Institut für Rechnergestützte Ingenieursysteme
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 2

Rechnerunterstützte Problemformulierung in der Entwurfsoptimierung: Ein internetgestützter Ansatz

Florian Straßer

Studiengang: Master Informatik

Prüfer: Univ-Prof. Hon-Prof. Dr. Dieter Roller
Projektkoordinator: M. Sc. Julian Eichhoff

begonnen am: 16. April 2014
beendet am: 16. Oktober 2014

CR-Klassifikation: D.2, D.3, E.2, F.4.1, H.0, H.2.1, J.2

Kurzfassung

In industriellen Entwicklungsprozessen ist aufgrund von immer komplexeren Produkten Multidisziplinäre Entwurfsoptimierung ein wesentlicher Bestandteil geworden. Um Multidisziplinäre Entwurfsoptimierung anwenden zu können, muss das jeweilige Optimierungsproblem erst aufwändig in eine mathematische Form gebracht werden. Während Anwendungsgebiete und Berechnungsalgorithmen Multidisziplinärer Entwurfsoptimierung breit erforscht sind, führt die Unterstützung dieses Problemformulierungsprozesses momentan noch ein Nischendasein. Die Aufgabe dieser Arbeit ist es, diesen Umformungsprozess von Optimierungsproblemen zu vereinfachen. Hierzu wurde das Tool *Problem Formulator* entwickelt, welches in der Form eines Expertensystems erstellt wurde. Für die Umsetzung wurde ein internetgestützter Ansatz gewählt, dieser ermöglicht den Problemformulierungsprozess über einen Web-Browser. Zur Umsetzung wurde das Google Web Toolkit (GWT) verwendet, welches durch Technologien des semantischen Webs in Form von OWL ergänzt wurde. Zur Problemmodellierung wurde ein modularer Ansatz gewählt, der durch Instanziierung und Verkettung von Modulen die Grundstruktur eines Problems definieren lässt. Die Wissensdatenbank ist in Form einer Ontologie in OWL modelliert. Die Problemformulierung durch den Endbenutzer erfolgt über eine eigens entwickelte und aus der Wissensdatenbank dynamisch erzeugte grafische Benutzeroberfläche. Zur Evaluierung der Funktionalität wurde erst ein Basismodulkatalog erstellt, um damit ausgewählte, in der Praxis relevante, Modellierungsprobleme umzusetzen.

Abstract

Multidisciplinary design optimization has become an integral part in industrial development processes due to more complex products. To apply multidisciplinary design optimization, the problem has to be transformed into a mathematical form. This is a complex process. Applications and algorithms for calculation of multidisciplinary design optimization are well explored. At the other hand tools to support the mathematical formulation process are still niches. The purpose of this thesis is to ease the process of transformation of optimization problems. Therefore, the tool Problem Formulator has been developed. The Problem Formulator is an expert system. In order to enable the problem formulation process through a web browser, an internet-based approach has been chosen. This is implemented by the use of Google Web Toolkit (GWT) in combination with semantic web technologies (OWL). For the problem modulation, a modular approach has been chosen. The modular-approach for the basic structure of a problem, works through instantiation and linking of modules. The knowledge base is modelled as an OWL ontology. The end user problem formulation works through a dynamically generated graphical user interface, based on the modelling in the knowledge base. To evaluate the functionality a set of modules with base functionality has been developed. Finally, some practically relevant optimization problems have been implemented with these modules.

Inhaltsverzeichnis

Danksagung

Ich möchte an dieser Stelle allen Personen danken, die mich bei der Bearbeitung dieser Masterarbeit unterstützt und motiviert haben.

Zuerst einmal gilt mein Dank Professor Roller für die Vergabe der Arbeit und das damit entgegengebrachte Vertrauen. Ebenso gilt mein Dank Professor Eggenberger für seine tatkräftige Unterstützung bei der Klärung einiger Formalitäten vor dem Start dieser Arbeit.

Meinem Betreuer Julian Eichhoff gilt mein Dank für seine stete Unterstützung als Ansprechpartner in allen Phasen dieser Arbeit. Insbesondere möchte ich mich für sein offenes Ohr bei Problemen in der Anfangsphase dieser Arbeit, bedingt durch die sehr offen gehaltene Themenstellung, bedanken.

Für die geduldige Hilfe beim Korrekturlesen in der Endphase dieser Arbeit möchte ich mich bei meiner Schwester Carmen, sowie meiner Freundin Sandra ganz herzlich bedanken.

Ein besonderer Dank geht an meine Eltern die mir in meinem Studium stets moralisch, sowie finanziell zur Seite standen und mich in allem herzlich unterstützen.

1 Einleitung

In freien Märkten geht es immer darum sich Wettbewerbsvorteile gegenüber der Konkurrenz zu erarbeiten. Um Wettbewerbsvorteile zu erzielen gibt es nach [HK06:147] eine Reihe grundlegender (generischer) Wettbewerbsstrategien, diese gliedern sich wie folgt [HK06:147]:

- Kostenführerschaft
- Differenzierung
    - Differenzierung auf der Basis überlegener Produkte
    - Differenzierung auf der Basis besserer Kundenbeziehungen

Egal ob mit der Kostenführerschaft die Erreichung der günstigsten Kosten in der Branche angestrebt wird, oder mittels Differenzierung eine leistungsbezogene Überlegenheit angestrebt wird, immer wird ein für das jeweilige Kriterium optimaler Zustand angestrebt [HK06]. Es ist also ein stetiger Prozess der Optimierung von Nöten. Im Zuge des technischen Fortschritts ist dieser Optimierungsprozess mittlerweile in vielen Bereichen weit auf die Spitze getrieben. Dies zeigt sich z.B. darin, dass sich die Forschungs- und Entwicklungskosten im Jahr 2012 in Deutschland um 11,9% auf insgesamt 49,6 Milliarden € steigerten [PwC13].

Eine Abschwächung der damit einhergehenden Kostenexplosion wird nur durch immer umfassendere maschinelle Unterstützung im Entwicklungsprozess erzielt. Ein probates Mittel hierzu stellt die multidisziplinäre Entwurfsoptimierung dar.

Multidisziplinäre Entwurfsoptimierung (engl. multidisciplinary design optimization = MDO) basiert auf der Idee die Parameter, die in der Kontrolle des Designers (bzw. des Designteams) liegen, so zu wählen, dass sie in der gefundenen Kombination das effektivste Produkt oder System ergeben. Für die heutigen weit entwickelten Produkte ist der Designprozess eine komplexe Sequenz von Aufgaben, die sich neben ihrer eigenen Komplexität auch vor allem durch ihren interdisziplinären Charakter auszeichnen. [AHEC97]

Es stellt sich nun die Frage wie die maschinelle Unterstützung im Entwicklungsprozess vorangetrieben werden kann. Hierbei ergeben sich grundsätzlich zwei Möglichkeiten:

Zum einen die Erweiterung der Funktionalität innerhalb der Werkzeuglandschaft zur Unterstützung des Designprozesses. Ein Beispiel hierfür ist die Erhöhung der Berechnungsgeschwindigkeit eines Optimierungsproblems, durch einen neuen, schnelleren Berechnungsalgorithmus. Zum anderen gibt es die Möglichkeit die Benutzbarkeit der bestehenden Werkzeuge zu erhöhen.

Nach [Shn03:58] handelt die neue Informatik nicht mehr von der Frage „What can computers do?“, sondern vielmehr von der Frage „What can humans do with computers“. Der Flaschenhals in der Zusammenarbeit zwischen Mensch und Maschine hat sich also weg von der Prozessorleistung, hin zu der Mensch-Maschine-Schnittstelle verschoben.

> *“You don’t think about the process of how a light switch works when you turn it on. It just works. That’s what needs to happen to computers in order for people to get the most out of them.” (Donald Norman)* [Don14]

Dies ist eine Entwicklung die vor allem stark mit den Fortschritten in der Leistungsfähigkeit von Computern zusammenhängt. Erstmalig veröffentlicht wurde diese exponentielle Entwicklung von Gordon Moore in [Mo65], diese Feststellung wurde später als das Mooresche Gesetz bezeichnet. Moore geht von einer in regelmäßigen Abständen verdoppelnden Anzahl von Schaltkreiskomponenten auf einem Integrierten Schaltkreis (IC) aus, als Zeitraum für die Verdopplung nennt er (den aus heutiger Sicht zu optimistischen) Zeitraum von einem Jahr [Mo65]. Mit der exponentiellen Entwicklung hat er jedoch bis in die heutige Zeit hinein Recht behalten.

Heutige Rechner sind mit den gängigen Optimierungsmethoden in der Lage beliebig komplexe Optimierungsprobleme zu lösen, es ist hierbei nur wichtig, dass der Lösungsraum nicht zu groß wird. Dies lässt sich einfach damit begründen, dass es selbst beim kompliziertesten Problem ausreicht alle Möglichkeiten durchzuspielen. Viel schwieriger dagegen scheint es ein Problem Computer gerecht zu übersetzen, also in eine computergerechte mathematische Form zu bringen. Genau hier setzt diese Arbeit an.

1.1 Thema

Aufgrund der großen Bedeutung von Entwurfsoptimierungen im industriellen Alltag, kommt der Benutzbarkeit von Entwurfsoptimierungslösungen mittlerweile eine große Bedeutung zu. Hierbei spielen vor allem zwei Aspekte eine Rolle: Zum einen die Problemformulierung, die in ihrer grundlegendsten Form, der Formulierung in mathemati-

schen Gleichungen und Ungleichungen, eine erhebliche Komplexität birgt und bei weniger mathematisch versierten Personen gar zu Berührungsängsten führt. Zum anderen kann auch die Auswahl und Konfiguration eines geeigneten Optimierungsalgorithmus zu einem beliebig komplexen Problem werden.

Ziel dieser Arbeit ist es beide Aspekte abzudecken und so eine rechnerunterstützte Lösung zu schaffen, die sowohl die Problemformulierung, als auch die Auswahl und Konfiguration eines geeigneten Optimierungsalgorithmus ermöglicht.

1.2 Gliederung

Nachdem in diesem Kapitel (Einleitung) der Leser eine Einführung in das Themengebiet dieser Arbeit erhalten hat, folgen im Kapitel Hintergrund die Erläuterung einiger Basisthemen, die im weiteren Verlauf dieser Arbeit eine Rolle spielen. Das Kapitel Aufgabenstellung beschreibt die Aufgabenstellung dieser Arbeit und geht darauf ein. Im 4. Kapitel (Stand der Wissenschaft) werden verwandte Arbeiten vorgestellt und damit einen Überblick über relevante Arbeiten gegeben, die sich wie auch diese Arbeit mit Multidisziplinärer Entwurfsoptimierung und der Unterstützung in der Problemformulierung befassen. Das 5. Kapitel (Einordnung der Aufgabenstellung und Vorgehen) ordnet die gegebene Aufgabenstellung ein und leitet daraus das Vorgehen für Entwurf und Implementierung eines universellen Tools zur Unterstützung in der Problemformulierung ab. Darauffolgend wird im 6. Kapitel (Konzept) das technologieunabhängige Konzept des entwickelten Problemformulierungstools vorgestellt, dieses wird im 7. Kapitel (Implementierung) praktisch umgesetzt. Das Ergebnis ist ein Tool zur Unterstützung der Problemformulierung von Optimierungsproblemen und wird im 8. Kapitel auf seine Lösung der Problemstellung evaluiert. Das 9. Kapitel (Fazit) ordnet die Arbeit und das erstellte Tool zusammenfassend ein. Den Abschluss bildet das 10. Kapitel (Zusammenfassung und Ausblick), welches eine kurze Zusammenfassung der Arbeit und einen Ausblick auf mögliche, auf dieser Arbeit aufbauende, Arbeiten gibt.

2 Hintergrund

In diesem Kapitel werden die Grundlagenthemen erläutert auf denen diese Arbeit aufbaut. Im Abschnitt 2.1 wird zunächst der industrielle Entwicklungsprozess formal betrachtet und die Rolle der Entwurfsoptimierung darin eingeordnet. Anschließend werden in Abschnitt 2.2 Optimierungsprobleme formal definiert. Abschnitt 2.3 befasst sich mit Optimierungsverfahren zum Lösen der im vorigen Abschnitt definierten Optimierungsprobleme. Den Abschluss bildet eine Zusammenstellung von für diese Arbeit relevanten Technologien in Abschnitt 2.4.

2.1 Industrieller Entwicklungsprozess

Im folgenden Abschnitt wird der industrielle Entwicklungsprozess gemäß VDI-Norm 2221 für die „Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte [VDI93]“ vorgestellt. Der Bezug zur Entwurfsoptimierung wird mittels VDI-Norm 2211 für die „Informationsverarbeitung in der Produktentwicklung Berechnungen in der Konstruktion [VDI03]“ hergestellt.

Bei der Herstellung von wettbewerbsfähigen technischen Produkten ist der Entwicklungs- und Konstruktionsprozess ein wesentlicher Bestandteil. Die VDI-Richtlinien (Verein Deutscher Ingenieure) versuchen diesen vielschichtigen Prozess durch die Strukturierung wesentlicher Zusammenhänge und der Ableitung von konkreten Arbeitsanleitungen zu formalisieren. Die Richtlinien sind stets von allgemeingültiger Beschreibung und bleiben damit unabhängig von branchenabhängigen Details. [VDI93]

Im Produktkreislauf (siehe Abbildung 1) folgt die Entwicklung direkt auf die Produktplanung. Die Produktentwicklung nimmt hierbei eine zentrale Bedeutung für den weiteren Produktentstehungs-, -nutzungs- und -recyclingprozess ein, da von ihr die wesentlichen Merkmale des Produkts bestimmt werden. Gleichzeitig ist für eine erfolgreiche Produktentwicklung ein funktionierender Informationsfluss anderer Phasen zurück in die Entwicklung eine wesentliche Voraussetzung. [VDI93]

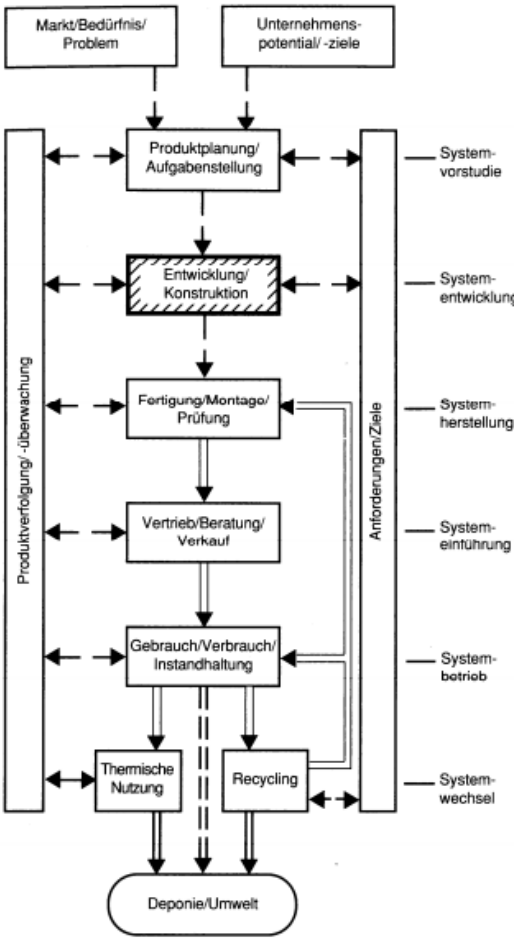


Abbildung 1: Produktkreislauf mit Produktentstehungs- und Produktlebensphasen [VDI93]

Die Entwurfsoptimierung gewinnt in der Entwicklung durch die stetig steigende Rechenleistung zunehmend an Bedeutung. Dies hängt damit zusammen, dass heute Berechnungen in einem Bruchteil der Zeit, die diese noch vor einigen Jahren beansprucht hätten, ausgeführt werden können. Ein wesentlicher Teil der Konstruktion ist und bleibt allerdings die Intuition des Konstrukteurs. Somit ist die Entwurfsoptimierung eher als Werkzeug im Entwicklungsprozess zu sehen und nicht etwa als Ersatz desselben. [And00]

Bevor ein Optimierungsverfahren angewandt werden kann, muss stets eine Sensitivitätsanalyse durchgeführt werden, die den Zusammenhang zwischen problembeschreibenden Größen und z.B. den Geometrieparametern herstellt. Danach kann, da nun der Einfluss von Parameteränderungen auf die beteiligten Funktionen bekannt ist, das Problem mathematisch formuliert werden und mithilfe eines geeigneten Optimierungsverfahrens ein verbessertes Design gefunden werden. [VDI03]

Die Entwurfsoptimierung kann ebenfalls als Phasenmodell formalisiert werden. Die Durchführung einer Entwurfsoptimierung erfordert hierbei mindestens die folgenden Schritte:

- Identifizierung von Designvariablen
- Definition Optimierungsfunktion(en)
- Definition von Restriktionen
- Auswahl eines Optimierungsalgorithmus
- Konfiguration des Optimierungsalgorithmus
- Durchführung der Berechnungen
- Interpretation der Ergebnisse

Abbildung 2 zeigt ein ähnliches aber etwas detaillierteres Phasenmodell des Entwurfsoptimierungsprozesses.

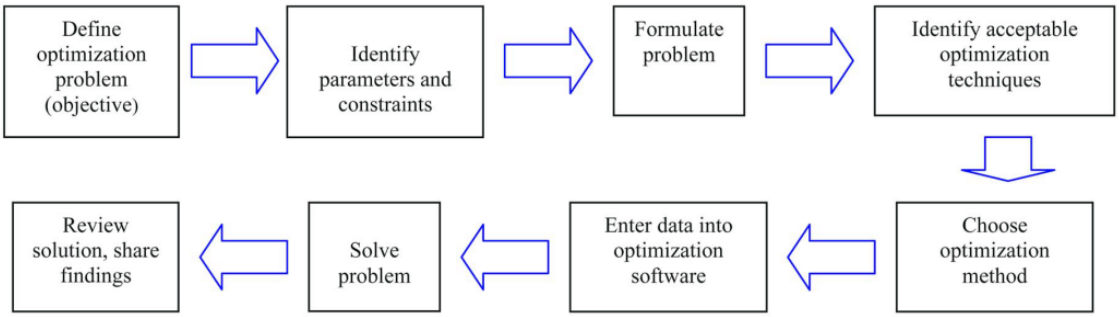


Abbildung 2: Ablaufdiagramm des industriellen Entwurfsoptimierungsprozesses [WKG07]

2.2 Optimierungsprobleme

Ein Optimierungsproblem lässt sich wie folgt definieren:

$$\min F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_k(x) \end{pmatrix}$$

$$such\ that\ (s.t.)\ x \epsilon S$$

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Formel 1: Formale Definition eines Optimierungsproblems nach [And00]

Wesentlicher Teil sind die Zielfunktionen, zusammengefasst im Vektor $F(x)$. Diese Funktion bildet den Suchraum S in einen Lösungsraum ab. Abbildung 3 zeigt die Abbildung einer zweidimensionalen Problemstellung auf einen zweidimensionalen Lösungsraum, definiert durch zwei Zielfunktionen. Der Parameter x ist hierbei wiederum ein Vektor, dieser enthält die Designvariablen. Die Menge S legt hierbei gleichzeitig den Wertebereich der Variablen fest und definiert welche Wertekombinationen zu einer gültigen (engl. feasible) Lösung führen. Anwendungsnaher ist jedoch eine Definition in der diese beiden Einschränkungen getrennt voneinander formuliert werden indem gültige Lösungen mittels Restriktionen ausgefiltert werden. [And00]

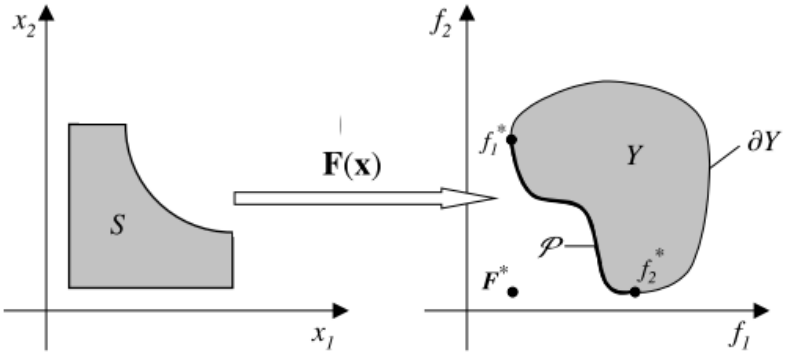


Abbildung 3: Visualisierung der Abbildung eines 2-dimensionalen Suchraums in einen 2-dimensionalen Lösungsraum [And00]

Ziel ist es x so zu wählen, dass $F(x)$ minimal wird. Während diese Problemstellung bei $k = 1$ zu einer eindeutigen Lösung für $F(x)$ führt, ist die Lage bei $k > 1$ nicht mehr eindeutig. Es kann dann nämlich zu Lösungen $x \neq x'$ kommen für die gilt $(x \nleq x') \wedge (x \ngeq x')$, da kein Vektor über den jeweils anderen Vektor dominiert. Die Vektoren lassen sich also nicht vergleichen und dem Benutzer müssen so all diese Lösungen als Ergebnis geliefert werden. Ist eine Menge an Ergebnissen als Optimierungsergebnis nicht gewünscht, müssen die verschiedenen Zielfunktionen zu einer Zielfunktion, z.B. als gewichtete Summe, zusammengefasst werden. Ein Vektor x dominiert einen Vektor x' $(x \succ x')$, wenn

$$(\forall i \in \{1,2,\dots,k\}: f_i(x) \leq f_i(x')) \wedge (\exists j \in \{1,2,\dots,k\}: f_i(x) < f_i(x'))$$

Formel 2: Kriterium für die Domination von Vektoren

gilt. [And00]

Eine Menge von Lösungen, die sich gegenseitig nicht dominieren, bildet die Menge der sogenannten Pareto-Optimalen-Lösungen. Diese Menge ist die Lösungsmenge, die mithilfe von Optimierungsverfahren gesucht wird.

