

WPV C# & Design Patterns

Live-Tuner

Alexander Baitinger, Tobias Grimm

8. Juni 2014

Betreuer der Hochschule: Prof. Dr. Gerd Unruh

Inhaltsverzeichnis

1	Einleitung	1
1.1	Wichtiges über die Regelungstechnik	1
2	Aufbau des Live-Tuners	4
2.1	Graphische Oberfläche	5
2.2	Simulation	8
2.2.1	Fassade Entwurfsmuster	10
2.2.2	Beobachter-Entwurfsmuster	11
2.3	Speichern und Laden	14
3	Zusammenfassung	16
3.1	Ausblick	16
4	Quellenverzeichnis	17
	Abbildungsverzeichnis	18

Personen des Projekts

Name	Foto	E-Mail Adresse
Alexander Baitinger		alexander.baitinger@hs-furtwangen.de
Tobias Grimm		tobias.grimm@hs-furtwangen.de

1 Einleitung

Das 6. Semester der Hochschule Furtwangen beinhaltet für den Studiengang Elektronik und Technische Informatik (ETI) die Möglichkeit durch Wahlpflicht-Vorlesungen (WPVs) das Gesamtbild des angehenden Ingenieurs abzurunden.

Gerade für einen Elektrotechniker bietet es sich daher an, nicht nur die elektrotechnische Welt zu kennen, sondern auch die unzähligen Möglichkeiten der Informatik kennen zu lernen und diese in Kombination zu nutzen.

Aus diesem Gedanken heraus ist die Idee entstanden ein Programm mit C# zu entwickeln, welches den Arbeitsalltag eines Elektrotechnikers enorm erleichtern kann, indem regelungstechnische Auslegungen schnell und komfortabel durchgeführt werden können.

Um beide Welten sauber in Einklang bringen zu können wird auf der elektrotechnischen Seite auf die Regelungstechnik mit der Laplace-Transformation und die Z-Transformation zurückgegriffen und auf der Informatik Seite auf die Vorzüge der objektorientierten Programmiersprache C# und die Verwendung von „Design Patterns“.

1.1 Wichtiges über die Regelungstechnik

Warum Regelungstechnik?

In sehr vielen Anwendungen kommt es vor, dass man einen Soll-Wert vorgibt und diesen mit einem Ist-Wert vergleicht und dann entscheidet, was gemacht werden soll. Dieser Vorgang wird in der Regelungstechnik behandelt.

So kann man sich z.B. vorstellen, dass ein Motor auf eine gewisse Soll-Drehzahl beschleunigt werden soll. Da die Drehzahl sich wegen der Trägheit der Masse des Motors nicht direkt auf die gewünschte Drehzahl erhöht, wird ein Regler eingesetzt. Dieser schaut sich den Soll-Ist-Vergleich an und entscheidet dann, ob entweder *mehr Drehmoment* oder *weniger Drehmoment* aufgebracht werden muss, um den Motor auf die gewünschte Drehzahl zu bringen.

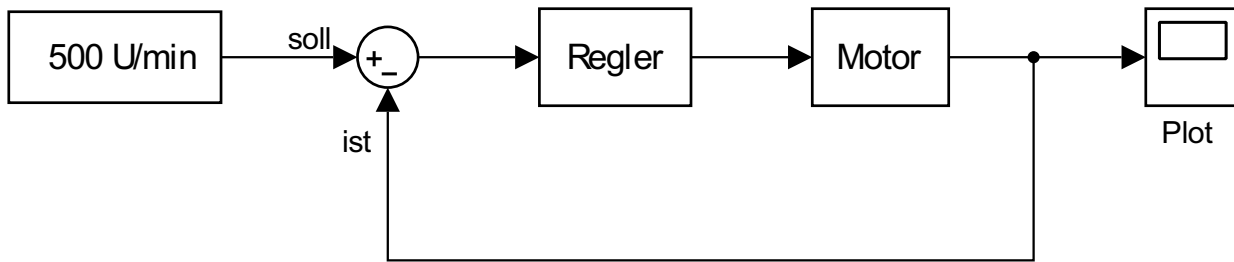


Abbildung 1.1: Beispiel Regelkreis für einen Motor

In der Regelungstechnik werden nun Verfahren untersucht, um diesen Regler ideal einzustellen, damit das zeitlich dynamische Verhalten eines Vorgangs ideal in den Griff gebracht werden kann.

Was sind Übertragungsfunktionen?

Um dynamische Vorgänge in der physikalischen Welt mathematisch beschreiben zu können werden in der Regel so genannte Differentialgleichungen eingesetzt. Hierdurch kann z.B. das dynamische Verhalten eines Motors, einer elektronischen Schaltung, einer Temperatur, eines mechanischen Vorgangs, ... beschrieben werden.

Das eigentliche Problem besteht nun oft darin, dass das Lösen dieser Differentialgleichung sehr schwer ist. An dieser Stelle bietet die Laplace-Transformation einen eleganten Weg, um aus einer Differentialgleichung eine Übertragungsfunktion im Laplace-Bereich zu gewinnen.

Diese Übertragungsfunktion (Transferfunktion) bietet für unsere Zwecke entscheidende Vorteile:

► *Handhabung*

Es ist wesentlich einfacher mit einer Übertragungsfunktion zu rechnen.

► *Universell*

Die Regelungstechnik ist für alle dynamischen Vorgänge äquivalent!

► *Simulierbar*

Durch Verwendung der Z-Transformation kann eine Übertragungsfunktion „relativ“ einfach auf einem Computer simuliert werden.

► *Betrachtung als Block*

Die einzelnen Übertragungsfunktionen für Regler, Motor, ... können als Blöcke wie in Bild 1.1 betrachtet werden.

Warum ein neues Software-Tool?

Die gesamte Regelungstechnik besteht zum größten Teil aus Mathematik. Hier kann es sehr schnell passieren, dass man den Überblick verliert, da es einem schwer fällt sich gewisse abstrakte Gebilde vorzustellen.

