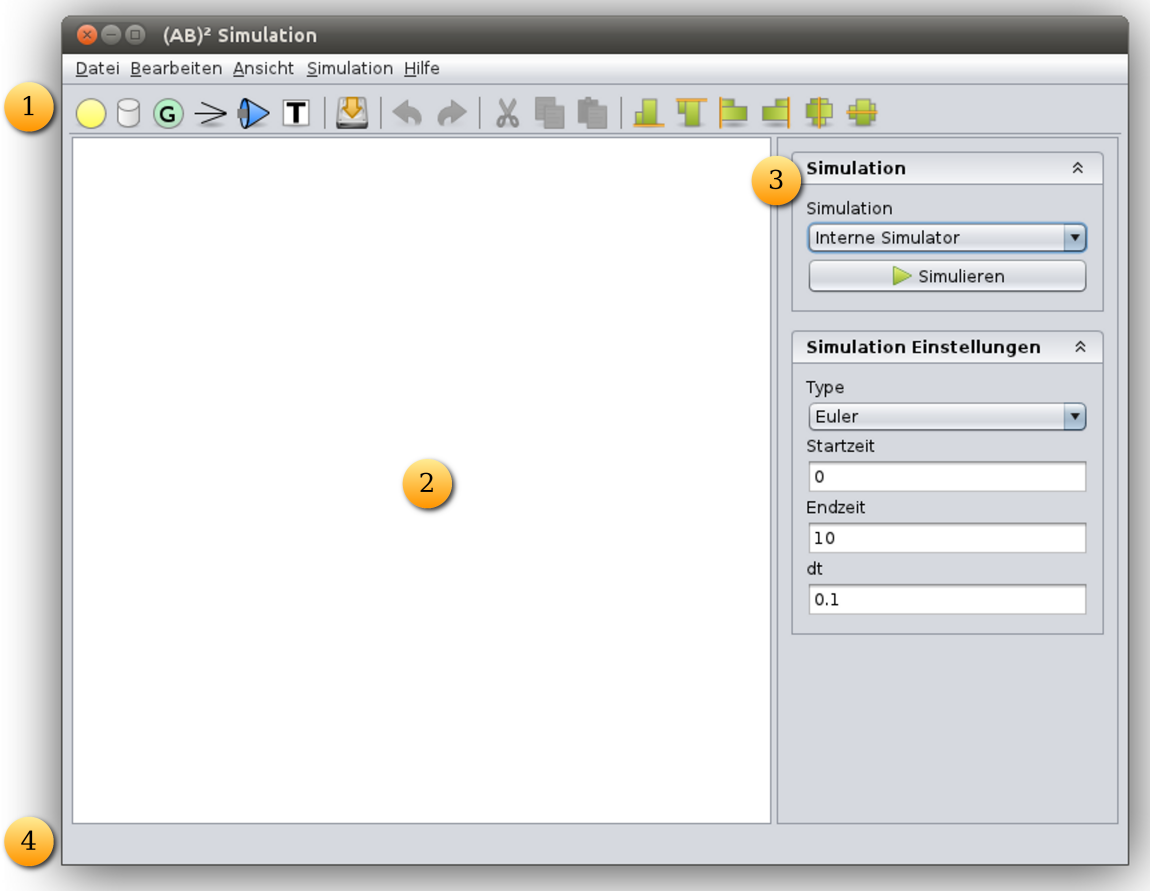

Abbildung 11.1: Übersicht Applikation

11.3. Erstellen eines einfachen Flowmodells

Hier wird erläutert wie ein Flow Modell erstellt werden kann.

1. Die Applikation wird gestartet / es wird «Datei / Neu» bzw. «Datei / Neu - Flow Model» gewählt, je nachdem ob man bereits ein Flow oder ein XY Modell geöffnet hat.
2. Es wird ein Container erstellt: Entweder wird auf den grauen Zylinder in der Toolbar geklickt oder es wird «C» gedrückt. Danach wird in die Zeichnungsfläche geklickt.
3. Wenn auf den Container gecklickt wird wird er orange, er ist nun selektiert. Zudem wird oben rechts vom Container ein kleiner blauer Pfeil dargestellt, mit diesem Pfeil können direkt Verbindungen vorgenommen werden. Verbindungen können aber auch über die Toolbar erstellt werden.
4. asdf

11.4. Tastenkombinationen

Tastenkombinationen werden hier in der Auflistung immer mit CTRL angegeben. Auf Mac OS X ist anstelle von CTRL normalerweise COMMAND ("Blumenkohl") zu verwenden.

11.4.1. Grundlegend

Dies ist Liste mit den gängigen / nützlichen Tastenkombinationen die oftmals bereits vom Betriebssystem bereitgestellt werden. Diese Liste ist nicht vollständig. Diese Kombinationen funktionieren an vielen Orten der Applikation, z.B. im Hauptfenster, in einzelnen Texteingabefeldern, im Formeleditor etc.

Tastenkombination	Funktion
CTRL + A	Alles markieren
CTRL + C	Kopieren
CTRL + V	Einfügen
CTRL + X	Ausschneiden
CTRL + S	Speichern
CTRL + O	Öffnen
CTRL + N	Neu

11.4.2. Hauptfenster

Für die Tastenkombinationen ohne Hilfstaste (CTRL etc.) muss der Fokus auf der Zeichnungsfläche liegen. Dies geschieht indem einfach kurz in einen freien Bereich der Zeichnungsfläche geklickt wird. Die Meisten Tastenkombinationen werden im Menü angegeben, oder in den Tooltips der Toolbuttons in Klammern angegeben. Diese Liste ist nicht vollständig.

Tastenkombination	Funktion
p	Einfügen eines Parameters (Nur Flow)
c	Einfügen eines Containers (Nur Flow)
g	Einfügen eines Global
t	Einfügen eines Textes
m	Meso Kompartment (Nur XY)
F9	Simulation Starten