2.3 Optimierungsverfahren

Optimierungsverfahren lassen sich in derivative und in nicht derivative Verfahren einteilen. Diese Arbeit konzentriert sich auf nicht derivative Verfahren, da diese auf komplexe Probleme besser anwendbar sind. Hier werden keine differenzierten Zielfunktionen zur Berechnung der optimalen Lösung benötigt. Ein weiterer Vorteil von nicht derivativen Verfahren ist, dass es mit diesen leichter ist globale Optima zu finden. Derivative Verfahren finden zwar schneller lokale Optima, sind aber nicht fähig in bestimmten Fällen globale Optima zu finden. [And00]

In diesem Abschnitt werden exemplarisch zwei populäre nicht derivative Verfahren vorgestellt. Jedes dieser Verfahren besitzt wiederum verschiedene konkrete Algorithmen, die diesen Ansatz umsetzen.

2.3.1 Randomisierte Suche

Die Randomisierte Suche (engl. random search) ist ein generischer Ausdruck für sämtliche Algorithmen, die basierend auf den Werten eines Zufallsgenerators den Suchraum nach Lösungen durchsuchen. Der Vorteil dieser Algorithmen ist eine leichte Implementierbarkeit und der geringe Overhead zur Generierung neuer Lösungsinstanzen. Nachteil ist die langsame Konvergenz. [And00]

Ein populärer Algorithmus der Randomisierten Suche ist der Random Walk. Dieser startet von einem Startpunkt (kann ebenfalls randomisiert bestimmt werden) und bewegt sich dann iterativ von diesem Startpunkt zu Punkten weiter, deren Zielfunktionswert besser ist. Die Richtung der Bewegung wird randomisiert bestimmt. Kann kein solcher Punkt im Suchraum gefunden werden, wird die Schrittgröße verringert und weiter gesucht. Dies geschieht solange bis die Schrittgröße unter eine bestimmte Schranke fällt oder ein anderes Terminierungskriterium erreicht wird. [And00]

2.3.2 Evolutionäre Algorithmen

Zu den populärsten nicht derivativen Verfahren zählen die evolutionären Algorithmen. Evolutionäre Algorithmen basieren auf der Idee der natürlichen Selektion und der Fortpflanzung von erfolgreichen Individuen. Jeder Optimierungsparameter stellt hierbei ein Gen dar, alle Gene zusammen ergeben ein Chromosom, welches ein Individuum definiert. Die besten Individuen werden gepaart und produzieren durch die Kombination ihrer Gene Nachkommen. Die Nachkommen werden in die neue Generation eingefügt

und der Prozess beginnt von neuem. Auch hier lassen sich verschieden Kriterien für die Terminierung definieren. [And00]

Genetische Algorithmen sind sehr robust und konvergieren meist sehr schnell [And00]. Darüber hinaus sind genetische Algorithmen oft umfangreich konfigurierbar, z.B. durch die Populationsgröße, die Nachkommen pro Generation oder der Anzahl gepaarter Individuen pro Generation.

2.4 Verwendete Technologien

Der folgende Abschnitt geht auf Technologien ein, die bei der Implementierung des Problemformulierungstools eine Rolle spielen. Dies beinhaltet die Programmiersprache (Java), die Entwicklungsumgebung (Eclipse), einer Systemarchitektur für Verteilte Systeme (Client-Server-Architektur), einer Programmsystemarchitektur (Expertensysteme) verschiedene Webtechnologien (Web Services, Google Web Toolkit, Google App Engine, Semantisches Web, Ontologien und Web Ontology Language (OWL)), Modellierungstools (Protégé) und Programmier-Frameworks (Apache Jena, Opt4J und Guava Framework).

2.4.1 Java

Die Entwicklung des Tools erfolgte in der Programmiersprache Java[1] (unter Verwendung des Google Web Toolkits (GWT)).

Die Programmiersprache Java wird seit 1991 von Sun Microsystems (heute Oracle[2]) entwickelt. Der ursprüngliche Name war „Oak“, welcher 1995 in Java umbenannt wurde. Syntaktisch basiert Java stark auf C bzw. C++. Eine große Stärke von Java gegenüber C++ ist die Plattformunabhängigkeit, so kann derselbe Programmcode inklusive GUI auf unterschiedlichen Plattformen ausgeführt werden. [Cla06:329]

2.4.2 Eclipse

Eclipse[3] ist eine Entwicklungsumgebung oder genauer eine IDE (Integrated Development Environment). Mit Eclipse kann in verschiedensten Programmiersprachen, wie Java, PHP oder C++, entwickelt werden. Der Eclipse Marketplace stellt einen Markt-

[1] http://www.oracle.com/de/technologies/java/overview/index.html
[2] http://www.oracle.com
[3] http://www.eclipse.org/ide/

platz für verschiedenste Plugins dar und bietet somit eine vielfältige Möglichkeit zur Erweiterung der Entwicklungsumgebung.

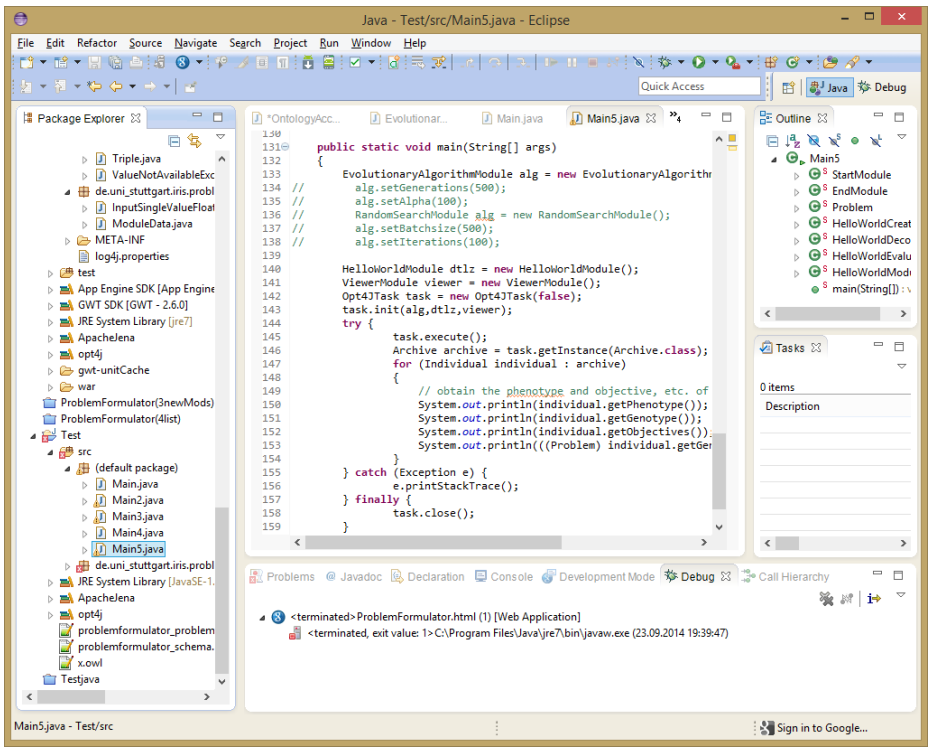


Abbildung 4: Screenshot der Eclipse IDE bei der Java-Entwicklung

Mit dem Google Plugin for Eclipse[4] lassen sich mit Eclipse Cloud-Basierte-Anwendungen erstellen und auf einfache Weise Anwendungen auf Google App Engine (siehe Abschnitt 2.4.7) bereitstellen.

2.4.3 Client-Server-Architektur

Die Client-Server-Architektur, auch 2-Tier-Architektur genannt, baut auf der Zentralrechner-Architektur auf. Bei der Zentralrechner-Architektur gibt es einen zentralen Rechner an den eine Reihe von Endgeräten angeschlossen ist. Die Verarbeitung der Daten der Endgeräte findet im Zentralrechner statt. In den 80er Jahren wurden die Endgeräte selbst zu Rechnern, die einfache Aufgaben wie Texterstellung, Ausführung von Programmen oder die Unterstützung der Arbeitsabläufe, selbst ausführen konnten. Der Zentralrechner behielt die Aufgabe seltener benötigte oder umfangreichere Aufgaben, wie die Verwaltung von Datenbeständen, auszuführen. Die Endgeräte wurden so zu einer Art von Kunden, denen der Zentralrechner Dienste anbietet. Damit erschließt sich

[4] https://developers.google.com/eclipse

auch die alternative Bezeichnung für die Client-Server-Architektur, das „Kunden-Anbieter-Prinzip“. [Cla06]

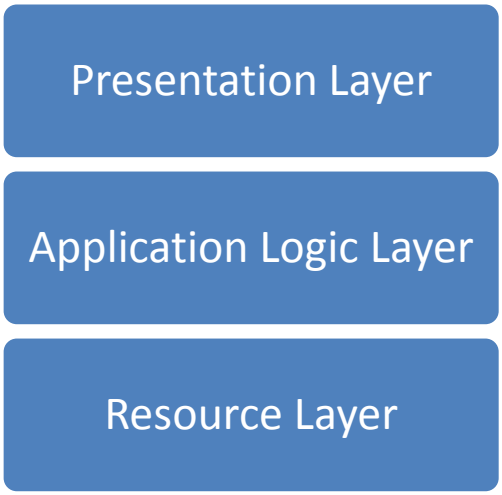


Abbildung 5: Applikationsschichten nach [WCL+05]

In Abbildung 5 sind die Schichten einer Applikation abgebildet. Diese Schichten müssen zwischen Client und Server aufgeteilt werden. Fest steht in jedem Fall, dass der Server einen Teil des Resource Layers und der Client einen Teil des Presentation Layers beinhalten muss. Wo allerdings die Grenze zwischen Client und Server genau gezogen wird, hängt vom Anwendungsfall ab. Tabelle 1 gibt einen Überblick über die möglichen Aufteilungen zwischen Client und Server. [WCL+05]

Trennung der Schichten	Name	Beschreibung	Beispiele
Presentation Layer Application Logic Layer Resource Layer	Distributed Presentation	Nur ein Teil des Presentation Layers liegt auf dem Client. Alles Weitere liegt auf dem Server.	Php-Webanwendung
Presentation Layer Application Logic Layer Resource Layer	Remote Presentation	Die komplette Ein- und Ausgabe liegt auf dem Client.	Windows GUI
Presentation Layer Application Logic Layer Resource Layer	Distributed Application	Ein Teil der Anwendungslogik liegt auf dem Client.	Java Applikation, die auf dem Client, läuft stellt Anfragen an eine andere Java-Applikation auf dem Server.
Presentation Layer Application Logic Layer Resource Layer	Remote Data	Nur die Daten liegen auf dem Server.	Applikation die über SQL auf eine Datenbank zugreift
Presentation Layer Application Logic Layer Resource Layer	Distributed Data	Ein Teil der Daten liegt auch auf dem Client.	Lotus Notes

Tabelle 1: Mögliche 2-Tier-Aufteilungen der Applikationsschichten nach [WCL+05]

2.4.4 Expertensysteme

Ein Expertensystem ist ein Programmtyp, der Wissen in einer Wissensdatenbank ansammelt, um daraus zu konkreten Problemen Lösungen anzubieten. Das Wissen wird hierbei interaktiv in den Problemlösungsprozess eingebracht. Expertensysteme finden oft Anwendung, wenn allgemeine Problemlösungsansätze aufgrund des Facettenreichtums der Problemstellung scheitern. Durch die dem Expertensystem zugrundeliegende Wissensdatenbank lässt sich Wissen fallspezifisch speichern um den Benutzer zielgerichtet zu assistieren. [Cla06:243ff.]

2.4.5 Web Services

Gemäß der W3C-Definition [BHM04] ist ein Web Service ein Software-System, das interoperable Maschine-Zu-Maschine-Interaktionen über ein Netzwerk erlaubt. Ein Web Service besitzt eine Schnittstellendefinition, die ebenfalls in einem maschinenlesbaren Format (z.B. in der Web Service Description Language (WSDL)) geschrieben ist.

2.4.6 Google Web Toolkit (GWT)

Das Google Web Toolkit[5] (GWT) ist ein Entwicklungswerkzeug um browserbasierte Anwendungen zu erstellen. Es ermöglicht die Entwicklung von komplexen Anwendungen ohne Kenntnisse in den dazu üblicherweise erforderlichen Web-Technologien wie z.B. JavaScript vorauszusetzen. Erreicht wird diese Funktionalität durch einen entsprechenden Compiler, der clientseitigen Java-Programmcode in JavaScript-Programmcode umwandelt. [Goo14a]

Der serverseitig ausgeführte Code kann in seiner Form in Java ohne Umwandlung in JavaScript ausgeführt werden. Um zusätzliche Dateien für den Zugriff bereitzuhalten, reicht es nicht aus, diese wie bei anderen Java-Applikationen im Workspace bzw. im gleichen Verzeichnis abzuspeichern. Der Grund hierfür ist, dass die Applikation nicht lokal ausgeführt wird, sondern auf den Server bzw. den Client geladen werden muss. Dateien, die auf den Server geladen werden sollen, müssen im Verzeichnis „`war/WEB-INF/`“ gesammelt werden.

[5] http://www.gwtproject.org/

Clientseitig, stellt das Google Web Toolkit, neben den verfügbaren Standard Java-Klassen und primitiven Datentypen, vor allem GUI-Elemente bereit. Dateien, die auf den Client geladen werden sollen, müssen im Verzeichnis „`war/`" liegen, damit mit dem Client-Code darauf zugegriffen werden kann. Der auf dem Browser des Clients befindliche Java-Code ist ohne Zutun des Entwicklers lauffähig, da dieser automatisch in JavaScript-Code umgewandelt wird.

Der Code im Browser ist Single-Threaded, d.h. der Browser wartet bei einem RPC-Call solange bis eine Antwort vom Server empfangen wird. Deshalb sollten asynchrone Funktionsaufrufe verwendet werden.

Soll allerdings eine Methode einen Rückgabewert haben, funktioniert auch dies durch die Verwendung einer jeweils eigens geschriebenen Instanz des AsyncCallback-Interfaces.

Ein Beispiel eines solchen RPC-Aufrufs ist in Listing 1 aufgeführt.

```
someService.getSomeList(
      new AsyncCallback<List<String>>()
{
    public void onFailure(Throwable caught)
    {
        ...
    }

      public void onSuccess(List<String> result)
    {
        ...
    }
});
```

Listing 1: Codebeispiel eines RPC-Aufrufs einer mit Google App Engine implementierten Funktion

Die Kommunikation zwischen Client und Server funktioniert über RPC-Aufrufe (Remote Procedure Call Aufrufe).

Besonderheit: Damit Klassen serialisierbar sind, muss zum einen das Interface `IsSerializable` (ähnlich wie bei Java `Serializable`) implementiert werden und zum anderen auch die Klasse einen Konstruktor ohne Parameter besitzen. Dieser Konstruktor darf jedoch auch `private` sein, was einem wenn dieser lokal nicht aufgerufen wird zu einer Warnung in Eclipse führt. Fügt man diesen Konstruktor nicht ein,

führt dies erst zur Laufzeit zu einem Serialisierungsfehler, welcher jedoch nicht direkt auf das Problem hinweist.

Ein Beispiel für eine fehlerlos serialisierbare und deserialisierbare Klasse ist in Listing 2 aufgeführt.

```
public class Triple<C1, C2, C3>
            implements Serializable, IsSerializable
{

    private static final long serialVersionUID = 1L;

    ...

    @SuppressWarnings("unused")
    private Triple(){}

      ...
```

Listing 2: Codebeispiel einer fehlerfrei vom Google Web Toolkit serialisierbaren und desereialisierbaren Klasse um diese bei RPC-Aufrufen zu verwenden.

Eine weitere Besonderheit, die sich nicht von selbst erschließen lässt ist, dass GWT-GUI-Code nicht auf dem Server benutzt werden kann. So ist es also nicht möglich z.B. ein Widget serverseitig dynamisch zu erstellen und dann zum Client zu schicken.

2.4.7 Google App Engine (GAE)

Google App Engine (GAE) ist eine Plattform zum Hosten von Web Services und Webanwendungen (z.B. mit Google Web Toolkit entwickelt) auf der Server-Infrastruktur von Google. Ziel ist es das Hosten einer Web-App so einfach wie möglich zu gestalten und die Entwickler anschließend dabei zu unterstützen eine stets ausreichende Infrastruktur für einen wachsenden Benutzerstamm bereitzustellen.

Auf eine Stolperfalle trifft man bei der Verwendung von Nebenläufigkeit anhand von Threads. Google App Engine erlaubt keine Standard-Java-Threads. Soll etwas nebenläufig ablaufen, muss der Thread über den Thread-Manager der Google App Engine (`com.google.appengine.api.ThreadManager.currentRequestThreadFactory()`) geschehen und nicht wie im reinen (engl. „plain“) Java üblich über `new Thread()`. Dass Objekte, die über den Google Thread Manager zu einem Thread gemacht werden, das Interface `Runnable` unterstützen ist dagegen identisch mit dem reinen Java. [Goo14]

Zu den Besonderheiten beim Erstellen von Threads kommen Ressourcen-Einschränkungen. So ist jeder Request auf 50 Threads beschränkt [Goo14]. Beim Versuch mehr Threads zu erstellen wird eine `IllegalStateException` geworfen. Thread-Aufgaben, die längere Zeit zum Berechnen benötigen und nicht innerhalb weniger Sekunden abgeschlossen werden können, werden terminiert um eine Überlastung der Server zu verhindern [Goo14].

2.4.8 Semantisches Web

Ziel des semantischen Webs ist es sämtliche Inhalte des Internets für Computer interpretierbar zu machen. Technologien um dieses Ziel umzusetzen sind XML, RDF und OWL. XML ist hierbei das Konzept um die Web-Informationen für Computer interpretierbar zu machen. Mit RDF ist es möglich Entitäten durch Beziehungen untereinander zu beschreiben. OWL ist eine Erweiterung von RDF um logische Aussagen objektorientiert zu modellieren. [RGKW09]

2.4.9 Ontologien

Der Begriff der Ontologie stammt ursprünglich aus der Philosophie, in der die Ontologie ein methodisches Zeugnis der Existenz von etwas darstellt. In der Informatik stellt eine Ontologie eine Menge von Begrifflichkeiten sowie deren Beziehung zueinander dar und dient als ein Konzept zur Darstellung komplexer Wissensbeziehungen. Eine Ontologie spezifiziert die Namen und Eigenschaften einer Domain und die Beziehungen zwischen ihnen. [Gru93]

2.4.10 Web Ontology Language (OWL)

Die Web Ontology Language (OWL) ist eine Ontologiesprache die auf der Prädikatenlogik erster Stufe basiert. OWL wurde 2004 vom W3C als Ontologiesprache standardisiert. Bei der Entwicklung von OWL waren sowohl eine große Ausdrucksstärke als auch effiziente Schlussfolgerungsmechanismen wichtig. [HKRS08:125] OWL besitzt die Teilsprachen OWL Full, OWL DL und OWL Lite, die jeweils eine unterschiedliche Anzahl an Sprachelementen erlauben [HKRS08:151ff.].

Bestandteile von OWL sind:

- Klassen: Hauptbestandteil von OWL sind Klassen. Diese können mittels `subClassOf` zueinander in Beziehung gesetzt werden. [HKRS08:129ff.]