Das hier entwickelte Software-Tool möchte genau an dieser Stelle ansetzen, dem Benutzer ein Gefühl dafür zu geben, wie die einzelnen abstrakten Gebilde zusammenhängen, und welchen Einfluss die einzelnen Parameter auf das dynamische Verhalten des Systems haben.

Darüber hinaus sind bereits Verfahren hinterlegt, um einen Regler gut auszulegen und diesen visuell in Echtzeit an die Bedürfnisse des Ingenieurs nach zu tunen. Aus diesem Grund hat das neu entstandene Tool den Namen „Live-Tuner“ erhalten.

2 Aufbau des Live-Tuners

In diesem Kapitel wird der eigentliche Aufbau der Software des Live-Tuners näher beschrieben. Eine erste Übersicht bietet hier das UML-Klassendiagramm 2.1.

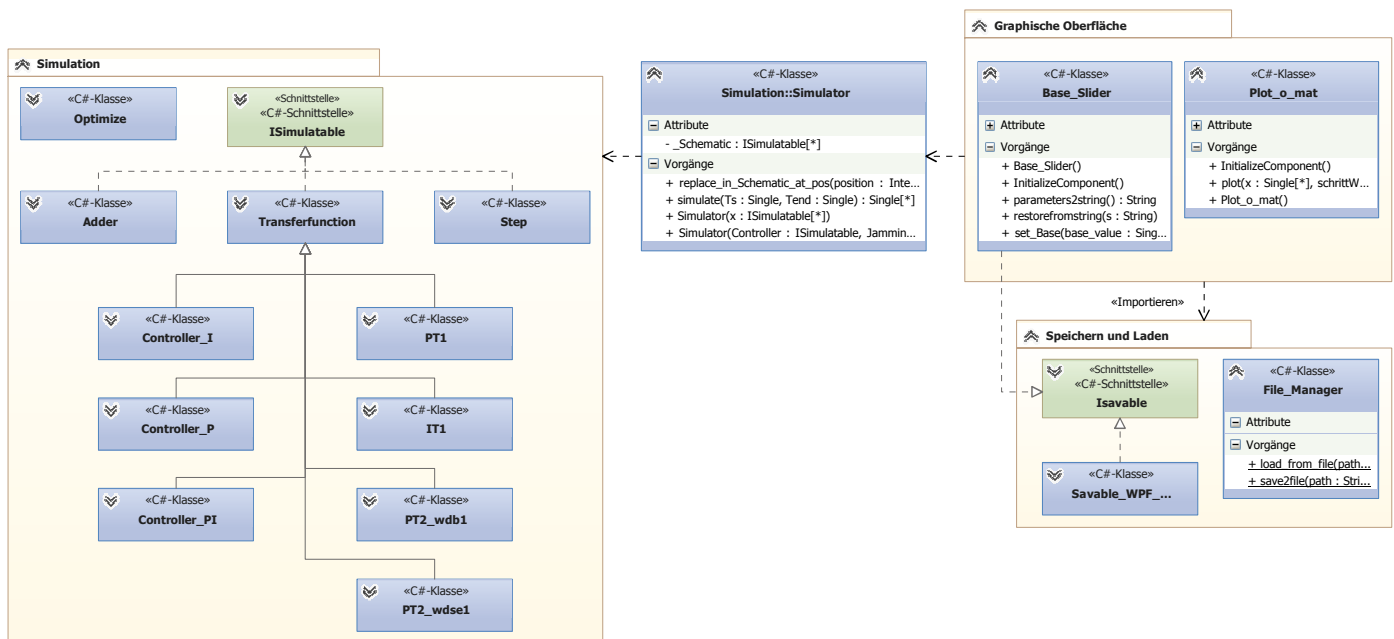


Abbildung 2.1: UML-Klassendiagramm

Wie leicht zu erkennen ist, baut sich der Live-Tuner aus drei großen Komponenten auf:

- *Graphische Oberfläche* 2.1 auf der nächsten Seite
- *Simulation* 2.2 auf Seite 8
- *Speichern und Laden* 2.3 auf Seite 14

Diese drei Komponenten werden nun in den folgenden Kapiteln näher beleuchtet.

2.1 Graphische Oberfläche

Das Ziel

Das Hauptziel der Oberfläche ist es, diese so zu konzipieren, dass eine Veränderung an einem der einstellbaren Parametern sofort zu einer Veränderung am Plot führt, also sozusagen "Live". So ist es leichter, ein Gespür für die verschiedenen einstellbaren Werte zu erlangen. Des Weiteren soll dem Nutzer viel Komfort entgegengebracht werden, z.B. gibt das Programm selbst vor, welche Regler zu welcher Regelstrecke sinnvoll sind und welche eher nicht.

Von der Aufgabestellung war vorgegeben, die Oberfläche mit der Windows Presentation Foundation (WPF) zu erstellen. In dieser ist es zwar noch möglich, die Oberfläche in C# zu konfigurieren, jedoch nicht mehr so, wie es in den älteren Windows-Forms Anwendungen der Fall war. Die Einstellungen für die Oberfläche werden nun in einem XML-Dokument vorgenommen.