- Properties: OWL unterscheidet zwischen diesen Hauptkategorien an Properties
    - Object Properties: Verbinden ein Individual mit einem anderen Individual [BvHH04]
    - Data Properties: Verbinden ein Individual mit einem Datenwert [BvHH04]
    - Annotation Properties: Verbinden eine beliebige Entität (Individual, Klasse, Property, usw.) mit Ausnahme von einer Auswahl von reservierten Entitäten mit einem Wert (Data Value, Instance, Klassen oder Properties). Annotation Properties bleiben im Reasoning Prozess unberücksichtigt. In OWL DL sind Annotation Properties, Datatype Properties und Object Properties paarweise disjunkt. In OWL Full dagegen kann willkürlich ein Datatype Property oder ein Object Property auf einer Klasse oder einem Property benutzt werden. [BvHH04]
- Individuals: Sind Instanzen die als Typ eine Klasse haben können und sich mit weiteren Properties bestücken lassen.

In dieser Arbeit wird eine OWL-Modellierung ausgelesen um sie in Java möglichst ähnlich nach zu modellieren. Dabei kann es Probleme geben, wenn die Sprachen unterschiedliche Arten von Beziehungen zulassen. Eine solche Diskrepanz besteht in der Mehrfachvererbung - während sie in OWL erlaubt ist, ist sie in Java nicht möglich. Dies hat dazu geführt, dass in dieser OWL-Modellierung auf Mehrfachvererbungen verzichtet wurde. In der Java-Modellierung sind zwar grundsätzlich Möglichkeiten vorhanden diese Beziehungen anders zu implementieren, allerdings wurde darauf verzichtet, um die Modulentwicklung so einfach wie möglich zu gestalten.

2.4.11 SPARQL

SPARQL ist eine Abfragesprache des semantischen Webs. SPARQL erlaubt die Abfrage von strukturierten und teilstrukturierten Daten. Durch SPARQL-Abfragen lassen sich auch bisher unbekannte Zusammenhänge erschließen. [Fei13]

SPARQL-Abfragen liefern als Ergebnis RDF-Datensätze, diese bestehen immer aus Tripeln aus Subjekt, Prädikat und Objekt. Die Abfragen in SPARQL sind deshalb auch in dieser Tripel-Form aufgebaut. Eine Abfrage kann allerdings durchaus aus mehreren Tripeln bestehen, diese Tripeln werden mit dem Operator „`.`“ konkateniert. Eine weitere Möglichkeit der Konkatenation bietet der Operator „`;`“, dieser stellt eine Möglichkeit

zur Abkürzung dar, indem das Subjekt des anschließenden Tripels weggelassen werden kann.

Ein Beispiel einer SPARQL-Abfrage ist in Listing 3 dargestellt.

```
PREFIX type: <http://dbpedia.org/class/yago/>
PREFIX prop: <http://dbpedia.org/property/>
SELECT ?country_name ?population
WHERE {
    ?country a type:LandlockedCountries ;
             rdfs:label ?country_name ;
             prop:populationEstimate ?population .
    FILTER (?population > 15000000 &&
            langMatches(lang(?country_name), "EN")) .
} ORDER BY DESC(?population)
```

Listing 3: Beispiel SPARQL-Abfrage, die alle von Land eingeschlossenen Länder mit einer Einwohnerzahl größer 15 Millionen, sortiert nach der Einwohnerzahl zurückgibt. [Fei13]

2.4.12 Protégé

Protégé[6] ist ein Java-basiertes Open-Source-Tool zur Entwicklung von Ontologien. Diese Ontologie-Entwicklungsumgebung wurde von der Universität in Stanford entwickelt. [WKG07]

Protégé bietet eine grafische Benutzeroberfläche durch die es möglich ist, auch ohne Kenntnisse der Syntax von Ontologie Sprachen wie RDF oder OWL, eigene Ontologien zu erstellen [NSD01]. Trotzdem sollten dem Benutzer die Grundzüge der modellierenden Sprachen und Ontologien im Allgemeinen bekannt sein. Aufgrund der gewachsenen Sprachenvielfalt im Bereich der Ontologien und des semantischen Webs gewinnen Tools wie Protégé, die von der reinen Syntax abstrahieren, an Bedeutung [NSD01].

[6] http://protege.stanford.edu/

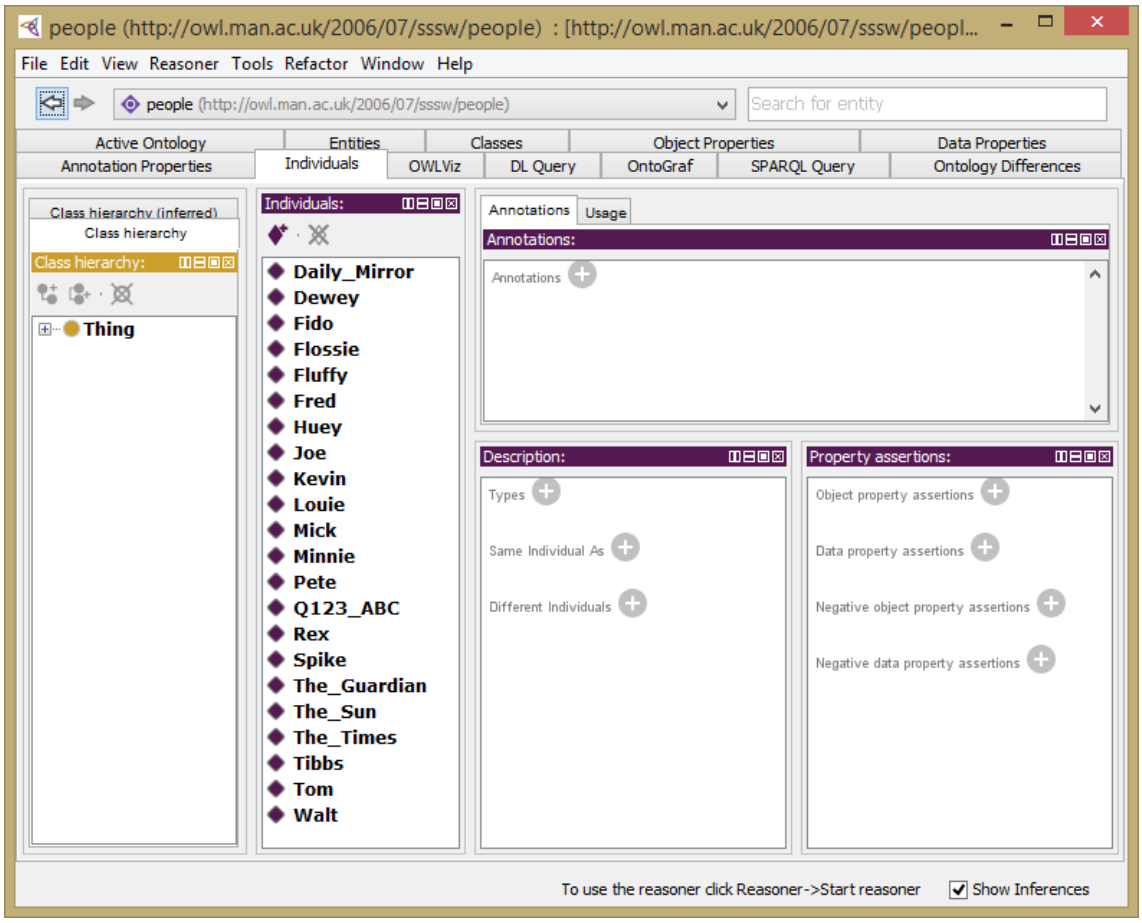


Abbildung 6: Screenshot der Ontologie-Entwicklungsumgebung Protégé

2.4.13 Apache Jena

Apache Jena[7] ist ein Open-Source-Framework um Software im Kontext des semantischen Webs und vernetztem Wissen zu erstellen. Das Framework ist eine Zusammenstellung aus mehreren Application Programming Interfaces (APIs) die in ihrem Zusammenspiel RDF-Daten verarbeiten. [Apa14]

Eine Alternative ist OpenRDF Sesame[8]. Die Entscheidung fiel letztendlich zugunsten Apache Jenas, da damit durch die Einbeziehung von Jena Rules mehr Flexibilität bezüglich der Ausgestaltung der Modellierung geboten wird. Jena Rules bietet die Möglichkeit eigene Regeln für den Reasoner zu erstellen und so frei über die Logik des Reasoners zu entscheiden. Ein weiterer Grund ist die Performance, die wie in [HMRSZ11] dargestellt, bei Apache Jena etwas höher ist. Die Unterschiede in der Performance sind jedoch minimal.

[7] https://jena.apache.org/
[8] http://www.openrdf.org/

2.4.13.1 Laden einer Ontologie

Um auf eine Ontologie zugreifen zu können, muss diese erst in ein `InfModel` geladen werden. An das `InfModel` können dann SPARQL-Abfragen gestellt werden. Listing 4 zeigt wie man eine Ontologie in ein `InfModel` mit dem Apache-Jena-Framework laden kann.

```
01 public static InfModel loadOntologyFromFile(String sourceFile,
02                                             String schemaFile)
03 {
04    Model schema = FileManager.get()
05          .loadModel("file:"+schemaFile);
06    Model data = FileManager.get()
07          .loadModel("file:"+sourceFile);
08
09    Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
10    reasoner = reasoner.bindSchema(schema);
11    InfModel infModel =
12          ModelFactory.createInfModel(reasoner, data);
13    return infModel;
14 }
```

Listing 4: Laden einer Ontologie und des passenden Schemas aus XML-Dateien mit dem Apache-Jena-Framework

2.4.13.2 SPARQL-Requests an eine Ontologie stellen

Um einen Request an eine Ontologie zu stellen, muss ein String mit der SPARQL-Query mithilfe der `QueryFactory` in ein Query-Objekt verwandelt werden. Das Query-Objekt wird dann mithilfe der `QueryExecutionFactory` in eine `QueryExecution` umgewandelt. Aus dem `QueryExecution`-Objekt lässt sich dann ein `ResultSet` erstellen über das alle Ergebnisse der Abfrage zugänglich sind.

Ein Fallstrick stellt hierbei das Problem dar, dass das `QuerryExecution`-Objekt erst mit der Funktion `close()` geschlossen werden darf, wenn vom `ResultSet` alle Daten abgefragt wurden. Danach ist das `ResultSet` leer (und gibt somit vor, dass die Abfrage keine Ergebnisse geliefert hat).

```
// Create a new query string
String queryString =
"SELECT ?x " +
"WHERE {" +
"       ?x ?y ?z " +
"       }";

//create query
Query query = QueryFactory.create(queryString);

// Execute the query and obtain results
QueryExecution qe = QueryExecutionFactory.create(query, model);
ResultSet results = qe.execSelect();
```

Listing 5: Java-Codeausschnitt um eine SPARQL-Query mit Apache Jena auszuführen

2.4.13.3 Jena Rules

Jena Rules ist eine eigene Regel-Engine des Apache-Programmierframework mit einer eigenen Sprache. Die Regeln arbeiten auf RDF-Daten. Regeln bestehen aus Tripeln, welche wiederum Variablen enthalten können. [Pau11]

2.4.14 Opt4J

Das Opt4J-Framework[9] ist ein Java-basiertes Framework für evolutionäre Berechnungen. Ziel von Opt4J ist es, die Evolutionäre Optimierung von selbst definierten Problemen zu vereinfachen. Ein Alternativframework stellt das ECJ-Framework[10] dar. Opt4J erlaubt die metaheuristische Optimierung von komplexen Optimierungsaufgaben. Das Framework erlaubt die separate Entwicklung von Optimierungsalgorithmen und Optimierungsaufgaben. [LGRT11]

Um ein Problem in Opt4J zu definieren, müssen die 3 Hauptklassen Creator, Decoder und Evaluator erstellt werden, welche alle wichtigen Eigenschaften des Problems enthalten. Sind diese 3 Klassen erstellt, kann der Optimierungsalgorithmus mit dem Problem arbeiten.

[9] http://opt4j.sourceforge.net/
[10] http://cs.gmu.edu/~eclab/projects/ecj/

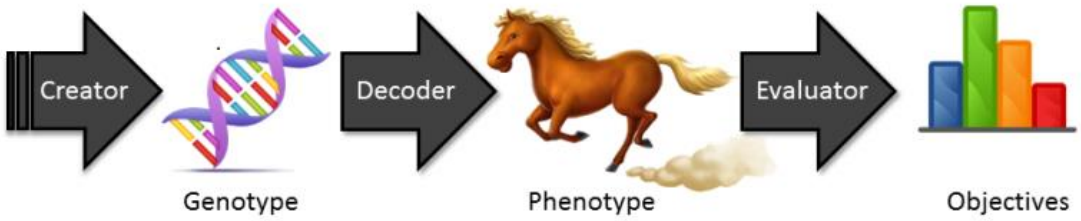


Abbildung 7: Schema des Ablaufs der Problemdefinition im Opt4J-Framework [sou14]

Der Creator generiert den Genotype, dieser stellt eine genetische Repräsentation des Problems dar. Der Decoder verwandelt den Genotype in den Phenotype dieser enthält alle messbaren Eigenschaften des Problems. Der Evaluator verwandelt schließlich den Phenotype in die Objectives, die die Qualität des Individuums repräsentieren. [sou14]

Eine Besonderheit ist in der Entwicklung zu beachten. Bei Erstellung eines Genotypes, darf dieser im Konstruktor nicht selbst `values` erstellen. Ansonsten führt dies zu nicht auf das Problem hinweisenden Laufzeit-Fehlern. Die Fehlermeldungen treten an der Stelle auf, an der der Optimierungstask mittels `task.execute();` ausgeführt wird. Das Problem liegt daran, dass der Optimierungsalgorithmus während dessen Ausführung weitere Genotypes erstellt, die jedoch zunächst leer sein müssen, da dieser die Genotypes selbst befüllt.

Ein weiteres Problem auf das man in der Entwicklung von eigenen Genotypes stoßen kann ist, dass verlangt wird, dass eine Subklasse vom Typ DoubleGenotype einen Konstruktor mit Parameter des Typs `Bounds<?>` haben muss. Auch hier tritt nur ein Fehler zur Laufzeit auf, aus dem das Problem nicht sofort ersichtlich wird. Der Grund für dieses Verhalten liegt in der Implementierung der Funktion `newInstance()`, welche in Listing 6 aufgeführt ist. Hier wird auf dynamische Weise der Konstruktor der eventuellen Subklasse aufgerufen.

```
104             /*
105              * (non-Javadoc)
106              *
107              * @see org.opt4j.core.Genotype#newInstance()
108              */
109             @Override
110             @SuppressWarnings("unchecked")
111             public <G extends Genotype> G newInstance() {
112                     try {
113                             Constructor<? extends DoubleGenotype>
cstr = this.getClass().getConstructor(Bounds.class);
114                             return (G) cstr.newInstance(bounds);
115                     } catch (Exception e) {
116                             throw new RuntimeException(e);
117                     }
118             }
```

Listing 6: Implementierung der Funktion newInstance() der Klasse DoubleGenotype aus dem opt4j Framework mit einem dynamischen Aufruf, der zu einem Laufzeitfehler führen kann.

Bei der Verwendung von Opt4J mit dem Google Web Toolkit (GWT) gibt es immer dann ein Problem wenn Opt4J versucht einen Thread zu erzeugen. Dies hängt damit zusammen, dass GWT keine Threads erlaubt, sondern Nebenläufigkeit über einen eigenen Scheduler umsetzt. Um dies in Opt4J umzusetzen, müsste allerdings der gesamte Opt4J-Quellcode überarbeitet werden, darauf wurde in dieser Arbeit verzichtet. Wird allerdings kein Optimierungsalgorithmus mit paralleler Ausführung verwendet, wird von Opt4J nur eine bereits korrekt gefangene `InvocationTargetException` in der Konsole ausgegeben, da der `ReferenceFinalizer`-Thread nicht ausgeführt werden konnte. Da dieser keine kritische Komponente von Opt4J darstellt, wird der Optimierungsprozess korrekt ausgeführt. [Yeg09]

2.4.15 Guava Framework

Das Guava-Framework[11] enthält vor allem Basisfunktionalitäten, wie String-Verarbeitung, Ein- und Ausgabeprozesse oder Nebenläufigkeitsfunktionen, für Java-basierte Projekte. [gua14]

Das Guava-Framework ist beim GWT-Framework ohnehin bereits enthalten, weshalb bei der Entwicklung dieses Tools auch darauf zugegriffen wurde, wenn es eine sinnvolle Funktionalität liefert. Z.B. bietet das Guava-Framework die Möglichkeit das Interface der `Multimap` unter Verwendung der Klasse `HashMultimap` zu nutzen.

[11] https://code.google.com/p/guava-libraries/

3 Aufgabenstellung

In diesem Kapitel wird die Problem- bzw. Aufgabenstellung dieser Arbeit beschrieben.

Die folgende Aufgabenstellung ist die Grundlage dieser Arbeit und legt somit die in den folgenden Kapiteln beschriebenen Schwerpunktaufgaben fest:

Rechnergestützte Verfahren zur Entwurfsoptimierung dienen heute in vielfältigen Konstruktionsbereichen zur automatischen Bestimmung optimaler Entwürfe für ein gegebenes Konstruktionsproblem. So kann zum Beispiel automatisch jene Brückenkonstruktion identifiziert werden, die unter einer vorgegebenen Traglast (und anderen Vorgaben) so wenig wie möglich Material benötigt. Bevor ein solches Problem jedoch an einen Lösungsalgorithmus übergeben werden kann, muss es jedoch relativ aufwändig nach folgendem Schema formal korrekt definiert werden:

$$\begin{array}{ll} x & \textit{design variables} \\ y & \textit{state variables} \\ \min\limits_{x} f(x,\ y) & \textit{objective function} \\ g(x,y) \leq 0 & \textit{inequality constraints} \\ h(x,y) = 0 & \textit{equality constraints} \end{array}$$

Formel 3: Problemformulierungsschema zur Eingabe eines Entwurfsoptimierungsproblems in einen entsprechenden Lösungsalgorithmus

Ziel dieser Masterarbeit ist zunächst ein rigoroser Vergleich existierender Ansätze zur Unterstützung der Formulierung von Entwurfsoptimierungsproblemen. Anschließend soll ein Software-Prototyp in JAVA realisiert werden, der es ermöglicht diese Probleme möglichst einfach über eine Web-Schnittstelle zu definieren und sie zur Lösungsfindung an eine JAVA-basierte Optimierungsumgebung übergibt (z.B. JOptimizer[12]). Als Plattform zur Entwicklung des Prototypens soll Google App Engine[13] dienen.

[12] http://www.joptimizer.com

[13] https://developers.google.com/appengine/

4 Stand der Wissenschaft

In diesem Kapitel wird ein Überblick über die mit diesem Thema verwandte wissenschaftliche Literatur geschaffen. Hierbei wurde bewusst ein breites Spektrum gewählt. Im Abschnitt 4.1 (Anwendungen rechnergestützter Entwurfsoptimierung) wird zunächst ein Überblick über das Spektrum der Anwendungen rechnergestützter Entwurfsoptimierung geschaffen. Darauffolgend wird im Abschnitt 4.2 (Unterstützung der Problemformulierung) auf Ansätze zur Unterstützung des Prozesses der Problemformulierung eingegangen. Im Kapitel 4.3 (Methoden zur Beschleunigung der Berechnungszeit) werden Methoden analysiert, die helfen das Problem für den Lösungsalgorithmus vorzubereiten und den Lösungsalgorithmus dann möglichst effizient bezüglich der Berechnungszeit auszuführen. Es kann beispielsweise auf die Problemstruktur eingehen um eine Laufzeitverbesserung zu erreichen.

4.1 Anwendungen rechnergestützter Entwurfsoptimierung

In diesem Abschnitt werden verschiedene Anwendungen rechnergestützter Entwurfsoptimierung betrachtet um eine Idee für mögliche Anwendungen zu geben. Rechnergestützte Entwurfsoptimierung kann in einem breiten Spektrum an Anwendungsgebieten angewandt werden, dazu zählen sowohl wissenschaftlich orientierte, als auch praktisch orientierte Problemstellungen.