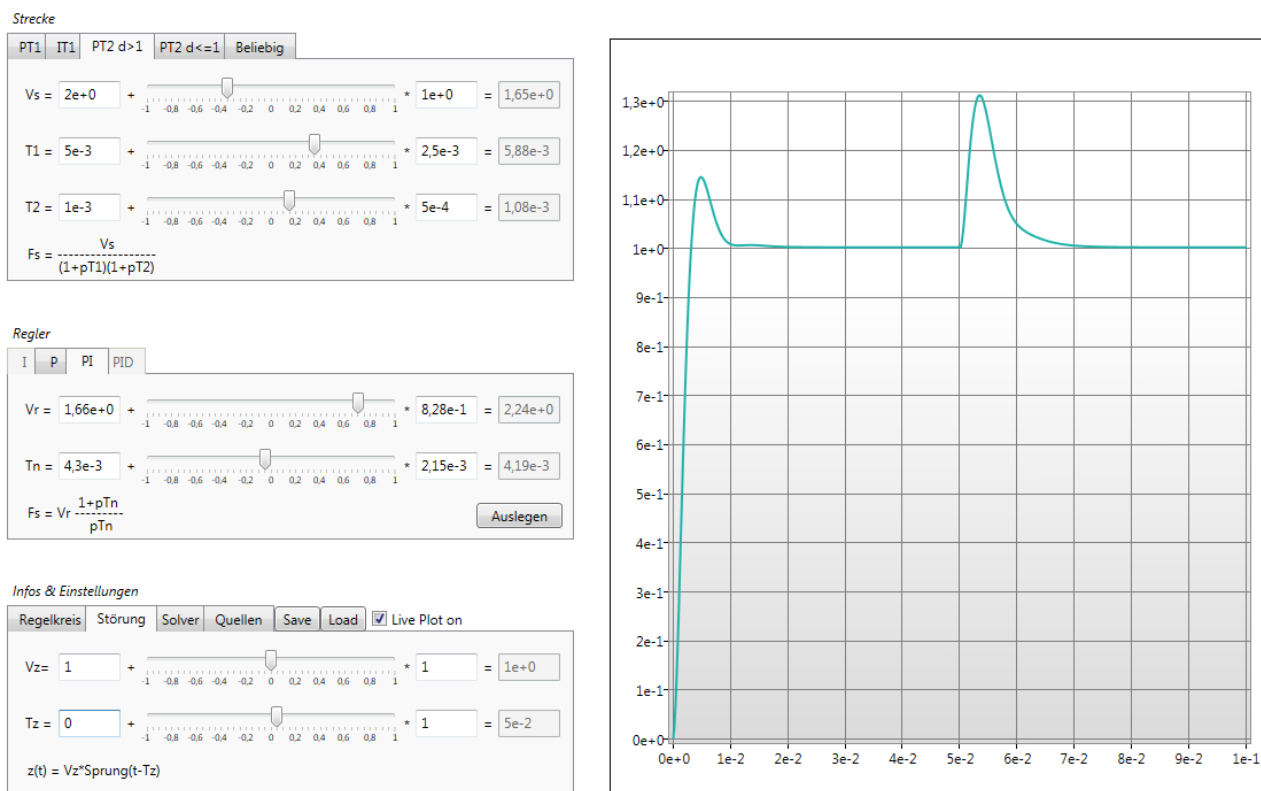


Abbildung 2.2: Graphische Oberfläche des Live-Tuners

Live Plot

Der Live Plot hat die Aufgabe die ihm übergebenen Werte in einem Graph darzustellen. Da die Oberfläche mit dem Windows Presentation Foundation (WPF) Oberflächendesigner erstellt wurde, wurde für die Darstellung des Graphen das WPF-Toolkit verwendet. Dieses enthält alle rudimentären Darstellungsformen und -einstellungen für verschiedene Graphentypen. Für den Graph wurde der Graphentyp "LineSeries" verwendet. Diese plottet anhand von x- und y-Werten den Graphen.

Einer der Vorteile des WPF-Toolkits ist, dass der Graph sich je nach den übergebenen Werten automatisch skaliert. Die Größenordnung der Werte werden aber nicht angepasst. Dies muss in den Einstellungen separat durchgeführt werden. Die Ausgabe der Werte wurde nun so angepasst, dass diese vorher in eine Exponentialschreibweise umgeformt werden. So können nun extrem große, aber auch extrem kleine Werte ausgegeben werden.

Strecke und Regler

Unter diesen Punkten kann die Regelstrecke und der dazu passende Regler eingestellt werden. Die Oberfläche gibt automatisch je nach gewählter Regelstrecke den passenden Regler vor. Auch hier werden die Werte der verschiedenen Parameter in Exponentialschreibweise dargestellt. Die Einstellungen für die einzelnen Parameter werden mit Hilfe der Base Slider vorgenommen.

Base Slider

Die Einstellungsmöglichkeiten für die Regler bzw. Strecke sind alle gleich. Deshalb wurde ein separates Oberflächenobjekt erstellt, der sogenannte Base Slider. Dieser besteht im wesentlichen aus einem Anfangswert, zudem dann noch ein zweiter Wert hinzuaddiert wird. Dabei wird der zweite Wert per Slider zwischen den Werten 1 und -1 eingestellt und mit einem Faktor multipliziert. Durch diesen Trick kann ideal in dem gewünschten Zahlenbereich gearbeitet werden.

Infos & Einstellungen

Unter dem Punkt Infos & Einstellungen befinden sich folgende Punkte:

- ▶ *Regelkreis*
Ein Bild des verwendeten Regelkreises, welcher simuliert wird.
- ▶ *Störung*
Hier lassen sich Störungen in den Regelkreis einspeisen.
- ▶ *Solver*
Unter dem Punkt Solver lässt sich die Schrittweite der Simulation, sowie der Endzeitpunkt der Simulation bestimmen. Darüber hinaus gibt es hier die Möglichkeit für den Plot weniger Werte zu verwenden als die Simulation berechnet. Dieses Feature ermöglicht es einem die aufzuwendende Rechenleistung zu verringern, da die Berechnung für die Simulation sehr viel geringer ist als die Berechnung des Plots.
- ▶ *Quellen*
Das Programm wurde mithilfe der angegebenen Quellen entwickelt und erstellt.
- ▶ *Save & Load*
Alle Einstellungen lassen sich mit dem Save-Befehl in einer Datei speichern. Mit dem Load-Befehl kann so zu einem späterem Zeitpunkt auf die Einstellungen zurückgegriffen werden.
- ▶ *Live Plot*
Wenn das Häkchen bei Live Plot gesetzt wird, werden die berechneten Werte direkt geplottet. Will man sehr viele Werte auf einmal ändern, so empfiehlt es sich dieses Häkchen zu deaktivieren, um schneller arbeiten zu können.

Warnings

Bei den Solver-Einstellungen ist es dem Benutzer frei gestellt diese an seine Bedürfnisse anzupassen. Jedoch gibt es Einstellungskombinationen, die sehr rechenintensiv sind. Dies ist erstens nicht notwendig, und kann zweitens zu sehr langen Simulationsdauern führen. Aus diesem Grund wird dem Benutzer bei kritischen Eingaben eine Warning ausgegeben, welche Sie in Abbildung 2.3 sehen können.