4.1.1 Next Release Problem

Beim Next Release Problem geht es darum zu entscheiden welche Features im nächsten Release umgesetzt werden. Es steht eine Liste von möglichen Features bereit. Jedes Feature hat bestimmte Kosten für die Umsetzung. Neben den Features gibt es auch Kunden, die das Produkt benutzen. Jeder Kunde hat für das Entwicklungsunternehmen ein bestimmtes Gewicht (z.B. nach Anzahl der Lizenzen pro Kunde, Wachstumsperspektive der Branche oder der Kommunikationsintensität zu zukünftigen potentiellen Kunden der Branche). Das Gewicht des Kunden gewichtet wiederum die Präferenzen des jeweiligen Kunden bzgl. umzusetzender Features. Jeder Kunde gibt jedem Feature eine Gewichtung bzgl. der eigenen Wichtigkeit der Umsetzung im nächsten Release. Ziel ist es nun die Summer der Scores über alle Features, berechnet aus den Gewichten

der Kunden und deren eigenen Gewichtungen der umzusetzenden Features, zu maximieren. Gleichzeitig gilt es die Kosten der umzusetzenden Features zu minimieren [ZHM07]. Eine mathematische Formulierung der Problemstellung ist in Formel 4 dargestellt.

$$max \sum_{i=0}^{n} (score_i * x_i)$$
$$min \sum_{i=0}^{n} (cost_i * x_i)$$

$$score_i = \sum_{j=0}^{m} \left(w_j * value_{ij}\right)$$
$$value_i = \sum_{j=0}^{m} \left(w_j * value_{ij}\right)$$
$$n = |\{products\}|$$
$$m = |\{customers\}|$$
$$x_i \in \{0,1\}$$

Formel 4: Mathematische Darstellung des Next Release Problems [ZHM07]

Selbstverständlich kann das Problem noch weiter verallgemeinert werden, z.B. können noch Restriktionen zugelassen werden, wie dies in [ZHM07] beschrieben wird. Eine solche Restriktion könnte z.B. die Unternehmensstruktur berücksichtigen und dafür sorgen, dass Abteilungen mit der Feature-Auswahl nicht überlastet werden.

Eine Alternative Formulierung des Problems wird in [GR04] angewandt. Hierbei gibt es nur eine Optimierungsfunktion, was dazu führt dass es eine eindeutige optimale Lösung (bezüglich der Optimierungsfunktion) gibt und nicht eine Vielzahl von unterschiedlich bewerteten Pareto-Optimalen-Lösungen, die sich gegenseitig nicht dominieren. Anstatt die Kosten zu minimieren, wird eine Obergrenze für die Kosten des nächsten Release gesetzt. Die mathematische Formulierung dieser abgewandelten Problemstellung ist in Formel 5 aufgeführt.

$$max \sum_{i=0}^{n} (score_i * x_i)$$

$$\boldsymbol{cost_{max}} \geq \sum_{i=0}^{n} (cost_i * x_i)$$

$$score_i = \sum_{j=0}^{m} (w_j * value_{ij})$$

$$value_i = \sum_{j=0}^{m} (w_j * value_{ij})$$

$$n = |\{products\}|$$

$$m = |\{customers\}|$$

$$x_i \in \{0,1\}$$

Formel 5: Mathematische Darstellung eines alternativen Next Release Problems gemäß [GR04]

4.1.2 Flächennutzungsoptimierung (Land Use Optimization)

Eine andere mögliche Anwendung rechnergestützter Entwurfsoptimierung stellt die Flächennutzungsoptimierung dar. Gesucht wird ein optimales Szenario für die Landflächennutzung eines festgelegten Gebietes. Eine Ausgestaltung eines entsprechenden Optimierungsproblems ist in [CBH+11] beschrieben.

Aufgrund der Komplexität des Problems in der Praxis, mit all seinen möglichen Facetten, muss für die Formulierung als computerlesbares Optimierungsproblem zum Teil stark abstrahiert werden. So wird in [CBH+11] zunächst das verfügbare Land in ein Raster aus N Zeilen und M Spalten eingeteilt.

Wenn diese Einteilung gemacht ist, sind die möglichen Nutzungsmöglichkeiten für ein Feld im Raster des Gebietes zu definieren. Allgemein lässt sich formulieren es gibt K Flächennutzungen. Daraus resultierend lässt sich eine Nutzenvariable für jedes Feld (i, j) und jede Nutzung k als B_{ijk} definieren. B_{ijk} gibt den Nutzenwert wieder, wenn Feld (i, j) dem Verwendungszweck k zugeführt wird. Synonym dazu lässt sich eine binäre Designvariable x_{ijk} definieren, die wiedergibt, ob Feld (i, j) dem Verwendungszweck k zugeführt wird. Zusätzlich lässt sich noch leicht eine obere (U_k) und untere (L_k) Grenze für jede Flächennutzung definieren. Eine mathematische Formulierung des Problems zeigt Formel 6. [CBH+11]

$$max \sum_{k=1}^{K} \sum_{i=1}^{N} \sum_{j=1}^{M} (B_{ijk} * x_{ijk})$$

$$subject\ to:$$
$$x_{ijk} \in \{0,1\}, \forall k = 1, \dots, K; i = 1, \dots, N; j = 1, \dots, M$$
$$\sum_{k=0}^{K} x_{ijk} = 1\ , \forall i = 1, \dots, N; j = 1, \dots, M$$
$$\sum_{i=1}^{N} \sum_{j=1}^{M} x_{ijk} \leq U_k, \forall k = 1, \dots, K$$
$$\sum_{i=1}^{N} \sum_{j=1}^{M} x_{ijk} \geq L_k, \forall k = 1, \dots, K$$

Formel 6: Mathematische Darstellung des Flächennutzungsoptimierungsproblems in Anlehnung an [CBH+11]

Die Beschränkungen in Formel 6 sorgen dafür, dass die x_{ijk} nur sinnvoll gesetzt werden können. So sorgt die erste Restriktion dafür, dass die Design-Variablen binär sind. Die zweite dafür, dass jedem Feld nur einer Flächennutzung zugeführt wird. Die dritte Restriktion sorgt dafür dass die Obergrenze für die Verwendung von jedem Land Use k eingehalten wird und die letzte Restriktion dafür, dass die entsprechende untere Grenze eingehalten wird.

Validiert wurde das Ganze in [CBH+11] anhand einer Fallstudie des Stadtteils Tongzou New Town der chinesischen Stadt Tongzhou. Die Stadt Tongzhou befindet sich südöstlich von Peking. Die zu optimierende Fläche erstreckt sich über eine Fläche von 906 km². Es wurde eine Auflösung von 400 m auf 400 m für die Größe der Zellen gewählt. Es wurden fünf Flächennutzungen definiert [CBH+11]:

- Wohnbereiche
- Industriebereiche
- Gewerbebereiche
- Freie (offene) Bereiche
- Unerschlossene Bereiche

Dazu werden drei Optimierungszielsetzungen benannt [CBH+11]:

- Minimierung der Umwandlungskosten der Flächennutzungen
- Maximierung der Erreichbarkeit
- Maximierung der Kompatibilität von aneinander angrenzenden Flächennutzungen

Für jede dieser Optimierungszielsetzungen sind umfangreiche weitere Parameterdefinitionen nötig. Für deren genaue Definition und Zusammensetzung sei für die exakte Optimierungsfunktionen auf [CBH+11:10 ff.] verwiesen.

Neben den in Formel 6 genannten Restriktionen sind noch weitere Restriktionen möglich. So ist es z.B. sinnvoll Wohnbereiche nach unten zu beschränken, so dass auch für ein zukünftiges Bevölkerungswachstumsszenario immer genügend Wohnraum zur Verfügung steht. Des Weiteren gibt es z.B. wie am Beispiel von Tongzhou New Town einen Kanal durch die Stadt der nicht bebaut werden kann. All diese Dinge können in weiteren zusätzlichen Restriktionen berücksichtigt werden. [CBH+11]

Aufgrund des Facettenreichtums und des 2-dimensionalen Modells im Hintergrund, ist die Umsetzung in einem universellen Tool für dieses Problems als zu umfangreich und die Benutzung desselben wenig praktikabel.

4.1.3 Produktionsprogrammplanungs-Optimierung

Ein weiteres Beispiel für ein in der Praxis relevantes Optimierungsproblem ist die Berechnung des optimalen Produktionsprogramms. Das Ziel hierbei ist es ein optimales Produktionsprogramm zu berechnen, das den Gewinn bzw. den Deckungsbeitrag maximiert. Alternativ können auch andere betriebswirtschaftliche Messgrößen optimiert werden, z.B. wie in [SF09] eine Maximierung der Rentabilität. Die Deckungsbeitragsmaximierung ist hierbei gleichbedeutend mit der Gewinnmaximierung, da die Gewinnfunktion lediglich die Deckungsbeitragsfunktion dahingehend erweitert, dass der konstante Faktor der Fixkosten mit einfließt. Dass es sich lediglich um einen konstanten Faktor handelt, wird durch Vergleich von Formel 7 und Formel 8 deutlich.

$$G = E - K_{Ges}$$
$$E = \sum_{i=1}^{n} (e_i * x_i)$$
$$K_{Ges} = K_f + K_v$$
$$K_v = \sum_{i=1}^{n} (k_{v_i} * x_i)$$
$$\boldsymbol{G = \sum_{i=1}^{n} (e_i * x_i - k_{v_i}) - K_f}$$
$$G \ldots Gewinn$$
$$E \ldots Gesamterlös$$
$$K_{Ges} \ldots Gesamtkosten$$
$$K_f \ldots Fixkosten$$
$$K_v \ldots Variable\ Kosten$$
$$e_i \ldots Erlös\ pro\ Stück\ Produkt\ i$$
$$x_i \ldots Produktionsmenge\ Produkt\ i$$
$$k_{v_i} \ldots variable\ Stückkosten\ von\ Produkt\ i$$

Formel 7: Betriebswirtschaftliche Gewinnfunktion

$$D = E - K_v$$
$$\boldsymbol{D = \sum_{i=1}^{n} (e_i * x_i - k_{v_i})}$$
$$D \ldots Deckungsbeitrag$$

Formel 8: Betriebswirtschaftliche Deckungsbetragsfunktion

Ein Produktionsprogramm enthält die Belegungen für die Variable x_i für alle i produzierbaren Produkte. Das optimale Produktionsprogramm enthält demnach die optimale Belegung der x-Variablen, um z.B. den Deckungsbeitrag zu maximieren. Die Produktionsmengen x_i unterliegen allerdings noch bestimmten Beschränkungen, ansonsten ergäbe sich bei positiven Stückdeckungsbeiträgen die triviale Lösung unendlich viel zu produzieren. Deshalb werden im Allgemeinen bei diesem Optimierungsproblem folgende Nebenbedingungen verwendet [BSW12]:

- Kapazitätsbeschränkungen: Jedes Produkt benötigt bestimmte Ressourcen (Maschinenstunden, Arbeitsstunden, Material, …). Jede dieser Ressourcen ist nach oben beschränkt.
- Absatzbeschränkungen: Der Markt lässt nur einen bestimmten maximalen Absatz eines Produktes zu. Es macht also keinen Sinn mehr vom Produkt herzustellen, da dieser Überschuss dann nicht abgesetzt werden könnte. Also werden diese Obergrenzen auf die Produktionsmengen x_i angewandt.

- Nichtnegativitätsbedingung: Stellt sicher, dass die Lösung keine negativen Produktionsmengen enthält, welche in der Praxis nicht möglich sind.

Aus der Maximierung der Deckungsbeitragsfunktion und der mathematischen Formulierung der Nebenbedingungen ergibt sich das vollständige mathematische Optimierungsproblem [BSW12]:

$$max \sum_{i=1}^{n} (e_i * x_i - k_{v_i})$$

$$s.t.:$$

$$\sum_{i=1}^{n} a_{ij} * x_i \leq b_j;\ \forall j \in \{1, \dots, m\}$$

$$x_i \leq h_i;\ \forall i \in \{1, \dots, n\}$$

$$x_i \geq 0;\ \forall i \in \{1, \dots, n\}$$

$$a_{ij} \dots Ressourcenverbrauch\ des\ Produktes\ i\ an\ der\ Ressource\ j$$

$$b_j \dots Ressourcenverbrauchsobergrenze\ f\ddot{u}r\ Ressource\ j$$

$$h_i \dots Absatzobergrenze\ f\ddot{u}r\ Produkt\ i$$

Formel 9: Mathematische Formulierung des Produktionsprogrammplanungs-Optimierungsproblems nach [BSW12]

Probleme niedriger Dimensionalität lassen sich auch mit der grafischen Methode lösen. Abbildung 8 zeigt hierfür ein Beispiel für 2 Produkte mit den Produktmengen x_1 und x_2, den Absatzbeschränkungen $X1$ und $X2$ und den Kapazitätsbeschränkungen A, B und C, sowie der Orthogonalen zur Gewinnfunktion G^*.

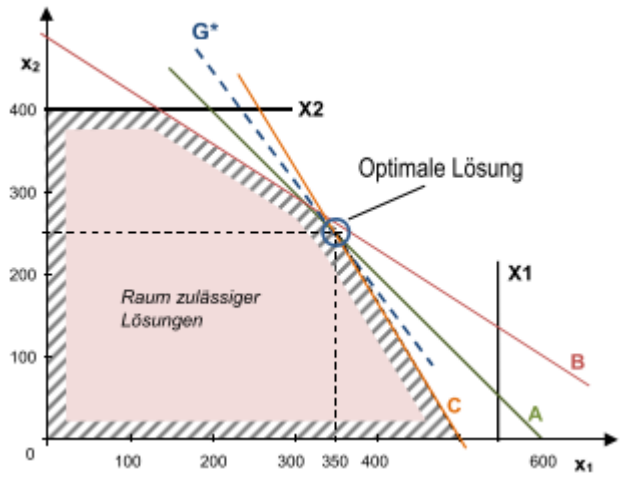


Abbildung 8: Grafische Lösung einer Produktionsprogrammplanungs-Optimierung [BSW12]

Anzumerken ist weiterhin, dass in der betriebswirtschaftlichen Praxis die Produktionsmengen x_i oft nicht als Ganzzahlen optimiert werden, sondern als Fließkommazahlen. Dies hängt damit zusammen, dass das Problem somit in der Komplexitätsklasse P liegt und effizient z.B. mit dem Simplex-Algorithmus lösbar ist. Letztendlich bleibt mit dieser Methode bei ungeraden Produktionsmengen dann allerdings auch keine andere Möglichkeit als die Produktionsmengen abzurunden, auch wenn dies nicht zwingend zum optimalen Ergebnis des ganzzahligen Problems führt. Des Weiteren besteht durch die Benutzung des Excel-Solvers ein im Allgemeinen gut verfügbares Tool zur Lösung.

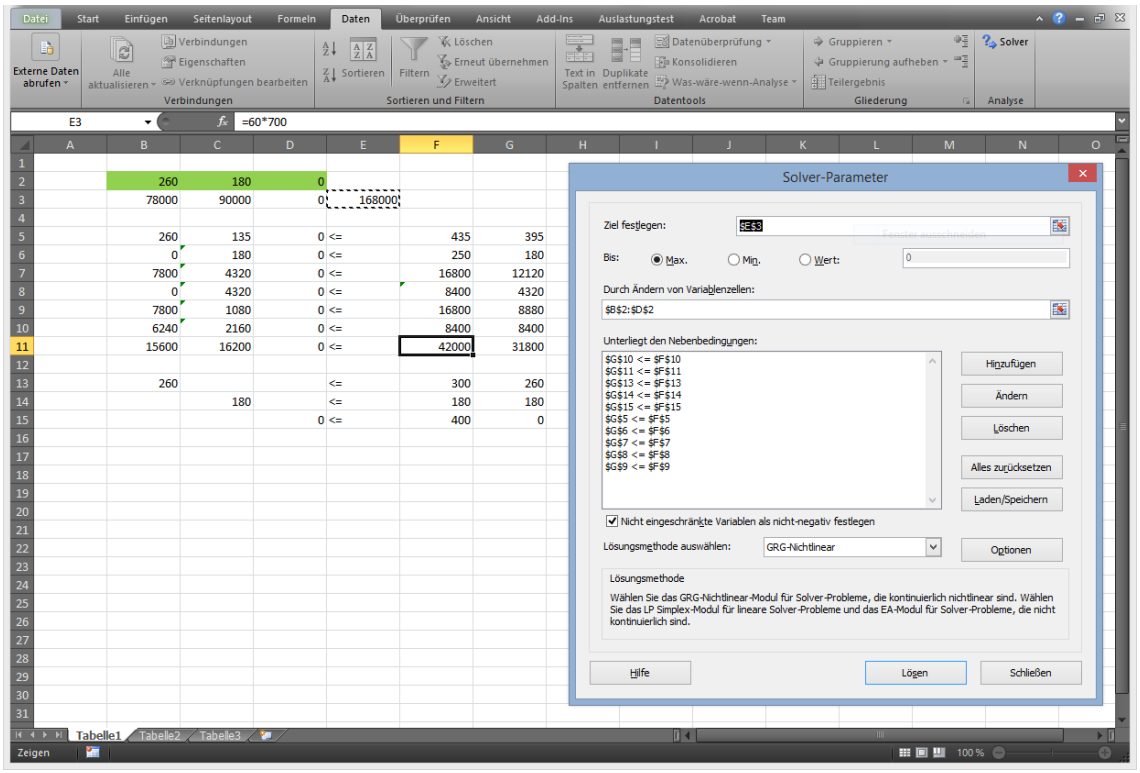


Abbildung 9: Screenshot des Excel-Solver Add-Ins bei der Lösung eines Optimierungsproblemen der Produktionsprogrammplanung

Abbildung 9 zeigt das Excel-Solver Add-In zur Lösung von Optimierungsproblemen am Beispiel eines im Screenshot bereits gelösten Produktionsprogrammplanungsproblems mit einem Deckungsbeitrag von 168.000 und den Produktionsmengen 260,180 und 0.

Zusammenfassend lässt sich feststellen, dass das Problem sehr gut geeignet ist in einem universellen Tool umgesetzt zu werden, das die Eingabe der Problemparameter erleichtert. Das Problem ist sowohl praxisrelevant als auch von der Komplexität her in einer ersten Version eines solchen Tools umsetzbar.

4.2 Unterstützung der Problemformulierung

Im folgenden Abschnitt wird auf Methoden eingegangen, die bei der Problemformulierung rechnergestützter Entwurfsoptimierungsproblem unterstützen. Besonderes Augenmerk wurde hierbei auf die Anwendung von Ontologien zur Unterstützung in der Problemformulierung gelegt.

Im Rahmen des immer fortschrittlicheren industriellen Entwicklungsprozesses, wurde die Designoptimierung immer bedeutender und ist heute ein essentieller Bestandteil im Entwicklungsprozess. Die Anwendungsmöglichkeiten von Optimierung in der Entwicklung werden immer vielfältiger und tiefgreifender, dies wird auch bereits durch verschiedenste Softwaretools unterstützt. [WKG07]

Ein Faktor der zunehmend an Bedeutung gewinnt ist die Tatsache, dass mit der Bedeutung und der Komplexität der Optimierungsprobleme auch stets das mit dem Optimierungsproblem assoziierte Wissen anwächst. So hängt konsequenter Weise immenses Wissen am Prozess der Optimierung, das zur Wahl der verwendeten Modelle und Methoden führte. Auch Abstraktionen der Problemstellung, wie sie in nahezu jedem Problem aus der Praxis getroffen werden müssen, können nur durch dieses Metawissen erklärt werden. [WKG07]

Bisher befindet sich dieses Detailwissen meist nur in den Köpfen der Entwickler und dazu oft nicht gesammelt, sondern fragmentiert nach Arbeitsgebieten. Das Ergebnis ist, dass es sehr aufwändig ist, einen Gesamtüberblick über die getroffenen Annahmen zu bekommen und sich die Situation bei Mitarbeiterabwanderungen noch weiter dramatisiert. Know-how kann dann verloren sein und das Verständnis über die eigene Entwicklung fehlt damit. Dies macht Änderungen im Modell schnell unmöglich und führt bei der Verwendung von unterschiedlichen Annahmen leicht zu falschen Ergebnissen. Doch selbst wenn das Optimierungsproblem nicht editiert werden soll, ist eine Interpretation der Ergebnisse ohne diese Hintergrundinformationen schwierig und ein Schluss auf ein fehlerhaftes Modell aufgrund von falschen Ergebnissen nahezu unmöglich. [WKG07]

In [WKG07] wird ein Ansatz verfolgt, indem es darum geht mittels Metadaten den Entwicklungsprozess speziell in der Designoptimierungsphase zu unterstützen. Das hierbei entwickelte prototypische Tool nennt sich ONTOP (engl.: ontology for optimization). ONTOP beinhaltet eine Wissensdatenbank mit standardisierten Optimierungsterminologien. Entwickelt wurde die Wissensdatenbank mit dem grafischen Tool

Protégé (siehe 2.4.12). Wichtigster Teil von ONTOP sind die speicherbaren Meta-Informationen. Mit diesen Meta-Informationen lassen sich viele Details des erstellten Modells erklären, so kann z.B. eine Meta-Information erklären warum eine bestimmte Restriktion existiert. ONTOP assistiert hierbei den Ingenieur Schritt-für-Schritt durch den Entwurfsoptimierungsprozess und frägt dabei Metainformationen vom Nutzer ab. [WKG07]

ONTOP bietet die Möglichkeit abstraktes, designbezogenes Wissen semantisch zu speichern. Das einzig ähnliche Tool, das laut [WKG07] noch auf dem Markt ist, ist der Optimization Modeling Assistant[14] (OMA), welcher jedoch mehr auf Ablauforganisation, Logistik und Finanzmanagement abzielt. Industrielle Designoptimierung benötigt jedoch eine weit abstraktere Wissensdatenbank als ein Programm mit einem stark begrenzten Anwendungsgebiet. [WKG07]

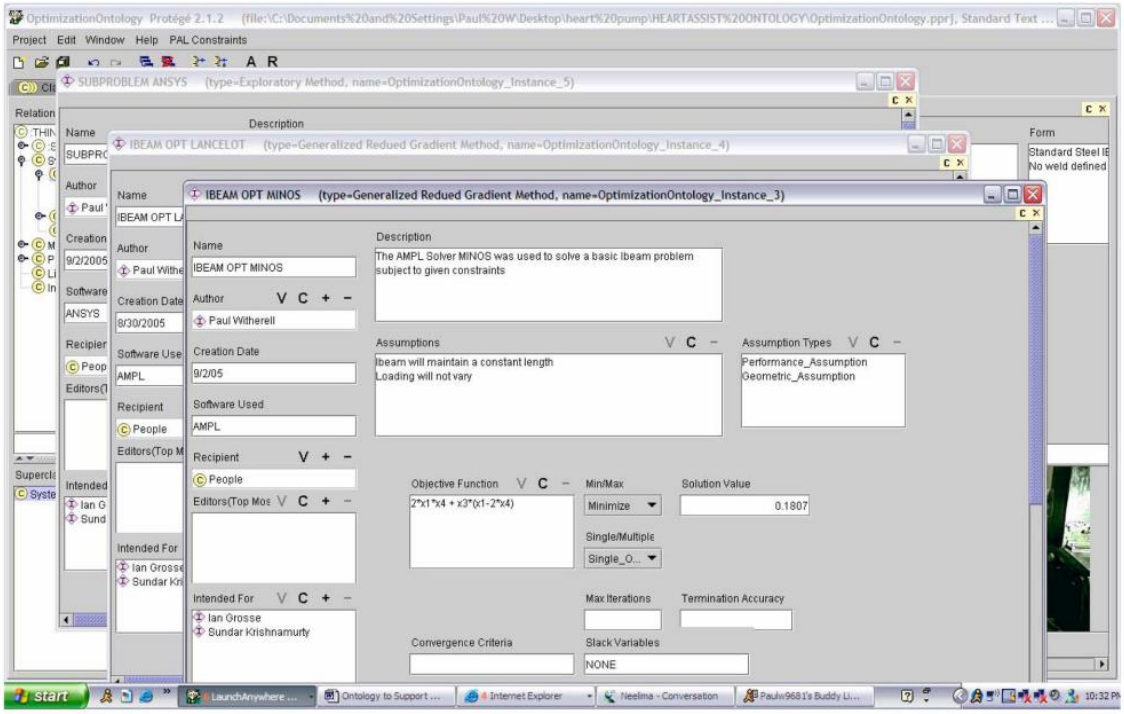


Abbildung 10: Screenshot des ONTOP-Tools zum Anhängen von Metadaten zu Optimierungsproblemen [WKG07]

In [EKGW11] wird ebenfalls ein Design-Framework vorgestellt, mit dem es möglich ist Designinformationen über den gesamten Entwicklungsprozess zu integrieren. Ziel ist es den Produktinnovationsprozess zu unterstützen. Mit diesem Framework kann die Effizienz im Industriedesign erheblich verbessert werden. Auch hier wird festgestellt, dass es

[14] http://sbir.gsfc.nasa.gov/SBIR/successes/ss/10-016text.html

sehr wenig Softwareunterstützung gibt, die die Wissensintegration im Designprozess unterstützt.

Der Artikel [RGKW09] befasst sich mit einem Framework, das die gemeinschaftliche Entscheidungsfindung im Industriedesign unterstützt. Zur Anwendung kommen auch hier wieder Ontologien um die Informationen zu strukturieren.

Komplexe Produkte und die Aufteilung von Entwicklungsaufgaben in verschiedene Domänen erfordern eine Designumgebung mit der eine erfolgreiche Zusammenarbeit über Domänengrenzen hinweg möglich ist. Ohne eine solche softwareseitige Unterstützung wird der Entwicklungsprozess komplexer Produkte sowohl sehr teuer als auch zu langwierig. [RGKW09]

Zu langwierige Entwicklungsprozesse, selbst wenn sie nicht zu hohen Kosten führen (z.B. bei geringer Parallelisierung), sind für Unternehmen mit Nachteilen im Wettbewerb verbunden. Eine schnelle Produkteinführungszeit (engl. Time-to-Market), bedeutet für ein Unternehmen immer, dass der Entwicklungsprozess verkürzt wird und das entwickelte Produkt früher seine Entwicklungskosten einspielen kann, dies schafft Liquidität. Hierzu kommen wettbewerbstechnische Vorteile, aufgrund der Tatsache, dass das Produkt einen Zeitvorsprung gegenüber Konkurrenzprodukten oder Substituten der Konkurrenz hat.

Aus diesen Gründen gibt es in vielen Sparten zunehmend keine andere Möglichkeit mehr als gemeinschaftliche Entwicklungsprozesse zu verwenden. Erschwert wird dies noch durch immer komplexer werdende Produkte, wobei auch diesem Problem nur durch eine weitere Parallelisierung von Entwicklungsprozessen entgegengetreten werden kann. Um den Prozess möglichst zeit- und ressourcenschonend zu gestalten muss ein entsprechendes Unterstützungstool die Kommunikation von Designwissen verbessern. [RGKW09]

Das in [RGKW09] vorgestellte Konzept strukturiert das Wissen über Designentscheidungen um es dann domänenübergreifend bereitzustellen. Somit wird ein Framework zur Verfügung gestellt mit dem Designentscheidungen und assoziierte Informationen domänenübergreifend kommuniziert werden können. Das Framework ermöglicht so eine effiziente Zusammenarbeit in der Entwicklung.

4.3 Methoden zur Beschleunigung der Berechnungszeit

Wenn ein Optimierungsproblem formuliert ist, gilt es die optimale Lösung zu finden, dies geschieht mithilfe von Optimierungsalgorithmen. Bei Problemen mit kleiner Anzahl an Designvariablen in möglichst engen Grenzen brauchen wir uns heutzutage im Allgemeinen keine großen Sorgen über die Berechnungszeit machen.

Es gibt jedoch auch eine Vielzahl von Problemstellungen, bei denen Berechnungszeiten von Tagen und Wochen üblich sind. Ein Beispiel dafür sind Crash-Simulationen in der Automobilentwicklung. So ergab ein beispielhaftes Rohbaumodell eines Autos des Automotive Simulation Center Stuttgart (ascs) eine Berechnungszeit von 4,5 bis 12 Stunden für einen Simulationsschritt. Die Berechnung basiert auf einem System mit 256 Rechenkernen. [Aut12]

Es lässt sich leicht hochrechnen, dass die Berechnungszeit bei hunderten oder tausenden Iterationen deutlich zu hoch ist, also muss eine Lösung gefunden werden. Hierfür gibt es zwei grundlegende Möglichkeiten: Entweder der Suchraum der Designvariablen wird soweit reduziert damit bereits mit wenigen Iterationen Lösungen nahe des Optimums gefunden werden können. Die Alternative ist es das Problem an der Wurzel zu packen und die Berechnungszeit, ohne den Suchraum zu verkleinern, zu reduzieren.

Zusammenfassend lässt sich feststellen: Wenn der einzelne Iterationsschritt also schon sehr lange dauert und dann diese Berechnung in jeder Iteration des Entwurfsoptimierungsprozesses ausgeführt werden muss, dauert die Optimierung bereits deutlich zu lange um die Time-To-Market gering zu halten. Aus diesem Grund wird in [WS07] versucht mithilfe von Metamodellen die Berechnungszeit zu reduzieren. Basis des Ansatzes ist, dass Metamodelle existieren von denen es abgewandelte Produktvarianten gibt. Nun muss der langwierige Optimierungsprozess nur einmal für das Metamodell angeworfen werden, die Ergebnisse dieser Optimierung lassen sich dann in der Folge bei der Optimierung der Produktvarianten wiederverwenden. Damit wird eine drastische Reduzierung der Optimierungszeiten für die Produktvarianten erzielt.

Eine andere Möglichkeit die Berechnungszeit zu reduzieren, wird in [TER09] beschrieben. Hier wird die Struktur des Gesamtproblems näher betrachtet und versucht daraus miteinander verkettete Teilprobleme zu erzeugen.

Es gibt im Wesentlichen drei Möglichkeiten wie Sub-Probleme eines Optimierungsproblems miteinander verkettet sein können:

- Parallele Anordnung der Sub-Probleme: Es muss zwischen der Lösung der Sub-Probleme und des Master-Problems hin und her iteriert werden. Das bedeutet die Ergebnisse der Subprobleme müssen erst am Ende der Iteration ausgetauscht werden. Es bietet sich deshalb an, die Sub-Probleme parallel zu lösen, wenn die Optimierung auf einem parallelen Rechensystem ausgeführt wird, da keine Abhängigkeiten zwischen den Subproblemen bestehen. [TER09]

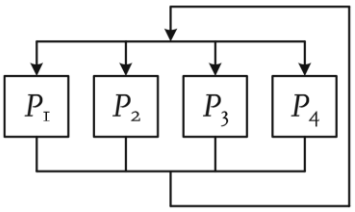


Abbildung 11: Parallele Struktur von Sub-Optimierungsproblemen eines Optimierungsproblems [TER09]

- Sequentielle Anordnung der Sub-Probleme: Die Lösung jedes Sub-Problems wird so früh benötigt wie diese vorhanden ist und fließt in die Lösung der anderen Sub-Probleme ein. Eine neue Iteration muss somit wieder mit der Lösung des ersten Sub-Problems beginnen. [TER09]

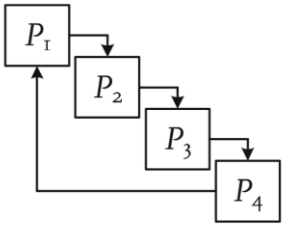


Abbildung 12: Sequentielle Struktur von Sub-Optimierungsproblemen eines Optimierungsproblems [TER09]

- Hybride Anordnung der Sub-Probleme: Ist eine Kombination von parallelen und sequentiellen Anordnungen der Sub-Probleme. [TER09]

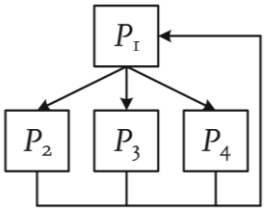


Abbildung 13: Hybride Struktur von Sub-Optimierungsproblemen eines Optimierungsproblems [TER09]

5 Einordnung der Aufgabenstellung und Vorgehen

In diesem Kapitel wird die Aufgabenstellung eingeordnet und das Vorgehen für Entwurf und Implementierung eines universellen Tools zur Unterstützung in der Problemformulierung erläutert.

5.1 Einordnung der Aufgabenstellung

Das Thema dieser Arbeit „Rechnerunterstützte Problemformulierung in der Entwurfsoptimierung: Ein internetgestützter Ansatz“ ist bewusst sehr offen ausgelegt. Gerade auch deshalb da erst ein Überblick darüber verschafft werden muss, zu welchen Themen es bereits Lösungen gibt und zu welchen nicht und was machbar ist und den Rahmen dieser Arbeit nicht sprengt.

Da, wie in Abschnitt 4.2 beschrieben, die Unterstützung des Problemformulierungsprozesses durch bessere Dokumentation desselben bereits in mehreren Arbeiten erfolgreich angewandt wurde, wird der Fokus dieser Arbeit mehr auf die Problemformulierung an sich gelegt.

Die Arbeiten [WKG07], [EKGW11] und [RGKW09] nehmen allesamt an, dass der potentielle Nutzer eines Problemformulierungstools die mathematischen Fähigkeiten besitzt um ein Problem formal korrekt zu definieren oder zu editieren. In dieser Arbeit gilt diese Annahme nicht. Es soll vielmehr ermöglicht werden, dass mehrere Benutzer unterschiedlicher Fähigkeiten gemeinsam zur Problemformulierung und Optimierung beitragen können. So kann sowohl vom Detailwissen des Informatikers in der Auswahl und Konfiguration der Algorithmen, des Produktmanagers bzgl. des Wissens zur Definition des Problemkonstrukts, als auch des Entwicklungsingenieurs um das Wissen zur Belegung der Umweltvariablen des Problems profitiert werden.

Ziel dieser Arbeit ist es also den Prozess der Problemformulierung in der Entwurfsoptimierung zu strukturieren, dass jede Nutzergruppe nur die Komplexität trifft, die für sie zu bewältigen ist. Aufgaben die nicht in den eigenen Wissensbereich fallen, sollten auch nicht selbst erledigt werden müssen, sondern von einem geeigneten Nutzer zeitlich und räumlich unabhängig erledigt werden können. So sollte es möglich sein Komplexität aus dem Entwicklungsprozess zu nehmen, indem Arbeit und Wissen anderer Nutzer

Schritt für Schritt kanalisiert werden. Ergebnis soll das fertig formulierte Problem als Eingabe in einen bereits konfigurierten Optimierungsalgorithmus, oder auch direkt das dem Problem optimale Ergebnis sein.

5.2 Vorgehen

Wie in der Aufgabenstellung (Kapitel 3) bereits definiert, soll als Lösung dieser Arbeit eine prototypische Implementierung eines Tools dienen, das den Problemformulierungsprozess dahingehend unterstützt, dass die Komplexität dessen reduziert wird.

Zunächst einmal wurde die Aufgabenstellung ein wenig verallgemeinert: Anstatt nur eine spezifische Zielfunktion (engl. objective function) im Optimierungsproblem zuzulassen, wie in Formel 3, sollen beliebig viele Zielfunktionen zugelassen werden. Das veränderte Optimierungsproblem ist in Formel 10 Formal dargestellt.

$$\begin{array}{ll} x & \textit{design variables} \\ y & \textit{state variables} \\ \forall \boldsymbol{i} \in \{\mathbf{1}, \dots, \boldsymbol{n}\}: \min\limits_{\boldsymbol{x}} \boldsymbol{f_i}(\boldsymbol{x}, \boldsymbol{y}) & \boldsymbol{\textit{objective functions}} \\ \forall j \in \{1, \dots, m\}: g_j(x, y) \leq 0 & \textit{inequality constraints} \\ \forall k \in \{1, \dots, k\}: h_k(x, y) = 0 & \textit{equality constraints} \end{array}$$

Formel 10: Verallgemeinertes Problemformulierungsschema zur Eingabe eines Entwurfsoptimierungsproblems in einen entsprechenden Lösungsalgorithmus

Dies hat zur Folge, dass ein solches Problem nicht mehr zwangsläufig eine einzelne optimale Lösung bezüglich der Optimierungsfunktion hat, sondern eine Menge von pareto-optimalen Lösungen, bei der jede in ihrer Kombination der verschiedenen Zielfunktionen pareto-optimal ist, also nicht von einer anderen Lösung dominiert wird.

Des Weiteren sei darauf hingewiesen, dass ein Wegfallen der Gleichheits-Restriktionen (engl. equality constraints) nichts an der Mächtigkeit des Problemformulierungsschemas ändern würde, da sich jede Gleichheits-Restriktion durch zwei Ungleichheits-Restriktionen abbilden lässt, dies ist Formal in Formel 11 abgebildet.

$$\begin{gathered} h_k(x, y) = 0 \\ \Leftrightarrow h_k(x, y) \leq 0 \wedge h_k(x, y) \geq 0 \\ \Leftrightarrow h_k(x, y) \leq 0 \wedge (-h_k(x, y)) \leq 0 \end{gathered}$$

Formel 11: Umformungsschema für Gleichheitsrestriktionen in zwei Ungleichheitsrestriktionen

Eine weitere Ausweitung der Aufgabenstellung betrifft den Umgang mit dem Optimierungsalgorithmus. Hier soll nicht mit der Übergabe an den Optimierungsalgorithmus

gestoppt werden, da die Interpretation der Ergebnisse oft ähnlicher Unterstützung bedarf wie die Problemformulierung. Deshalb soll auch die Präsentation der Ergebnisse Teil dieser Arbeit sein und in der Entwicklung des Prototyps berücksichtigt werden.

Die Lösung der Aufgabenstellung dieser Arbeit soll zunächst in einem Konzept für ein Tool liegen, das den Problemformulierungsprozess unterstützt. Danach soll das Konzept evaluiert werden, dies geschieht durch eine prototypische Implementierung und einer anschließenden Evaluierung des Prototyps bzgl. von praxisnahen Anwendungsfällen (engl. use cases).

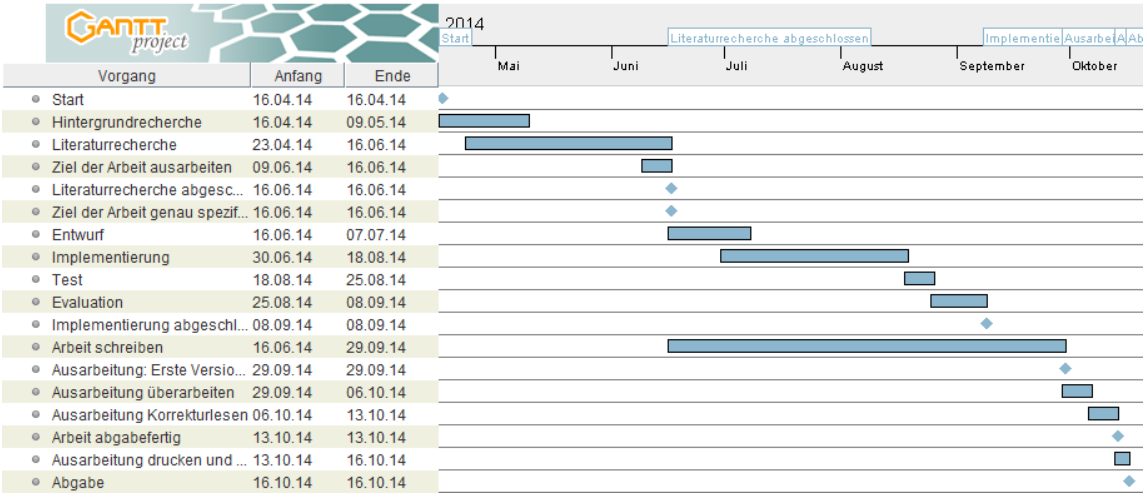
In dieser Arbeit wird ein lineares Vorgehensmodell gewählt, wobei Projektphasen teilweise mit anderen Projektphasen zeitlich überlappen. Der Beginn (16. April) und Ende (16. Oktober) stehen von Anfang an fest. Alle Projektphasen müssen in diesem Zeitraum eingeplant werden.

Die einzuplanenden Projektphasen sind die folgenden:

- Hintergrundrecherche
- Literaturrecherche
- Entwurf
- Implementierung
- Test
- Evaluierung
- Ausarbeitung schreiben
- Ausarbeitung drucken

In Abbildung 14 wurden die Projektphasen in einem Gantt-Diagramm dargestellt. Zusätzlich sind in den Ablauf Meilensteine eingefügt. Das Diagramm wurde mit dem Tool Gantt-Project[15] erstellt.

[15] http://www.ganttproject.biz/



Vorgang	Anfang	Ende
Start	16.04.14	16.04.14
Hintergrundrecherche	16.04.14	09.05.14
Literaturrecherche	23.04.14	16.06.14
Ziel der Arbeit ausarbeiten	09.06.14	16.06.14
Literaturrecherche abgesc...	16.06.14	16.06.14
Ziel der Arbeit genau spezif...	16.06.14	16.06.14
Entwurf	16.06.14	07.07.14
Implementierung	30.06.14	18.08.14
Test	18.08.14	25.08.14
Evaluation	25.08.14	08.09.14
Implementierung abgeschl...	08.09.14	08.09.14
Arbeit schreiben	16.06.14	29.09.14
Ausarbeitung: Erste Versio...	29.09.14	29.09.14
Ausarbeitung überarbeiten	29.09.14	06.10.14
Ausarbeitung Korrekturlesen	06.10.14	13.10.14
Arbeit abgabefertig	13.10.14	13.10.14
Ausarbeitung drucken und ...	13.10.14	16.10.14
Abgabe	16.10.14	16.10.14

Abbildung 14: Gantt-Diagramm dieser Arbeit

6 Konzept

In diesem Kapitel wird das technologieunabhängige Konzept der Software zur Unterstützung in der Problemformulierung vorgestellt. In erster Linie geht es hierbei um die Beschreibung der Softwarearchitektur, dies beinhaltet das Zusammenspiel der notwendigen Komponenten.