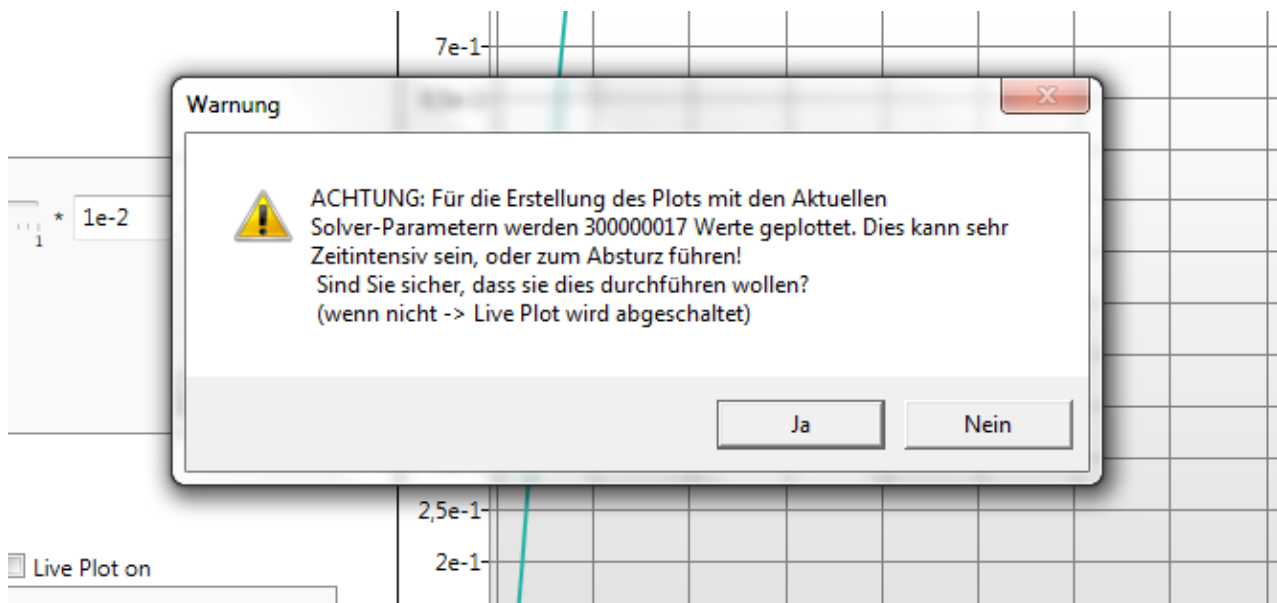


Abbildung 2.3: Warnung für rechenintensive Solver Einstellungen

2.2 Simulation

Durch die komplette Simulation eines Regelkreises soll dem Benutzer graphisch gezeigt werden, wie gut sein aktueller Regler ausgelegt ist. Ein allgemeiner Regelkreis besteht aus folgenden Komponenten:

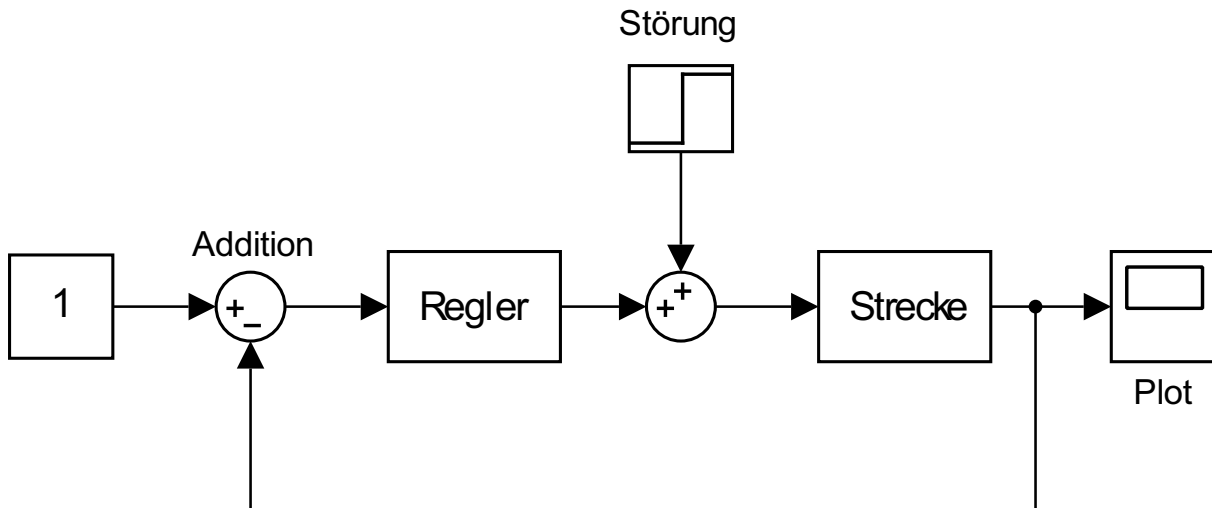


Abbildung 2.4: Allgemeiner Aufbau eines Regelkreises

- ▶ *Soll-Wert*
Gewünschter Wert, welcher erreicht werden soll.
- ▶ *Vergleich Soll-Ist*
Durch die Addition(+/-) wird ein Soll-Ist Vergleich gemacht, welcher zum Regler geht.
- ▶ *Regler*
Der Regler ist eine Übertragungsfunktion, und je nach Bedarf ein P-, I-, oder PI-Regler.
- ▶ *Störung*
Durch die Störung sollen mögliche Störfaktoren mitsimuliert werden.
- ▶ *Strecke*
Die Strecke ist die Übertragungsfunktion des physikalischen Systems (Motor, elektrische Schaltung,...)
- ▶ *Plot*
Der Plot beobachtet quasi die einzelnen Simulationsschritte und gibt diese graphisch aus.