6.1 Prozess der Problemumformulierung

In diesem Abschnitt wird der Prozess der Problemumformulierung betrachtet, hierbei wird der Fokus ausschließlich auf das Problem und dessen schrittweiser Umformulierung und Umwandlung bis hin zu den Optimierungsergebnissen betrachtet.

Aufgabe ist das abstrakte Problem, evtl. bereits in Fließtextform formuliert, Schritt für Schritt so weit umzuformulieren, dass daraus eine Konfiguration für einen Optimierungsalgorithmus entsteht. Im Anschluss daran den Optimierungsalgorithmus zu starten und nach Beendigung der Berechnungen die Ergebnisse zu präsentieren.

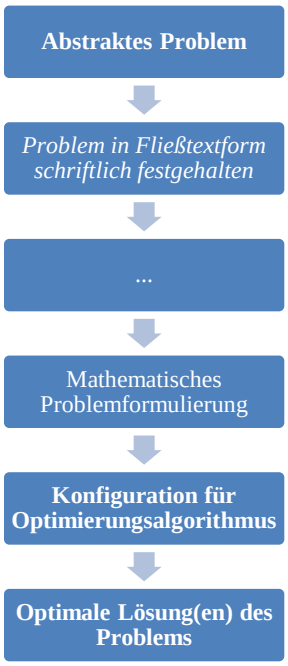


Abbildung 15: Prozess der Problemumformulierung von einem abstrakten Optimierungsproblem zu dessen optimaler Lösung

6.2 Systemkonzept

Im Folgenden wird das grundlegende Konzept des zu implementierenden Tools beschrieben, das heißt es wird ein grundlegendes Verständnis davon geschaffen, mit welchen Ansätzen die vom zu implementierenden Tool geforderte Funktionalität geschaffen wird.

Die wesentlichen Komponenten des Softwareprojektes sind in Abbildung 16 dargestellt. Das Tool ist in der Form eines Expertensystems (siehe Abschnitt 2.4.4) aufgebaut. Die Wissensbasis bildet hierbei eine Ontologie auf die mithilfe eines Reasoners zugegriffen wird. Die Wissensbasis beruht wiederum auf einem hinterlegten Modul-Repertoire. Die Programmlogik greift schließlich auf die Wissensbasis mithilfe des Reasoners zu um die Benutzeroberfläche zu kreieren. Maßgebend für das Erstellen der Benutzeroberfläche ist das hinterlegte Modul-Repertoire.

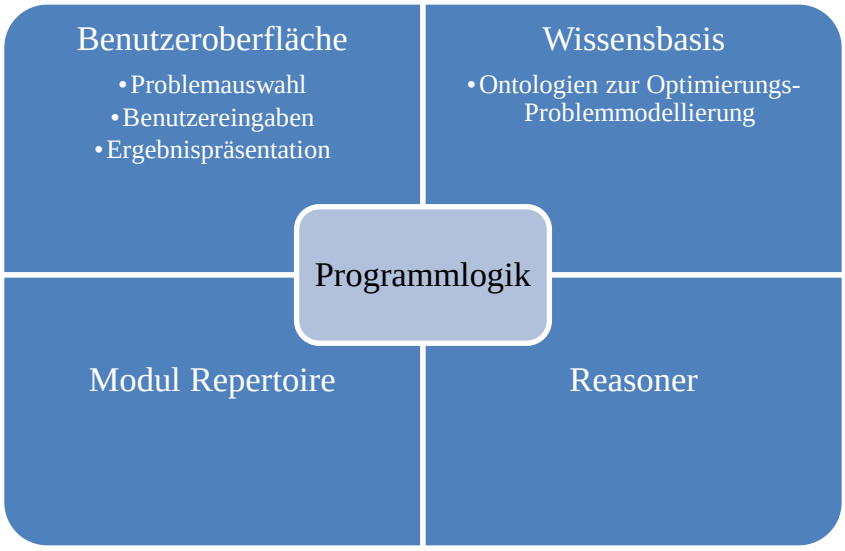


Abbildung 16: Die wesentlichen Komponenten der zu entwickelnden Software

6.3 Benutzeroberfläche

In diesem Abschnitt wird das Konzept aus Bedienersicht vorgestellt. Die Benutzeroberfläche spielt bei einem Tool, dessen Aufgabe es ist einen Prozess zu vereinfachen, eine zentrale Rolle. Die Bedienung erfordert genau wie die Problemformulierung ohne Softwareunterstützung kognitive Ressourcen. Ist die Bedienung zu kompliziert kann es zu dem Fall kommen, dass das Tool keinen Mehrwert mehr darstellt und dessen Benutzung somit nicht sinnvoll ist.

Aufgrund der im Systemkonzept (Abschnitt 6.2) getroffenen Entscheidung ein Expertensystem zu erstellen, das auf einen Datenbestand parametrisierbare Probleme (bestehend aus Problem-Modulen) zugreift, kann das Expertensystem diesen Datenbestand voraussetzen.

Das User-Interface soll die folgenden Haupt-Bedienschritte beinhalten:

- Problem Auswahl: Aus einer Liste vorkonfigurierter parametrisierbarer Probleme wird das passende Problem ausgewählt.
- Benutzereingaben: Für alle Module die eine Benutzereingabe erfordern wird ein Benutzerinterface angezeigt, das vom Modul frei bestimmt werden kann.
- Präsentation der Ergebnisse der Optimierung: Nachdem der konfigurierte Optimierungsalgorithmus gelaufen ist, werden die Ergebnisse dem Benutzer präsentiert.

Aufgrund der Forderung eines internetgestützten Ansatzes (siehe Kapitel 3), wird die komplette Bedienung browsergestützt und damit plattform- und geräteunabhängig ablaufen. Allerdings wurden die Bedienelemente größenmäßig auf Geräte mit großen Bildschirmen (Desktop PC, Laptop, Tablet, …) angepasst.

Das Bedienkonzept wurde in Abbildung 17 anhand von Wireframes grafisch dargestellt.

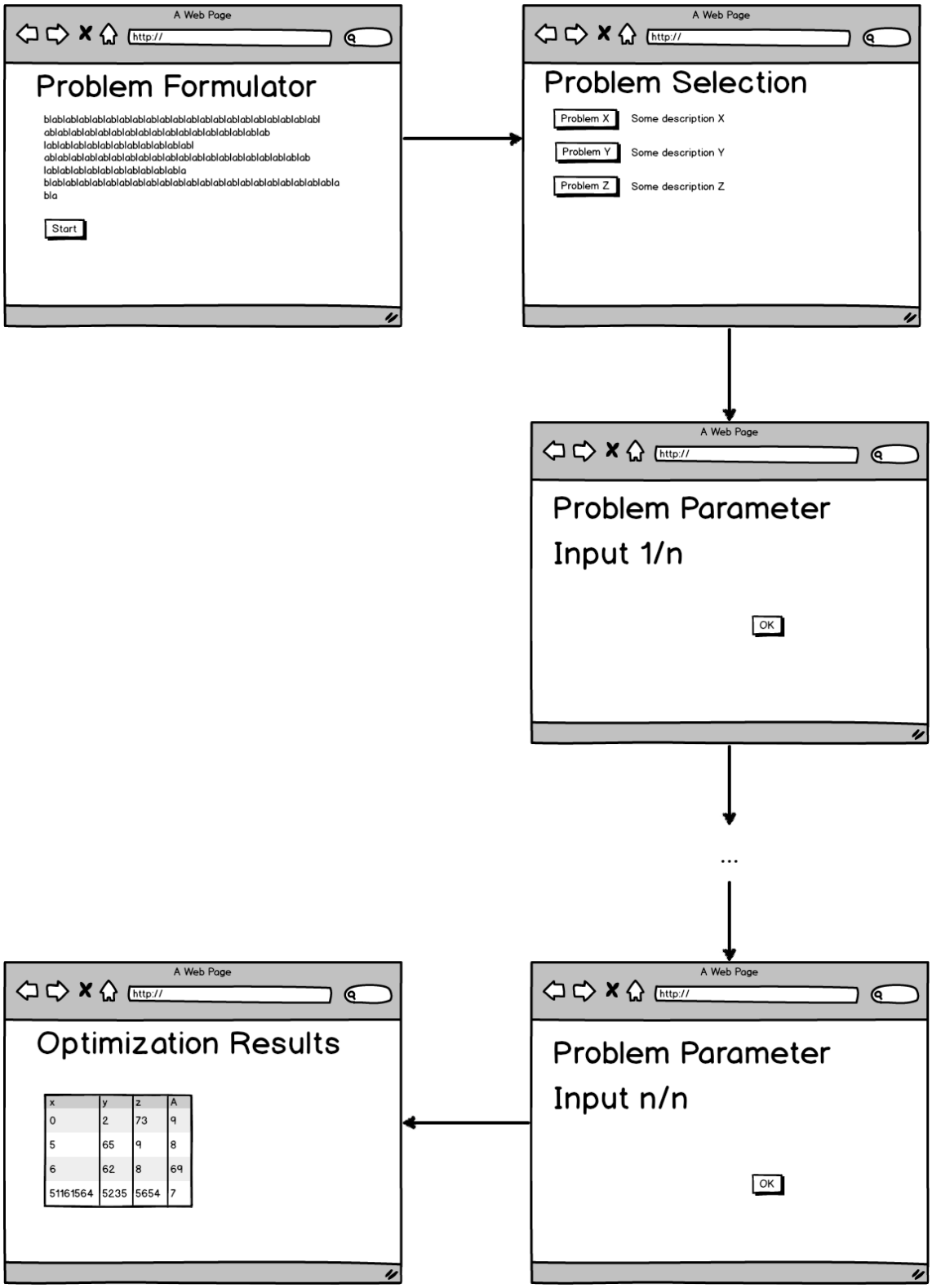


Abbildung 17: Wireframe des zu erstellenden Tools erstellt mit *Balsamiq Mockups*[16] *for Google Drive*

[16] https://balsamiq.com/products/mockups/

6.4 Verteiltes System

Als System-Architektur wurde eine klassische Zwei-Tier-Client-Server-Architektur (siehe Abschnitt 2.4.3) gewählt. Da die Anwendungslogik stark von der Wissensdatenbank abhängt und die Wissensdatenbank auf dem Server liegen soll, kommt das Modell der Remote-Präsentation oder, wenn Eingaben teilweise bereits auf dem Client validiert werden sollen, alternativ das Modell der Distributed Application in Frage.

Der Server liefert somit nur die Daten die zur Auswahl eines Problems und zu dessen Parametrisierung notwendig sind auf Anfrage des Clients an diesen zurück. Die Sammlung der Daten und die anschließende Optimierung der Lösung findet vollständig auf dem Server statt. Erst die Präsentation der Ergebnisse findet dann wieder auf dem Client statt.

6.5 Wissensmodellierung

In diesem Abschnitt wird der gewählte Ansatz zur Wissensmodellierung in Form von Modellierungsproblemen aufgezeigt. Hierfür wurde ein modularer Ansatz gewählt. Module sollen hierbei andere Module als Input-Module haben können. Das heißt ein Problem besteht aus Modulen, die zum Teil miteinander verkettet sind.

Hierbei muss es zum einen sogenannte Start-Module, die selbst keine Input-Module haben, und zum anderen sogenannte End-Module, die selbst von keinem anderen Modul Input-Modul sind. Module die keiner der beiden Kategorien zugeordnet werden können sind sogenannte Mid-Module. Diese können andere Module als Input haben und genauso selbst ein Input-Module eines anderen Moduls sein.

Wie die Problemmodellierung mittels solcher verketteter Module aussieht visualisiert Abbildung 18 mittels eines Graphs.

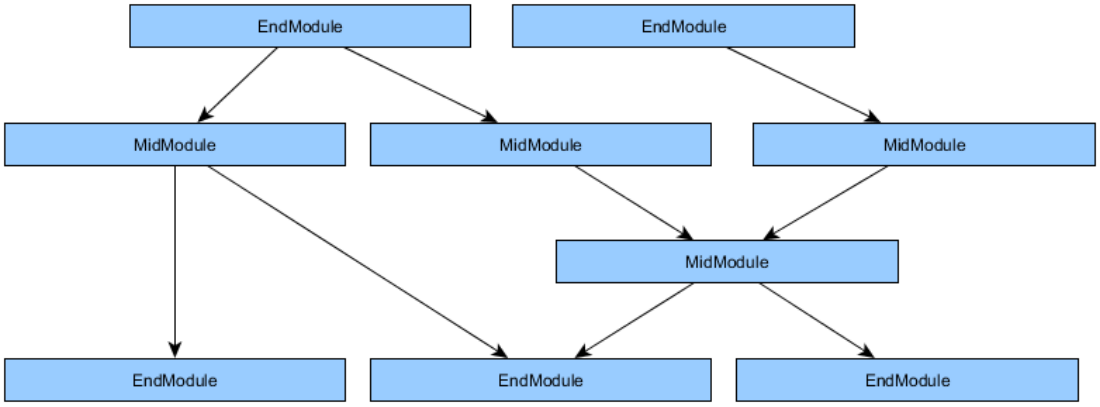


Abbildung 18: Angedeutete Problemmodellierung mittels verketteter Module visualisiert durch einen Graph

Hierarchie von Modulklassen:

- StartModule: Modul das keine InputModules haben kann. Es sind also all jene Module die im Modulbaum die Blätter darstellen.
    - DesignModule: DesignModules sind Module, dessen Wertebelegung offen bleibt bis das Problem an den Optimierungsalgorithmus übergeben wird. Dieser findet dann eine optimale Belegung für alle Module dieser Modulklasse.
    - FixedModule: Dies sind Module, dessen Wertebelegung im Vorhinein in der Wissensdatenbank festgelegt wird. Es sind also Konstanten, die in das Problem eingehen.
    - InputModule: Ist ein Modul dessen Wert nach der Problemauswahl abgefragt wird. Hierzu muss die implementierte Modulklasse separat eine grafische Oberfläche zur Eingabe bereitstellen.
- MidModule: Betrifft alle Module, die weder einem StartModule noch einem EndModule entsprechen. Es sind also Module gemeint, die im Modulbaum irgendwo in der Mitte stehen.
- EndModule: Modul, das selbst von keinem anderen Modul ein InputModule sein kann. Hiermit sind all jene Module gemeint, die die Wurzel eines Modulbaums darstellen.
    - OptimizationModule: OptimizationModules sind Module dessen Wert vom Optimierungsalgorithmus bestimmt (maximiert oder minimiert) wird. Der Wert bewertet letztendlich die Güte einer gültigen Lösung, jedoch ist ein guter Wert eines OptimizationModules nicht von Bedeutung,

wenn durch den Wert eines RestrictionModules bestimmt ist, dass die Lösung ungültig ist.

- RestrictionModule: RestrictionModules haben für den Optimierungsalgorithmus einen binären Charakter und geben entweder wahr oder falsch zurück. Um eine gültige Lösung zu erhalten müssen alle RestrictionModules wahr zurückgeben.

Wichtig ist es anzumerken, dass die angesprochenen Module stets Klassen darstellen, von denen dann im konkreten Problem Instanzen erstellt werden. Damit kann es sein, dass beispielsweise eine Modul-Klasse sich wiederum selbst als Input-Modul akzeptiert. Allerdings muss es sich hierbei um verschiedene Instanzen derselben Klasse handeln, ansonsten ist ein Zyklus im Problem-Graph und Zyklen dürfen nicht erlaubt sein, sonst kann der Graph später nicht vom Expertensystem aufgelöst werden.

Der Teil des Moduls, der sich in der Wissensdatenbank befindet, hat im Allgemeinen folgende Bestandteile:

- Input-Descriptions: Nur bei Input-Modules erlaubt. Beschreibung der durchzuführenden Eingabe.
- Input-Description-Type: Falls für ein Input-Module mehrere Input-Descriptions möglich sind, kann mit dem Input-Description-Type eine Unterscheidung zwischen den Input-Descriptions geschaffen werden.
- Input-Modules: Module die dieses Modul als Eingabe benötigt um die eigenen Werte zu berechnen.
- Module-Properties: Erlauben der Modul-Instanz weitere Informationen mitzugeben. Hierbei gibt es prinzipiell keine Beschränkungen, ein Modul kann beliebige Module-Properties zulassen, nur die Implemented-Module-Class muss diese Properties verstehen können, diese wird jedoch mit dem Modul zusammen individuell erstellt. Z.B. bei FixedModules wird hierbei der zu belegende Wert festgelegt.
- Associated-Implemented-ModuleClass: Voller qualifizierter Name der implementierten Klasse dieses Moduls. Dient zur eindeutigen Identifikation damit beim Auslesen der Wissensdatenbank die entsprechenden Klassen instanziiert werden können.

Eine grafische Übersicht über die Bestandteile eines Moduls ist in Abbildung 19 dargestellt.

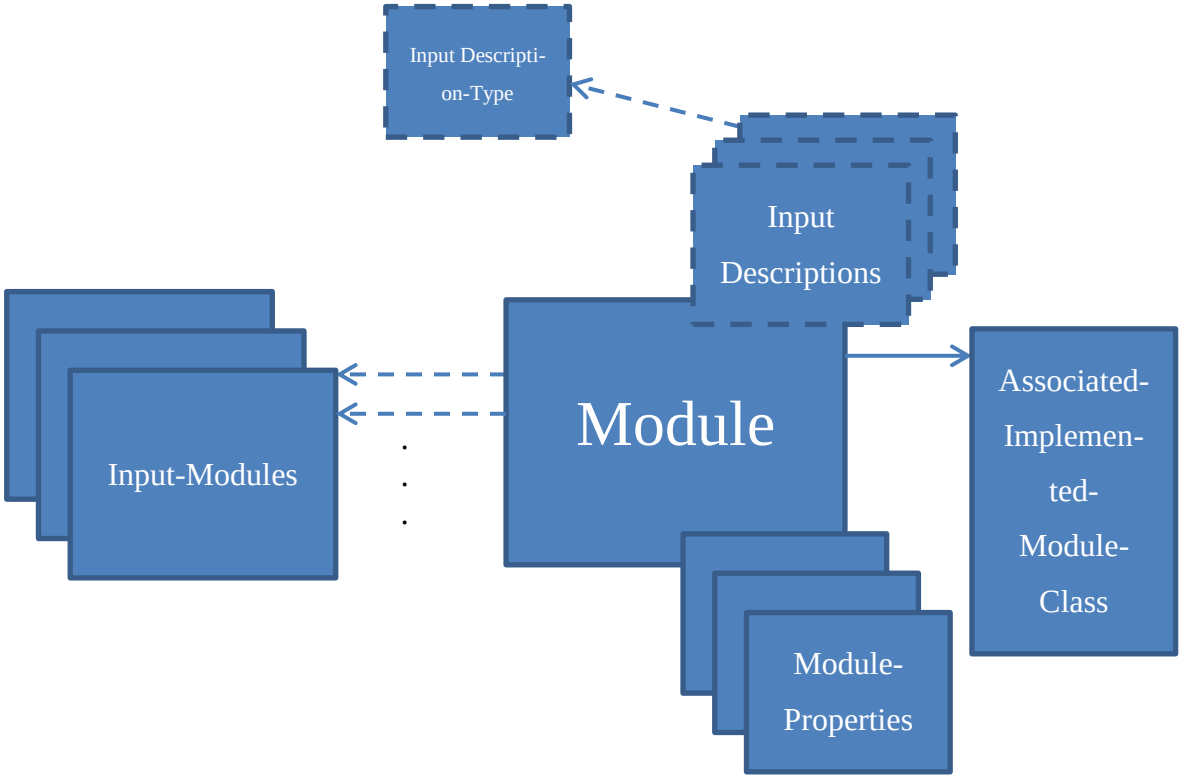


Abbildung 19: Bestandteile eines Moduls in der Wissensdatenbank

6.6 Softwarearchitektur

Dieser Abschnitt beschreibt die Softwarearchitektur des zu implementierenden Tools. Hierbei geht es nur um die technologieunabhängige Architektur. Die technologieabhängige Architektur wird im Anschluss in Kapitel 7 beschrieben.

Zunächst einmal muss das Tool, wie in Abschnitt 6.4 vorgegeben, in einer Client-Server-Architektur erstellt werden. Dafür müssen die Komponenten auch zwischen diesen Komponenten aufgeteilt werden. Gemäß dem gewählten Modell der Remote-Presentation (mit Ausnahme der Validierung von User-Eingaben direkt auf dem Client), befindet sich auf dem Server die gesamte Anwendungslogik und Datenhaltung. Zusätzlich sind Komponenten erforderlich, die sowohl dem Client als auch dem Server bekannt und von diesem genutzt werden können. Diese können dann zusätzlich zu den von der Programmiersprache bereitgestellten Datentypen in Nachrichten vom Client zum Server oder vom Server zum Client genutzt werden.

Der Server beinhaltet die folgenden Komponenten:

- Service: Anknüpfpunkt für den Client um Anfragen zu stellen. Dieser leitet die Verarbeitung der Anfrage auf dem Server ein und sendet eine Antwort zurück.
- Problem-Manager: Erstellt, verwaltet und bearbeitet Anfragen an das Problem und dessen Module.
- Problem: Verwaltet alle Modul-Instanzen des Problems und bietet Methoden zur Abfrage von bestimmten Modulen an.
- Problem-Calculator: Führt die Berechnungen durch um die Design-Variablen optimal zu belegen, das heißt die Optimierungsfunktionen zu maximieren oder zu minimieren und um alle Restriktionen zu erfüllen.
- Knowledge-Base-Accessor: Erlaubt den Zugriff auf die Wissensdatenbank.
- Knowledge-Base: Stellt die Wissensdatenbank an sich dar.
- Implemented-Modules: Für jedes Modul in der Wissensdatenbank muss es ein Gegenstück geben, das die erforderliche Funktionalität umsetzt.

Wenn an den Service eine Anfrage gestellt wird, greift dieser mit dem Knowledge-Base-Accessor auf die Knowledge-Base zu. Dabei wird im ersten Schritt eine Liste der Probleme abgefragt. Bei der zweiten Anfrage wird ein Problem ausgewählt, welches dann über den Problem-Manager erstellt wird, indem die zugehörigen Module instanziiert werden. Daraufhin werden durch eine Reihe von Anfragen Module abgefragt, die eine Eingabe erfordern und die entsprechenden Instanzen mit den vom Nutzer eingegebenen Werten befüllt. Sind alle Eingaben gemacht wird der Optimierungsprozess durch den Problem-Calculator durchgeführt.

Der Client beinhaltet die folgenden Komponenten:

- Service-Binding: Anknüpfpunkt für den Server damit dieser die Antwort auf Anfragen wieder an den Client zurück senden kann.
- Requestor: Generiert die Anfragen an den Server basierend auf den Benutzereingaben.
- Graphical User Interface (GUI)
    - Panels: Sind GUI-Panels für die verschiedenen Schritte im Eingabeprozess, wie in Abbildung 17 skizziert.
    - Implemented-Module-Widgets: Für jedes Input-Modul muss es ein passendes Module-Widget geben, das ein Panel darstellt mit dem der Benut-

zer die Eingaben machen kann und daraus die entsprechende Module-Data zu füllen.

Je nach Schritt im Eingabeprozess werden die erforderlichen Daten vom Requestor über das ServiceBinding vom Server abgefragt und im entsprechenden Panel angezeigt. Handelt es sich um die Benutzereingabe für ein Input-Module, wird in das Panel noch ein Implemented-Module-Widget eingebettet.

Die folgenden Komponenten sind sowohl dem Client als auch dem Server bekannt:

- Implemented-Module-Data: Für jedes Input-Modul muss es eine passende Module-Data geben. Damit können die Benutzereingaben an den Server gesendet werden und auch umgekehrt Informationen, die für die Eingabe des Benutzers benötigt werden, vom Server an den Client übermittelt werden.
- Additional-Datatypes: Sind weitere Datentypen, die nötig sind um Daten vom Server zum Client oder umgekehrt zu versenden.

Die folgende Abbildung stellt die wesentlichen Komponenten des zu entwickelnden Tools und deren Verteilung dar:

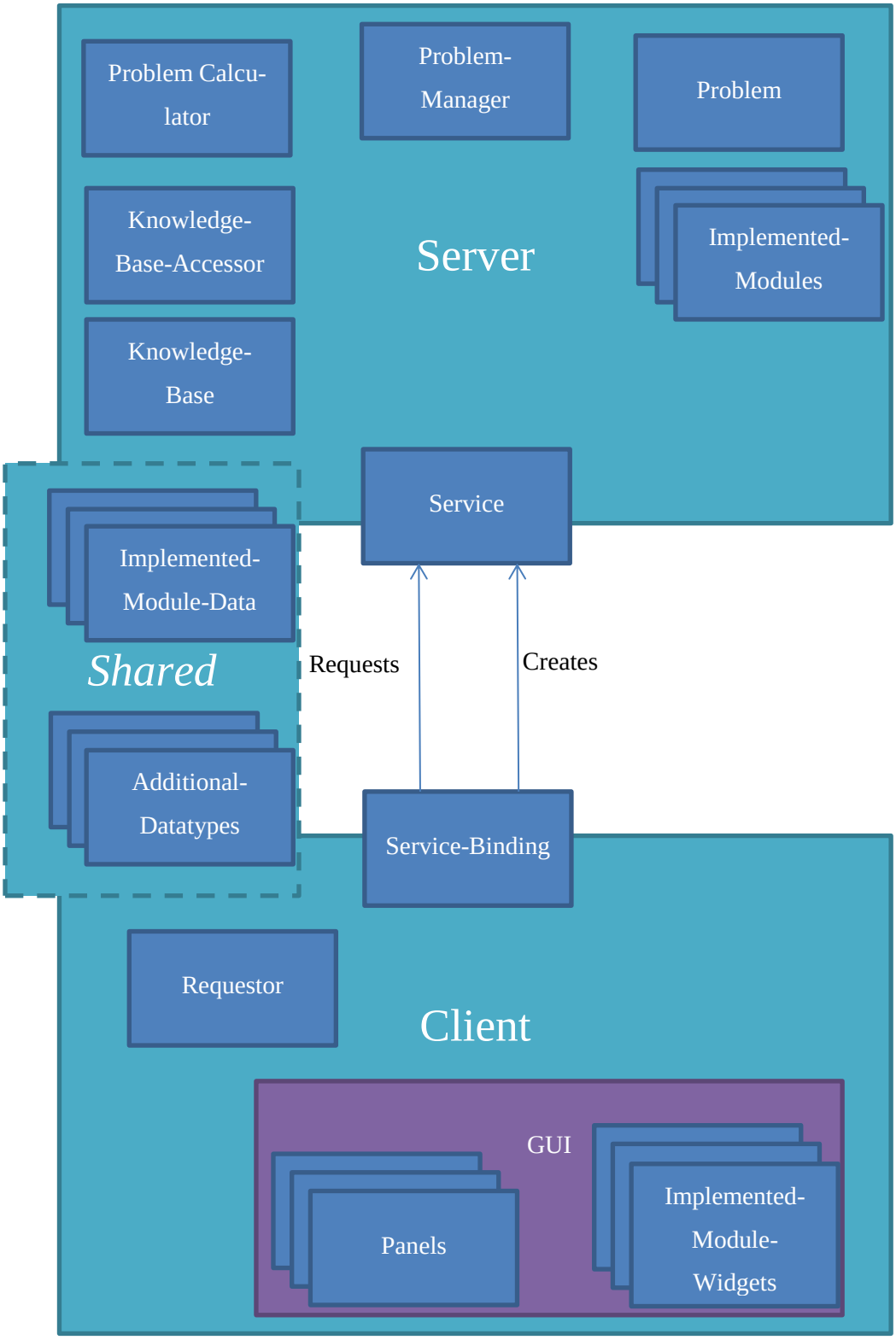


Abbildung 20: Hauptkomponenten des Problemformulierungs-Tools

Das Service Binding erstellt hierbei eine eigene Instanz auf dem Server, die dann fortan ausschließlich für diesen Client zuständig ist, dadurch erhält das System seine Mandantenfähigkeit (mehrere Benutzer gleichzeitig können mit dem System arbeiten).

7 Implementierung

Dieses Kapitel beschreibt die Implementierung des Problemformulierungstools unter Berücksichtigung der verwendeten Technologien. Die Struktur des Programms wurde bereits im Konzept beschrieben. Der Fokus dieses Kapitels liegt mehr im Detail und beschreibt die konkreten Implementierungsdetails.

7.1 Entwicklung der Wissensdatenbank

In diesem Abschnitt wird die Entwicklung der Wissensdatenbank beschrieben. Die Wissensdatenbank wurde mit OWL modelliert und im XML-Format in Dateien gespeichert. Zur Vereinfachung der Modellierung wurde das Werkzeug Protégé verwendet, damit ist gewährleistet, dass Optimierungsprobleme auch ohne die exakten syntaktischen Kenntnisse in XML und OWL modelliert werden können.

Die Wissensdatenbank besteht im Allgemeinen aus zwei Teilen zum einen aus der Schema-Ontologie. Das Schema gibt den Modellierungs-Rahmen für den zweiten Teil, die konkreten Problemmodellierungen vor.

Die Problemmodellierung geschieht in der Form, dass ein Baum bzw. Wald aus Modulinstanzen erstellt wird. Eine solche Problemmodellierung ist in Abbildung 21 dargestellt. Das Beispiel zeigt eine Optimierung, die die Summe zweier Variablen maximiert und gleichzeitig sicherstellt, dass das Produkt aus den zwei Variablen kleiner der vom Benutzer gemachten Eingabe ist.

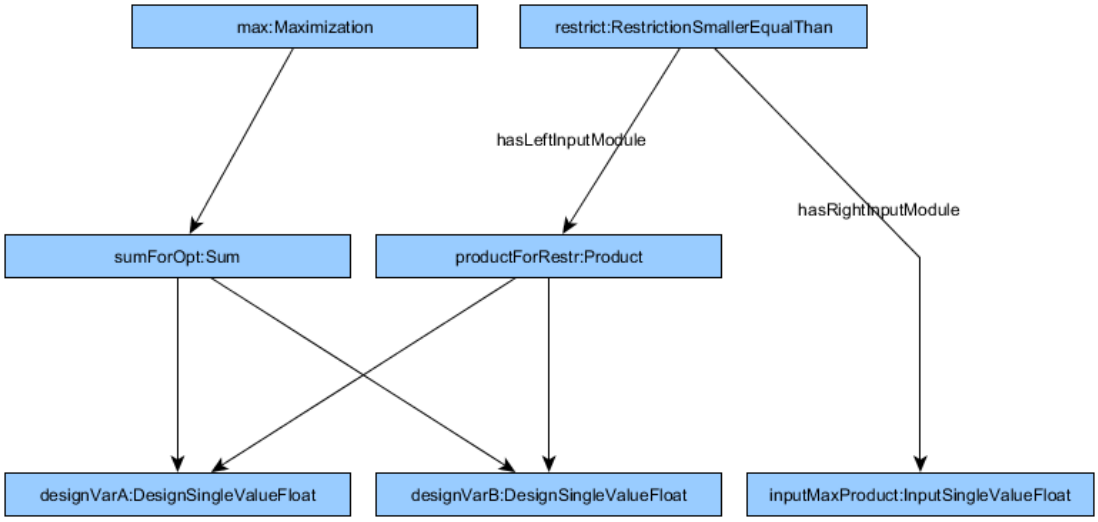


Abbildung 21: Beispiel Problemmodellierung durch Instanzen von Modulklassen als Graph visualisiert

7.1.1 Erstellung der Schema-Ontologie

Die Schema-Ontologie enthält in erster Linie Module aus denen ein Problem bestehen kann. Zum anderen wird spezifiziert wie diese Module zusammengesetzt und was an die einzelnen Module angehängt werden kann bzw. muss.

Ein weiterer wichtiger Teil ist eine Annotation an die Module, die es erlaubt die entsprechende Java-Klasse zu identifizieren. Erst diese enthält die gewünschte Funktionalität. Für den Optimierungsprozess müssen zusätzlich Informationen über Optimierungsalgorithmen und dessen Parametrisierung gegeben werden, damit diese Informationen dem Optimierungsproblem angehängt werden können und in diesem ohne Zutun des Benutzers der Optimierungsalgorithmus vollständig konfiguriert werden kann.

7.1.1.1 Klassenhierarchie

Im erstellten Prototyp dieser Arbeit werden bereits einige Basismodule umgesetzt, die es ermöglichen ein recht großes Spektrum an Optimierungsproblemen auf einfache Art und Weise umzusetzen. Dieser initiale Modulkatalog setzt in einem ähnlichen Ansatz wie in [GO94] eine mathematische Modellierung von Formeln in einer Art Baumstruktur um. Neben diesen Basismodulen wurde eine Modulhierarchie entwickelt, in die die Basismodule eingeordnet wurden. Diese Modulhierarchie ist von allgemeinem Charakter und lässt es zu, beliebige weitere Module hinzuzufügen. Auch diese Hierarchie hat ihr Gegenstück in Java in Form von abstrakten Superklassen.

Die umgesetzte Modulhierarchie enthält die folgenden OWL-Klassen:

- `owl#Thing`: Superklasse in OWL von der alle weiteren Klassen abgeleitet sein müssen.
    - `Module`: Superklasse für die Erstellung von Modulen. Alle erstellten Module und Input-Modul-Charakterisierungen müssen Subklassen dieser Klasse sein.
        - `EndModule`: Superklasse für all jene Module, die im Modulbaum bzw. Modulwald eine Wurzel darstellen. Module dieser Kategorie können keinem andern Modul als `InputModule` dienen, deshalb können Module dieser Kategorie keinen Input-Modul-Charakterisierungen zugeordnet sein.
            - `OptimizationModule`: Dies sind all jene Module deren Wert optimiert werden soll. Zur Identifikation in der Ergebnisausgabe, welche Zielfunktion mit diesem Modul optimiert wurde, muss jeder Instanz davon grundsätzlich eine `hasOptimizationDescription` angehängt werden.
                - `Maximization`: Gibt dem Optimierungsalgorithmus die Anweisung diesen Wert zu maximieren.
                - `Minimization`: Gibt dem Optimierungsalgorithmus die Anweisung diesen Wert zu minimieren.
            - `RestrictionModule`: Beinhaltet all jene Module die ein binäres Ausschlusskriterium für Lösungsinstanzen darstellen.
                - `RestrictionEqualTo`: Überprüft die Gleichheit eines `hasLeftInputModule` mit einem `hasRightInputModule` vom Typ `SingleValue`.
                - `RestrictionSmallerEqualThan`: Überprüft, ob das `hasLeftInputModule` kleiner

oder gleich dem `hasRightInputModule` vom Typ `SingleValue` ist.

- `MidModule`: Umfasst all jene Module, die im Modulwald weder Wurzel noch Blatt sind. Also Module die sowohl selbst InputModules besitzen als auch selbst welche sein können.
  - `Divide`: Dividiert ein `hasLeftInputModule` mit einem `hasRightInputModule` vom Typ `SingleValue`.
  - `Pow`: Potenziert ein `hasLeftInputModule` mit einem `hasRightInputModule` vom Typ `SingleValue`.
  - `Product`: Multipliziert mehrere `hasInputModule` vom Typ `SingleValue` miteinander.
  - `Subtract`: Subtrahiert ein `hasLeftInputModule` mit einem `hasRightInputModule` vom Typ `SingleValue`.
  - `Sum`: Summiert mehrere `hasInputModule` vom Typ `SingleValue` miteinander.
- `StartModule`: Dies sind all jene Module, die die Blätter im Modulbaum darstellen, also Module die selbst keine InputModules haben. Diese Module müssen deshalb allesamt aus sich selbst und nicht etwa aus InputModulen heraus einen Wert bzw. Werte generieren.
  - `DesignModule`: Superklasse von Modulen deren Wert erst im Optimierungsschritt vom Optimierungsalgorithmus bestimmt wird. Alle Module dieser Kategorie müssen das DataProperty `hasDesignName` angehängt haben, welches einen beschreibenden Namen dieses Moduls anhängt. Damit ist eine Identifikation für den Benutzer in der Ergebnisausgabe möglich.
    - `DesignSingleValueBool`: Modul, das die Werte `true` (1) oder `false` (0) vom Optimierungsalgorithmus gesetzt bekommt.

    - `DesignSingleValueFloat`: Modul, das Fließkommawerte vom Optimierungsalgorithmus gesetzt bekommt. Zusätzlich müssen noch Angaben zur oberen und unteren Schranke mittels `hasRangeMax` und `hasRangeMin` gemacht werden, da sonst der Suchraum schnell (vor allem bei Benutzung mehrerer solcher Module) zu groß wird.
    - `DesignSingleValueInt`: Modul, dessen ganzzahliger Wert vom Optimierungsalgorithmus gesetzt wird. Zusätzlich müssen noch Angaben zur oberen und unteren Schranke mittels `hasRangeMax` und `hasRangeMin` gemacht werden.
- `FixedModule`: Diese Modulkategorie bezeichnet Module die bereits einen festen Wert nach der Modellierung in der Wissensdatenbank besitzen. Diese Module haben entweder von sich aus einen festen Wert oder bekommen diesen durch ein `ModuleProperty` der konkreten Instanz angehängt.
    - `FixedSingleValueFloat`: Ein `FixedModule`, das als fixen Wert eine einzelne Fließkommazahl enthält.
- `InputModule`: Diese OWL-Superklasse beinhaltet Module, deren Wert vom Endbenutzer eingegeben werden soll. Um dem Benutzer eine Eingabebeschreibung zu geben, kann an jeder `InputModule`-Instanz eine `hasInputDescription` angehängt werden.
    - `InputSingleValueFloat`: Ein `InputModule`, das eine einzelne Fließkommazahl vom Nutzer als Eingabe fordert.

Die folgenden OWL-Klassen charakterisieren die obigen Modulklassen bezüglich ihrer Eignung als InputModule:

- `InputModuleSpecification`: Diese Klasse beschreibt Interfaces, nach denen die Module bezüglich ihrer Eignung als `InputModule` von anderen Modulen charakterisiert werden. Die Interfaces beschreiben also vor allem was das Modul aus- bzw. weitergeben kann.
  - `SingleValue`: Ein Interface, das als Ausgabe von den untergeordneten Modulen einen einzelnen Wert verlangt, der dann in folgenden Modulen weiterverarbeitet werden kann. Subklassen sind alle oben beschriebenen Module, die selbst dieses Interface implementieren.
    - `DesignSingleValueBool`
    - `DesignSingleValueFloat`
    - `DesignSingleValueInt`
    - `Divide`
    - `FixedSingleValueFloat`
    - `InputSingleValueFloat`
    - `Pow`
    - `Product`
    - `Subtract`
    - `Sum`

7.1.1.2 Object Properties

Zur Modellierung der Kanten im Modulwald werden Object Properties genutzt.

Die folgenden Object Properties sind in diesem ersten Basismodulkatalog enthalten:

- `topObjectProperty`: Ist die OWL-Superklasse für alle Object Properties.
  - `hasInputModule`: Klasse um InputModules eines Moduls zu beschreiben. Wenn keine genauere Charakterisierung des InputModules nötig ist, kann die Modellierung direkt über dieses Property vorgenommen werden, ansonsten über ein Sub-Property von `hasInputModule`.
    - `hasLeftInputModule`: Für Module die zwei InputModules haben können, die sich nicht anhand ihrer `InputModuleSpecification` unterscheiden, kann mit die-

sem Modul eine Unterscheidung getroffen werden und das Modul als linkes InputModule festgelegt werden.

- `hasRightInputModule`: Siehe `hasLeftInputModule`. Legt ein InputModule als rechtes InputModule fest.

Es können bei Modulen, die eine genauere Unterscheidung des InputModules erfordern, weitere Subklassen von `hasInputModule` hinzugefügt werden. Einzige Bedingung für die Verwendbarkeit ist, dass die Java-Implementierung des Moduls, die in der Modulinstanz verwendeten `hasInputModule`-Sub-Properties kennt und auf sie reagieren kann.

7.1.1.3 Data Properties

Die Data Properties hängen Informationen an die Modulinstanzen (OWL-Individuals) an. Die folgenden Data Properties stellt das Schema-File bereit:

- `topDataProperty`: Ist die OWL-Superklasse für Data Properties.
  - `hasModuleDataProperty`: Superklasse für Properties, die in die ModuleData-Klasse geschrieben werden.
    - `hasInputDescription`: Beschreibung, der vom Benutzer zu tätigenden Eingabe. Enthält eine Beschreibung, welche Eingaben der Benutzer machen muss, um das zugehörige Input-Module zu füllen.
  - `hasModuleProperty`: Superklasse für Properties, die dem Modul zugehörig sind.
    - `hasDesignProperty`: Enthält alle Properties, die an Design-Module angehängt werden können.
      - `hasDesignName`: Hängt einen String an, der einen für den Benutzer lesbaren Bezeichner für das Design-Modul enthält.
      - `hasRangeMax`: Gibt den Maximalwert des Intervalls einer Designvariablen vom Typ `DesignSingleValueFloat` oder `DesignSingleValueInt` an.
      - `hasRangeMin`: Gibt den Minimalwert des Intervalls einer Designvariablen vom Typ `DesignSingleValueFloat` oder `DesignSingleValueInt` an.
    - `hasFixedProperty`: Enthält alle Properties, die einem `FixedModule` angehängt werden können.