Die Idee für die softwaretechnische Umsetzung einer Simulation besteht nun darin, dass jede einzelne Komponente des Regelkreises durch eine einzelne Klasse beschrieben wird. Bei einer Simulation berechnet nun jeder Teilnehmer einen kleinen Zeitschritt und gibt das Ergebnis an den nächsten Teilnehmer im Regelkreis weiter. Dies wird so oft wie erwünscht wiederholt und am Schluss als Gesamtergebnis zurückgegeben.

Die Problematik

- ▶ Die ganzen Berechnungen zur Simulation eines Regelkreises sind sehr kompliziert, wie kann man es nach außen vereinfachen, sodass man sich nicht hinein denken muss, wenn man dieses verwenden will?
- ▶ Die einzelnen Klassen im Regelkreis für Regler und Strecke können je nach Anwenderbedarf unterschiedlich sein! Wie können nun diese Klassen miteinander kommunizieren, ohne sich direkt kennen zu müssen?

Lösungsansatz

Einen sehr guten Lösungsansatz bieten hier die Schablonen der Entwurfsmuster (Design Patterns). Hierbei handelt es sich um bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in der Softwareentwicklung.

Um diese zwei Probleme zu behandeln wurden folgende zwei Entwurfsmuster ausgewählt:

2.2.1 Fassade Entwurfsmuster

Beschreibung des Entwurfsmusters (wiki)

Wenn ein Subsystem viele technisch orientierte Klassen enthält, die selten von außen verwendet werden, hilft es, eine Fassade zu verwenden. Die Fassade ist eine Klasse mit ausgewählten Methoden, die eine häufig benötigte Untermenge an Funktionalität des Subsystems umfasst. Sie delegiert die Funktionalität an andere Klassen des Subsystems und vereinfacht dadurch den Umgang mit dem Subsystem.

Verwendung der Fassade

Wie im Ausschnitt des UML-Diagramms 2.5 gut zu erkennen ist, bildet die Klasse *Simulator* eine Fassade in diesem Projekt.

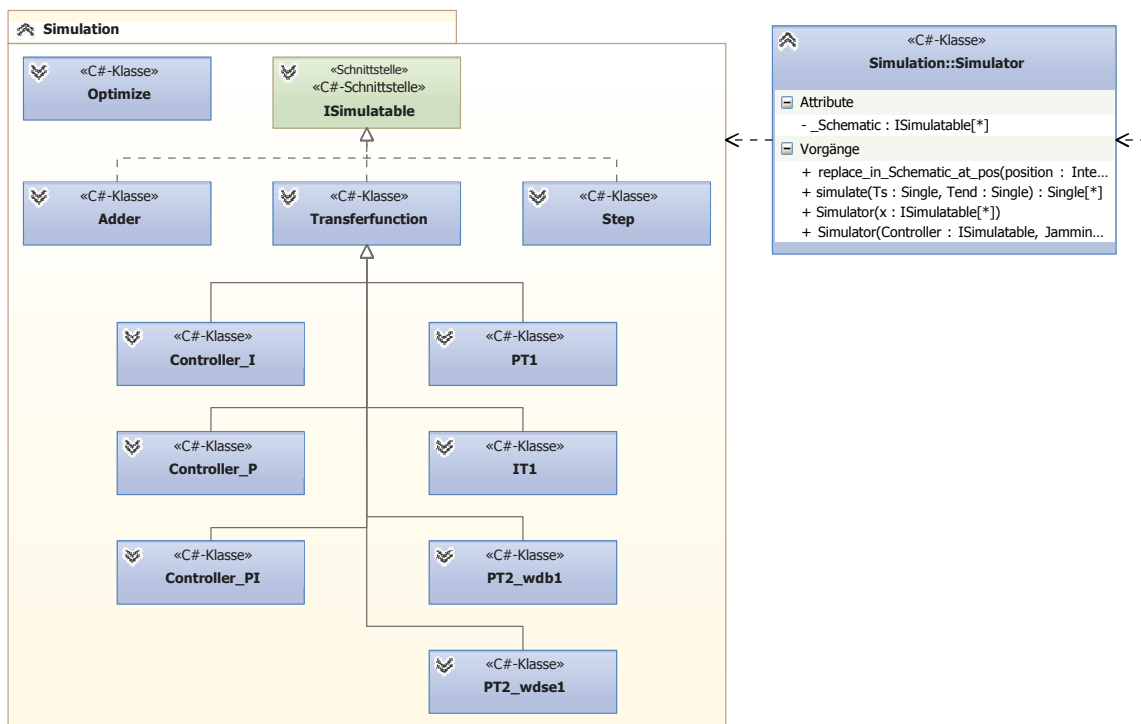


Abbildung 2.5: Klasse „Simulator“ als Fassade

Der *Simulator* kümmert sich um die komplette Verwaltung der einzelnen Klassen einer Simulation, vernetzt sie miteinander und leitet die einzelnen Berechnungen. Er merkt sich in einem Array von Objekten, welche das Interface *ISimulatable* implementieren, in welcher Reihenfolge die einzelnen Objekte bei einer Simulation aufgerufen werden müssen.

Durch diesen Einsatz wird die Komplexität einer Simulation auf ein Minimum von zwei Funktionen heruntergebrochen:

- `float[,] simulate(float Ts, float Tend)`

Simuliere den aktuellen Regelkreis mit den festen kleinen Zeitschritten T_s bis zur Endzeit T_{end} . Gib das Ergebnis zurück.

- `bool replace_in_Schematic_at_pos(int position, ISimulatable x)`

Ersetze im Array an der Stelle *position* das Objekt durch *x*. Löse die alten Vernetzungen und setze neue. (Hierdurch können die Objekte im Regelkreis vertauscht werden.)

2.2.2 Beobachter-Entwurfsmuster

Beschreibung des Entwurfsmusters (wiki)

Allgemein finden Beobachter-Muster Anwendung, wenn eine Abstraktion mehrere Aspekte hat, die von einem anderen Aspekt derselben Abstraktion abhängen, die Änderung eines Objekts Änderungen an anderen Objekten nach sich zieht oder ein Objekt andere Objekte benachrichtigen soll, ohne diese im Detail zu kennen.

Verwendung des Beobachter-Musters

Dieses Muster bietet sich ideal für das zweite Problem an, da sich die einzelnen Objekte in einem Regelkreis nicht direkt kennen müssen, um sich gegenseitig benachrichtigen zu können. Diese Benachrichtigungen sind immer dann erforderlich, wenn ein Objekt im Regelkreis einen Zeitschritt für sich berechnet hat. Ist die Berechnung fertig, so wird das Ergebnis an das beobachtende Objekt weitergegeben. Anschaulich bedeutet dies, dass alle Verbindungen im Regelkreis 2.6 solche Benachrichtigungen (events) sind.

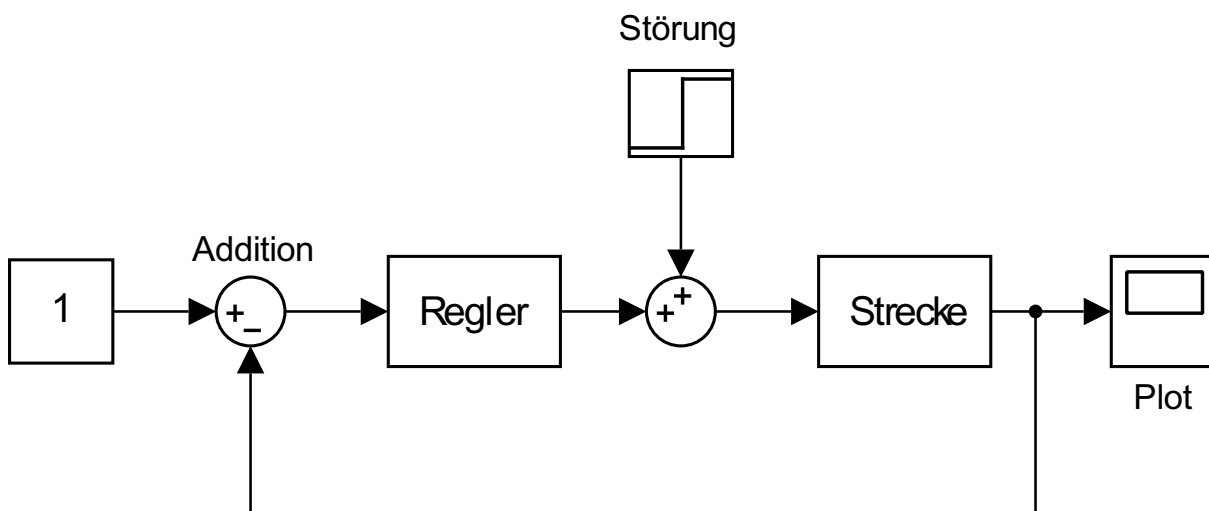


Abbildung 2.6: Benachrichtigungen in einem Regelkreis

Hierdurch ist nun klar, wie das Interface aufgebaut sein muss, welches eine Klasse implementiert, um simulierbar zu sein:

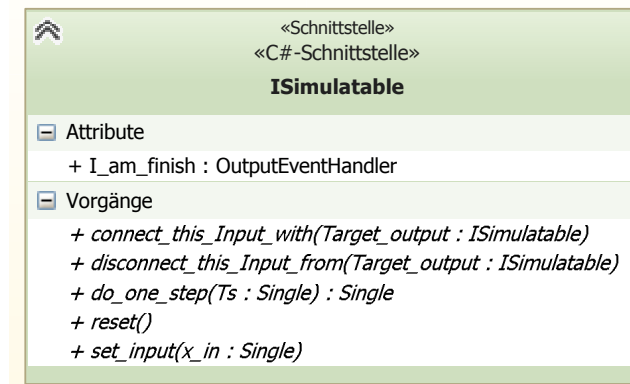


Abbildung 2.7: Interface für Simulation

- ▶ *event OutputEventHandler I_am_finish*
Dieses event wird gefeuert, wenn eine Berechnung fertig ist.
- ▶ *void set_input(float x_in)*
Setze den Eingangswert für die Berechnung des nächsten Zeitschrittes.
- ▶ *float do_one_step(float Ts)*
Berechne den Ausgangswert aus dem Eingangswert und feuere das Event *I_am_finish*.
- ▶ *void reset()*
Leere den Speicher der Vergangenheit.
- ▶ *void connect_this_Input_with(ISimulatable Target_output)*
Beobachte den Ausgang des anderen Objektes, indem du deinen Eingang mit diesem Ausgang verbindest.
- ▶ *void disconnect_this_Input_from(ISimulatable Target_output)*
Löse die Beobachtung auf.

Durch dieses Interface und die Verwendung des Simulators kann nun der Regelkreis simuliert werden. Im UML-Sequenzdiagramm 2.8 auf der nächsten Seite ist eine Benutzeranfrage an eine Simulation skizziert.

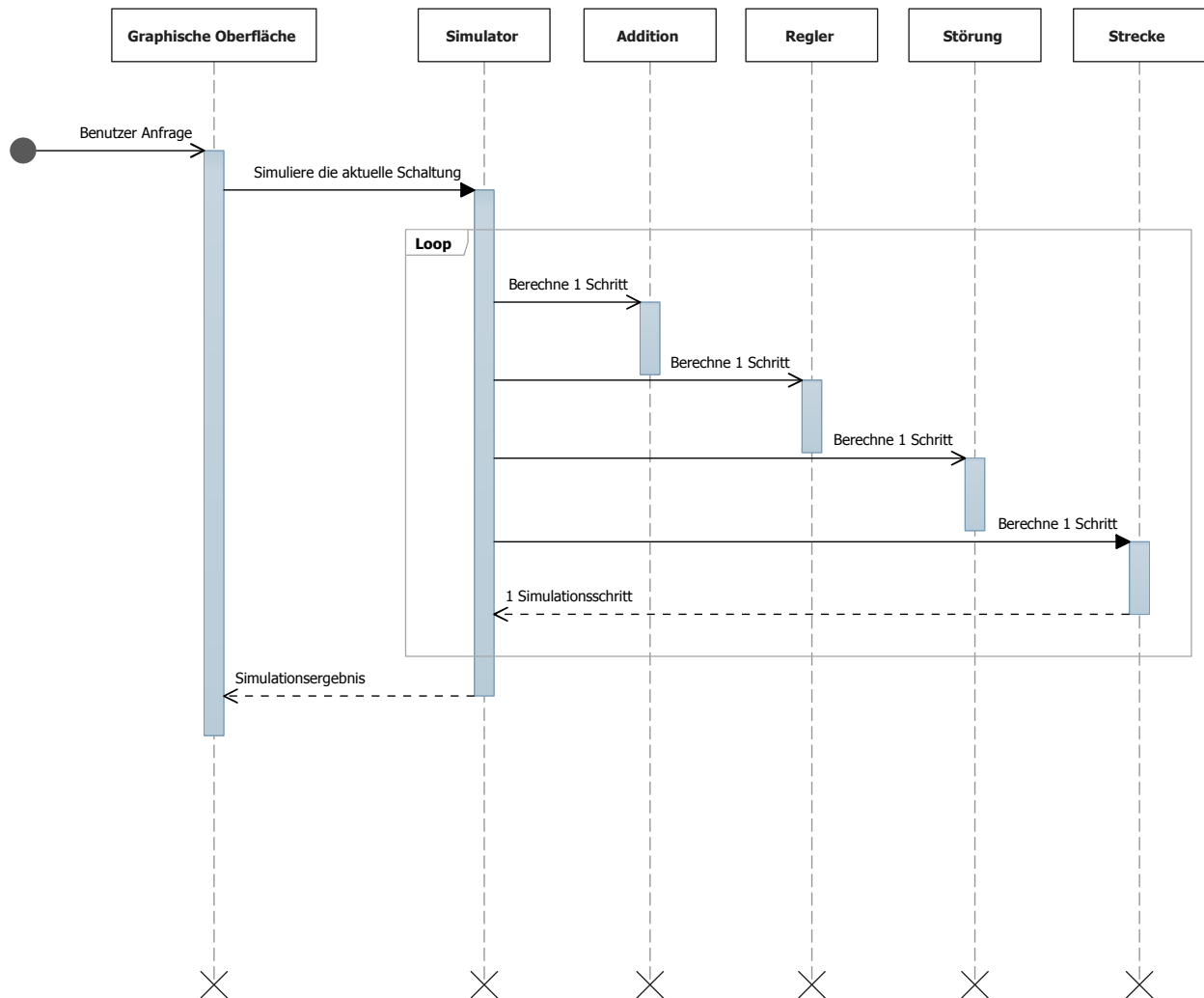


Abbildung 2.8: UML-Sequenzdiagramm für eine Simulationsanfrage

Wie deutlich zu erkennen ist, benutzt die graphische Oberfläche die Fassade des Simulators um eine Simulation zu starten. Dieser ruft nach und nach die einzelnen Komponenten des Regelkreises dazu auf einen Zeitschritt zu berechnen. Wie klar zu erkennen ist, kennen sich die einzelnen Komponenten des Regelkreises nicht, sondern beobachten sich nur gegenseitig, um Informationen zu erhalten.

Haben alle Komponenten einen Zeitschritt berechnet, so wird die Schleife erneut durchlaufen, bis die gewünschte Anzahl an Simulationsschritten erreicht ist, und das Gesamtergebnis an die graphische Oberfläche zurück gegeben wird.

2.3 Speichern und Laden

Die Motivation

Ein entscheidendes Feature eines Software-Tools ist selbstverständlich die Möglichkeit seine Arbeit speichern zu können, damit die Möglichkeit besteht diese zu archivieren und zu einem späteren Zeitpunkt auf den alten Stand zurückzugreifen. Ein weiterer großer Vorteil besteht darin, dass man die gespeicherte Datei ganz einfach an Arbeitskollegen verteilen kann, damit diese sich die Auslegung anschauen können.

Aus diesem Gedanken heraus ist es eine Notwendigkeit ein solches Feature für den Live-Tuner bereitzustellen.

Umsetzung

Im UML-Klassendiagramm 2.9 sehen Sie das Lösungskonzept für dieses Feature.

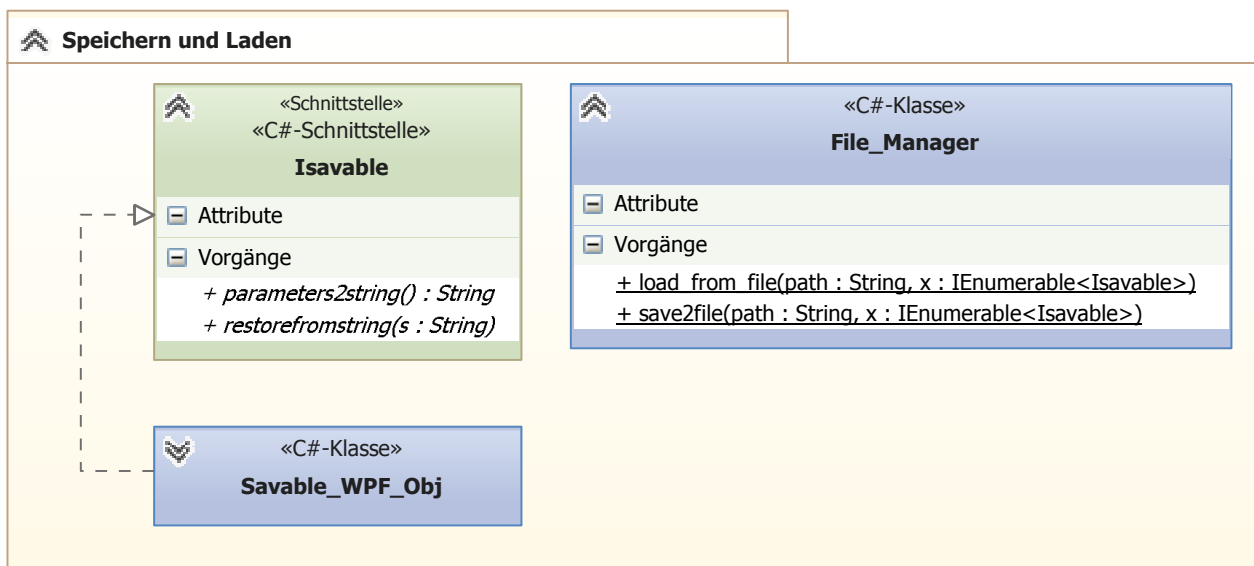


Abbildung 2.9: UML-Klassendiagramm für das Speichern und Laden

Die eigentliche Idee ist ganz einfach, jedes Objekt welches gespeichert werden soll, muss genau zwei Methoden zur Verfügung stellen, welche im Interface **ISavable** definiert sind:

- ▶ `string parameters2string()`
Baue alle Informationen, die du dir merken willst, zu einem String zusammen.
- ▶ `void restorefromstring(string s)`
Stelle dich aus dem gebauten String wieder her.

Durch dieses Interface ist es dem *File_Manager* möglich alle Objekte dieses Interfaces speichern zu können. Er baut alle Informationen der Teilnehmer zu einem großen String zusammen, und kann diesen dann als Text-Datei abspeichern und selbstverständlich auch wieder herstellen.

Dieser Ansatz zum Speichern von Informationen kann nun auch natürlich für beliebige Klassen eingesetzt werden.

Wie können nun bereits bestehende Klassen gespeichert werden?

Nun kann man sich vorstellen, dass es bereits sehr viele Standard-Klassen gibt, welche dieses Interface nicht implementieren. Im Falle des Live Tuners war es z.B. erforderlich Informationen über die WPF-Oberfläche wie TextBox, CheckBox, ComboBox, ... zu speichern.

Um hier Abhilfe zu schaffen, wurde die Klasse *Savable_WPF_Obj* geschaffen, welche ganz genau weiß, wie eine TextBox, eine CheckBox, eine ComboBox, ... zu einem String gebaut wird, und auch wieder hergestellt wird.

Durch diesen kleinen Trick ist es nur noch erforderlich diesem Objekt zu sagen, welche WPF Objekte gespeichert werden sollen, und dann mittels des *File_Manager* zu speichern.

3 Zusammenfassung

Mit Hilfe dieses Tools können nun schnell die passenden Werte für einen Regler zu einer bestimmten Strecke ermittelt werden. Umgekehrt kann natürlich auch schnell überprüft werden, ob die vorher errechneten Werte auch die gewünschte Sprungantwort liefern.

Damit dies so reibungslos wie möglich klappt, können die Werte mit den Schieberegler beliebig genau eingestellt werden und ermöglichen so eine perfekte Einstellung der Parameter.

Nachdem die Parameter nun eingestellt worden sind, kann der Regler gleich mit überprüft werden, wie dieser mit Störungen umgeht.

Mithilfe der Speicherfunktion ist es auch außerdem möglich, die gefundenen Parameter abzuspeichern und bei Bedarf zu späteren Zeitpunkten wieder abzurufen.

Aus diesem Grund werden wir in unserem zukünftigen Elektroniker-Alltag mit Sicherheit immer wieder auf dieses Programm zurückgreifen, um einen passenden Regler auszuwählen und einzustellen.

3.1 Ausblick

Im Moment berechnet die Software nur ein Blockdiagramm mit einem Regler, einer Strecke und einer Störung. Die Software kann nun so erweitert werden, dass verschieden Blöcke in einem separatem Fenster zusammengeschaltet werden können. Des Weiteren können so auch verschiedene Blockkomponenten implementiert werden z.B. ein Block, welcher den Logarithmus des Eingangssignals berechnet.

Auf Seiten der Speicher und Lade-Fähigkeit der Software kann eine Datenbank erstellt werden, die eine Vielzahl von voreingestellten Parametern bzw. Reglern und Strecken aufweist. So kann nach einer ähnlichen Problemstellung auf Einstellungen zurückgegriffen werden, die die grobe Auslegung schon voreingestellt hat. Bei Bedarf kann auf Basis dieser Werte dann Feintuning betrieben werden.

Bei hohen Werten fällt auf, dass das Programm einige Zeit braucht, da die Anzahl der berechneten Werte sehr hoch wird und die Oberfläche, solange die Rechnungen nicht abgeschlossen sind, nicht mehr reagiert. Um hier entgegenzuwirken, könnten die Berechnungen in einen separaten Prozess ausgelagert werden, sodass zumindest die Oberfläche nicht komplett einfriert.

4 Quellenverzeichnis

Bei der Einarbeitung in das Projekt und der eigentlichen Programmierung wurden folgende Dokumente verwendet.

Fachliteratur:

- ▶ C# Quick Reference - Apress
- ▶ Auf der Faehrtle von C# - Springer
- ▶ C# 3.0 Design Patterns - O'Reilly

Skripte:

- ▶ Regelungstechnik - Prof. Dr. Konstantin Meyl
- ▶ Systemtheorie - Prof. Dr. Peter Strobach

Internetseiten:

- ▶ <http://msdn.microsoft.com/>
- ▶ <http://stackoverflow.com/>
- ▶ [http://de.wikipedia.org/wiki/Fassade_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Fassade_(Entwurfsmuster))
- ▶ [http://de.wikipedia.org/wiki/Beobachter_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Beobachter_(Entwurfsmuster))
- ▶ <https://wpftoolkit.codeplex.com/>

Abbildungsverzeichnis

1.1	Beispiel Regelkreis für einen Motor	2
2.1	UML-Klassendiagramm	4
2.2	Graphische Oberfläche des Live-Tuners	5
2.3	Warnung für rechenintensive Solver Einstellungen	7
2.4	Allgemeiner Aufbau eines Regelkreises	8
2.5	Klasse „Simulator“ als Fassade	10
2.6	Benachrichtigungen in einem Regelkreis	11
2.7	Interface für Simulation	12
2.8	UML-Sequenzdiagramm für eine Simulationsanfrage	13
2.9	UML-Klassendiagramm für das Speichern und Laden	14