        - `hasFixedPropertyValue`: Hängt einen fixen Fließkommawert einem Modul vom Typ `FixedSingleValueFloat` an.
    - `hasOptimizationProperty`: Enthält Properties, die an Module vom Typ `OptimizationModule` angehängt werden können.
        - `hasOptimizationDescription`: Enthält eine für den Benutzer lesbare Beschreibung, welche funktionale Eigenschaft dieses Modul optimiert.

Wenn neue Module erstellt werden, sind weitere Subklassen von `hasModuleProperty` denkbar. Lediglich die neuen Module, die diese neuen Subklassen verwenden, müssen sie auch berücksichtigen. Andere Module und auch die sonstige Programmstruktur bleiben davon unberührt.

7.1.1.4 Annotation Properties

Annotation Properties können an eine beliebige Entität angehängt werden. So werden diese vor allem dann verwendet, wenn Eigenschaften an OWL-Klassen, andere Properties oder an die Ontologie selbst angehängt werden sollen.

Das entwickelte Schema stellt die folgenden Annotation Properties bereit:

- `hasInputDescriptionType`: Falls einem Modul mehrere `InputDescriptions` angehängt werden, kann hiermit eine Unterscheidung für das Modul ermöglicht werden.
- `hasOptimizationAlgorithm`: Ist eine Superklasse für alle Annotation Properties den Optimierungsalgorithmus betreffend.
    - `hasEvolutionaryOptimizationAlgorithm`: Gibt an, dass dieses Problem mit einem Evolutionären-Optimierungsalgorithmus gelöst werden soll.
        - `hasEvolutionaryAlpha`: Gibt die Größe der Population an.
        - `hasEvolutionaryCrossoverRate`: Gibt die Anzahl der Überführungen von einer Generation in die nächste Generation an.
        - `hasEvolutionaryGenerations`: Gibt die Anzahl an Generationen an, die der Algorithmus durchführt.

        - `hasEvolutionaryLambda`: Gibt die Anzahl an Nachkommen pro Generation an.
        - `hasEvolutionaryMu`: Gibt die Anzahl an Eltern pro Generation an.
    - `hasRandomSearchOptimizationAlgorithm`: Gibt an, dass als Optimierungsalgorithmus eine randomisierte Suche genutzt wird.
        - `hasRandomSearchBatchsize`: Gibt die Anzahl an Instanzen pro Generation an.
        - `hasRandomSearchIterations`: Gibt die Anzahl an Iterationen an, die der Algorithmus durchführt.
- `isJavaClass`: Stellt eine Verbindung zwischen der OWL-Klasse zur Java-Klasse dar und ermöglicht so eine Instanziierung von entsprechenden Java-Klassen aus einer Problemmodellierung aus OWL-Modulen.
- `problemCharacterization`: Hängt an die Ontologie eine Beschreibung des Problems an.
- `problemName`: Hängt an die Ontologie einen Namen für das modellierte Problem an.

Erweiterungen in den Annotation Properties sind vor allem dann nötig, wenn weitere Optimierungsalgorithmen (die auch andere Parameter benötigen) bereitgestellt werden sollen.

7.1.2 Problemmodellierungen

Die Problemmodellierung geschieht durch Erstellung einer neuen Ontologie, die die in 7.1.1 erstellte Schema-Ontologie importiert. Jedes Problem wird in einer eigenen Ontologie-Datei erstellt.

Eine Problemmodellierung beinhaltet die folgenden Bestandteile:

- Importieren der Schema-Ontologie
- Erstellung der notwendige Modulinstanzen als Individuals
- Festlegung der Modulbeziehungen durch die Definition der Input-Module
- ModuleDataProperties und ModuleProperties der Modulinstanzen
- Problemnamen und Problembeschreibung der Ontologie
- Informationen über den zu verwendenden Optimierungsalgorithmus und dessen Parameter

Zur Validierung der Modellierbarkeit von Problemen wurde das im Folgenden beschriebene Testproblem modelliert. Das Testproblem besteht aus 4 Modulen. Davon eine Design-Variable, eine Input-Variable, ein Optimierungs-Modul und ein Restriktionsmodul. Es wird die Design-Variable durch das Optimierungs-Modul maximiert und über das Restriktions-Modul sichergestellt, dass die Design-Variable nicht größer sein darf als die Zahl die durch das Input-Modul vom Benutzer abgefragt wird. Der Modulbaum dieser Modellierung ist in Abbildung 22 dargestellt.

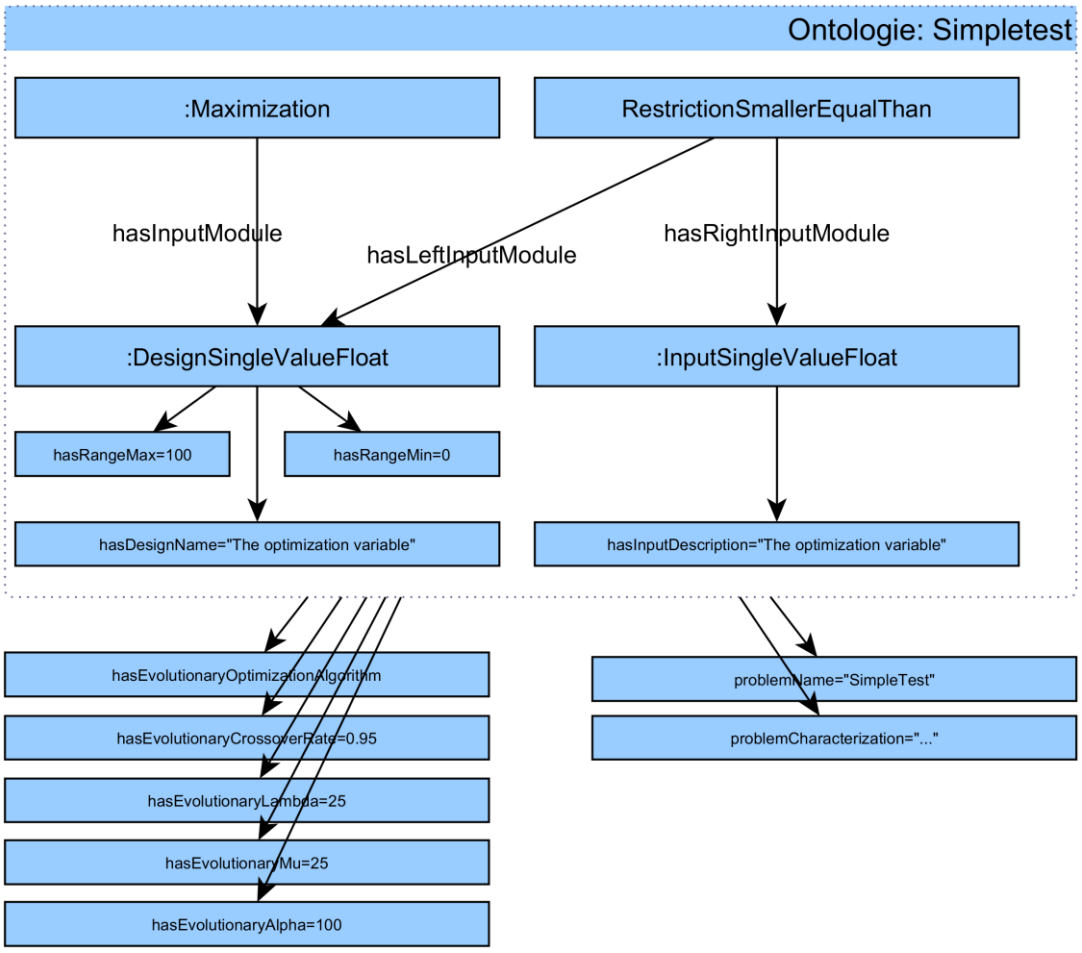


Abbildung 22: Problemmodellierung eines einfachen Testproblems als Graph visualisiert.

7.1.3 Zielkonflikte bei der Modellierung

Im Modellierungsprozess bleiben eine Reihe gestalterischer Freiheiten erhalten, das heißt ein und dasselbe Problem lassen sich auf unterschiedliche Art und Weise modellieren. Besonders viele Freiheiten ergeben sich, wenn man die Entwicklungsmöglichkeit

von neuen Modulen miteinbezieht. Wenn sich solche Alternativen ergeben, ist nicht immer klar, welche Alternative die bessere ist, da jede Modellierungsoption ihre eigenen Vor- und Nachteile bzgl. der angestrebten Modellierungsziele hat. Bei solchen Abwägungen spricht man von Zielkonflikten (engl. trade-off).

Sinnvolle Modellierungszielsetzungen sind:

- Gute Kapselung: Möglichst wenige, kleine, vielseitige Module, die ein großes Anwendungsspektrum abdecken. Eine gute Kapselung bedeutet gleichzeitig einen geringeren Entwicklungsaufwand für den Modulentwickler.
- Gute Benutzbarkeit für den (End-)Benutzer: Der (End-) Benutzer soll von den Modulen so gut wie möglich bei der Problemauswahl, Parametrisierung und Interpretation der Ergebnisse unterstützt werden.
- Gute Benutzbarkeit für den Problemdesigner: Auch dem Problemdesigner soll es so einfach wie möglich gemacht werden. Das heißt er soll möglichst wenige Module instanziieren müssen. Die Module sollen dabei möglichst gut auf sein spezifisches Problem zugeschnitten sein, was die Auswahl vereinfacht und die Modulparametrisierung gering hält.

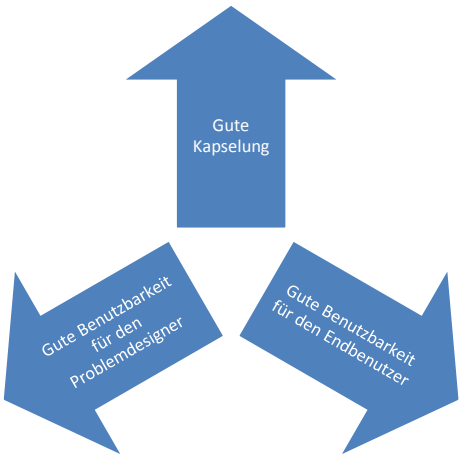


Abbildung 23: In Zielkonflikt stehende Modellierungszielsetzungen bei der Entwicklung eines Modulkatalogs

Diese Modellierungszielsetzungen widersprechen sich allerdings zum Teil stark. So widersprechen sich beispielsweise die Kapselung und die Benutzbarkeit für den Problemdesigner stark. Während kleine universelle Module viele Modulinstanzen mit einer starken Parametrisierung erfordern, macht die damit erzielte gute Kapselung die Modulauswahl schwierig, da ein gut gekapseltes Modul mit möglichst vielen Modulen kom-

patibel sein muss. Gut zu modellierende Module sollten dagegen auf bestimmte Input-Module spezialisiert sein und akzeptieren deshalb auch nur diese als Input-Module.

Einen anderen krassen Gegensatz stellen die Kapselung und die gute Benutzbarkeit für den Benutzer dar. Den Benutzer interessieren vor allem die Input-Module, diese sollten ihn möglichst gut durch das Problem führen. Dies gelingt meist dann am besten, wenn die Module möglichst exakt auf das Problem zugeschnitten sind. Dann geht aber die Generalität der Module verloren und solche Module können dann nur in einem sehr engen Anwendungsspektrum angewandt werden. Ein Beispiel für solch einen Fall wäre: Ein Automobildesigner möchte für sein zu lösendes Optimierungsproblem die Felgengröße parametrisieren. Die ideale Benutzbarkeit wird durch ein Input-Modul erreicht, welches die gewählten Felgen am Auto anzeigt und so die Auswahl erleichtert. Die optimale Kapselung wird allerdings erreicht, wenn ein universelles Modul die Felgengröße in Zoll als Ganzzahl ohne visuelle Unterstützung vom Benutzer abfragt.

Auch die letzte Kombination der Modellierungszielsetzungen von guter Benutzbarkeit für den Benutzer und guter Benutzbarkeit für den Problemdesigner widerspricht sich. So ergeben Module mit starker Parametrisierung meist eine gute Benutzbarkeit für den (End-)Benutzer, für den Problemdesigner birgt eine starke Parametrisierung der Module jedoch zusätzliche Komplexität und führt darüber hinaus zu zusätzlichem Arbeitsaufwand.

7.2 Entwicklung des Java-Teils

Im folgenden Abschnitt wird die Entwicklung des Java-Teils beschrieben. Es handelt sich dabei um das Expertensystem an sich, das entwickelt wird. Das Expertensystem muss die Modellierungen der Wissensdatenbank verarbeiten und damit den Benutzer im Problemformulierungsprozess unterstützen.

7.2.1 Struktur

Die Programmstruktur wurde in 54 Java-Klassen aufgegliedert, die in 13 Packages unterteilt sind.

Es wurde die folgende Package-Hierarchie erstellt, um die Klassen übersichtlich einzuteilen:

- `de.uni_stuttgart.iris.problem_formulator`: Basis-Package, das das gesamte Problemformulierungstool enthält.
  - `client`: Enthält Klassen, die ausschließlich auf dem Client verwendet werden.
    - `gui`: Umfasst alle Klassen, die die grafische Benutzerschnittstelle betreffen.
      - `module_widgets`: Enthält die Widgets von Input-Modulen und eine Superklasse von der die Modul-Widgets abgeleitet werden.
  - `server`: Enthält Klassen, die ausschließlich auf dem Server verwendet werden.
    - `problem_calculator`: Enthält Klassen, die der Berechnung des Optimierungsproblems dienen.
    - `problem_logic`: Enthält Klassen, die zusammen das Optimierungsproblem darstellen.
      - `module`: Enthält alle Klassen, die die Problem-Module betreffen.
        - `structure`: Enthält alle abstrakten Superklassen, von denen die implementierten Module abgeleitet sind.
        - `interfaces`: Enthält Interfaces, die das Gegenstück zu den in der Wissensdatenbank verwendeten `InputModuleSpecifications` darstellen.
        - `implemented`: Enthält die konkreten Module, aus denen das Problem besteht. Hier sind die mit Funktionalität gefüllten Gegenstücke zu den in OWL erstellten Modul-Klassen zu finden.
    - `tools`: Enthält verschiedene Hilfsklassen, die Funktionalitäten anbieten, die in der Entwicklung gebraucht werden. So z.B. der Zugriff auf Dateien oder das Auslesen von Ontologien.
  - `shared`: Enthält Klassen, die sowohl dem Client als auch dem Server bekannt sein sollen.

- `module_data`: Enthält die Module-Data-Datenstrukturen um Informationen jedes Input-Moduls zwischen dem Server und Client auszutauschen.

Es ist gut zu erkennen, dass sich die Entscheidung zum internetgestützten Ansatz und damit zum Client-Server-Modell auf die oberste Ebene der Strukturierung des Quellcodes auswirkt. Erst nachdem der Quellcode auf Client, Server oder Shared strukturiert ist, wird weiter nach Funktionalitäten der Klassen aufgeteilt.

Die gesamte Kommunikation zwischen Client und Server findet über die Klassen `ProblemFormulator` (liegt auf dem Client) und `ServiceImpl` (liegt auf dem Server) statt. Abbildung 24 zeigt ein UML-Sequenzdiagramm, welches einen kompletten Programmablauf des Problemformulierungstools zeigt. Dabei wird sowohl die Kommunikation zwischen diesen beiden Klasseninstanzen gezeigt, als auch die von diesen Objekten initiierten Aufrufe auf Client- bzw. Server-Seite.

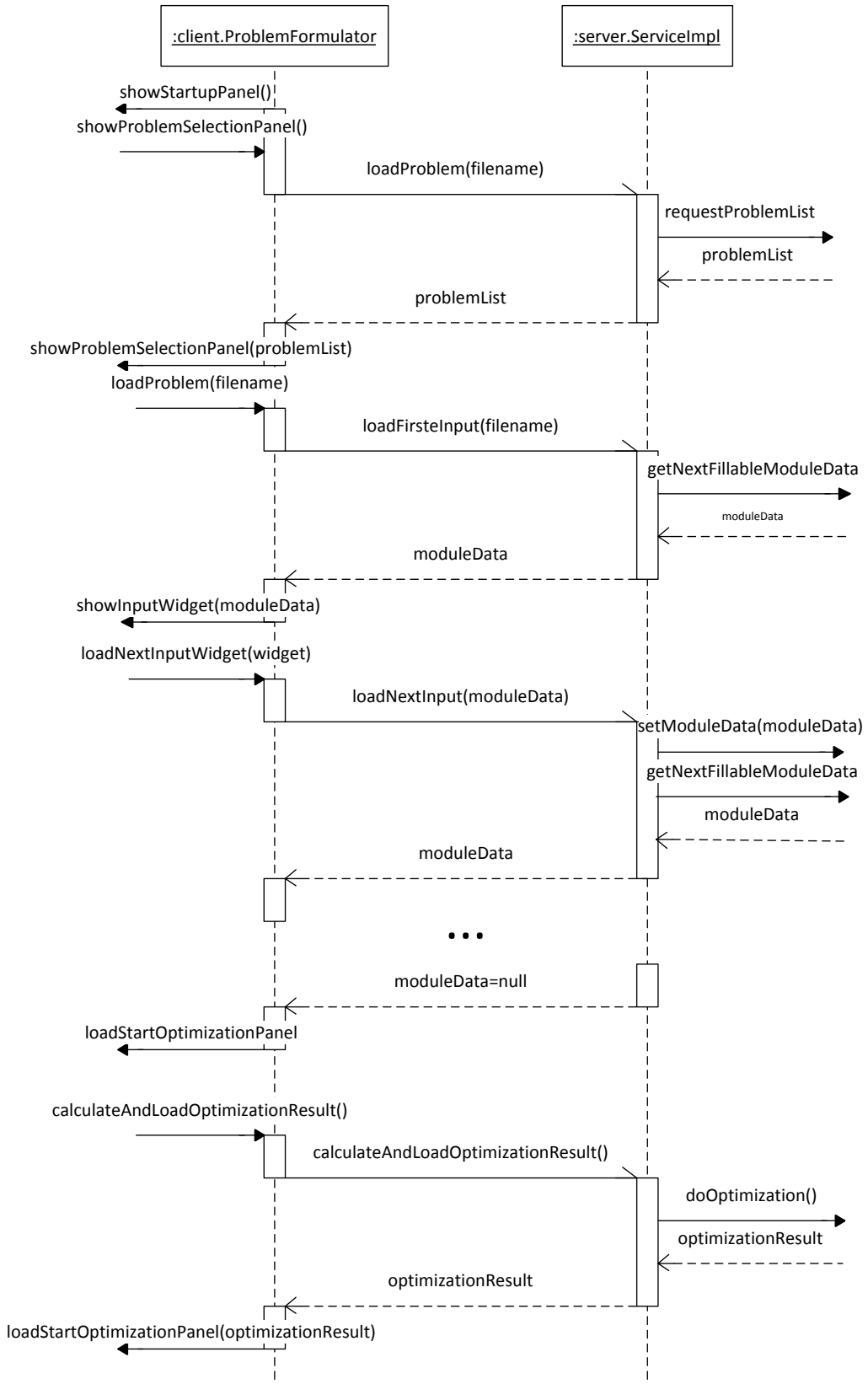


Abbildung 24: UML-Sequenzdiagramm über den Ablauf des Problemformulierungsprozesses für Instanzen der Klassen ProblemFormulator (Client) und ServiceImpl (Server)

Die abstrakten Superklassen im Package `server.problem_logic.structure` erfüllen bereits so viele Aufgaben wie möglich ihrer Subklassen, um die Implementierung der Subklassen zu vereinfachen und Redundanz im Quellcode zu verhindern. Die Klassen sind nach folgendem Schema strukturiert:

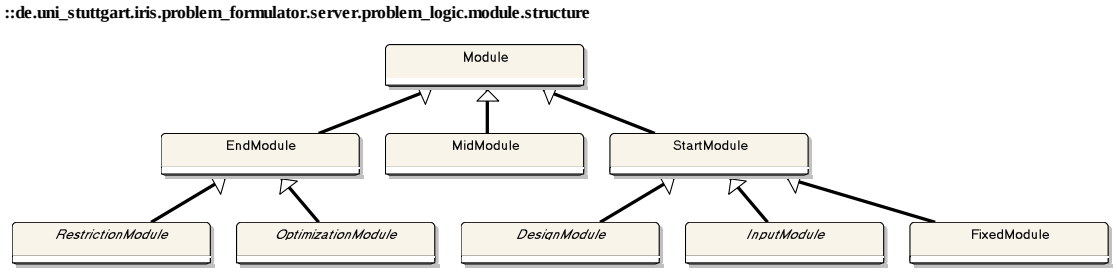


Abbildung 25: Klassendiagramm der Klassen im Package server.problem_logic.structure

Die Zuordnung der konkreten Modulklassen zu den obigen Superklassen ist in Abbildung 26 dargestellt.

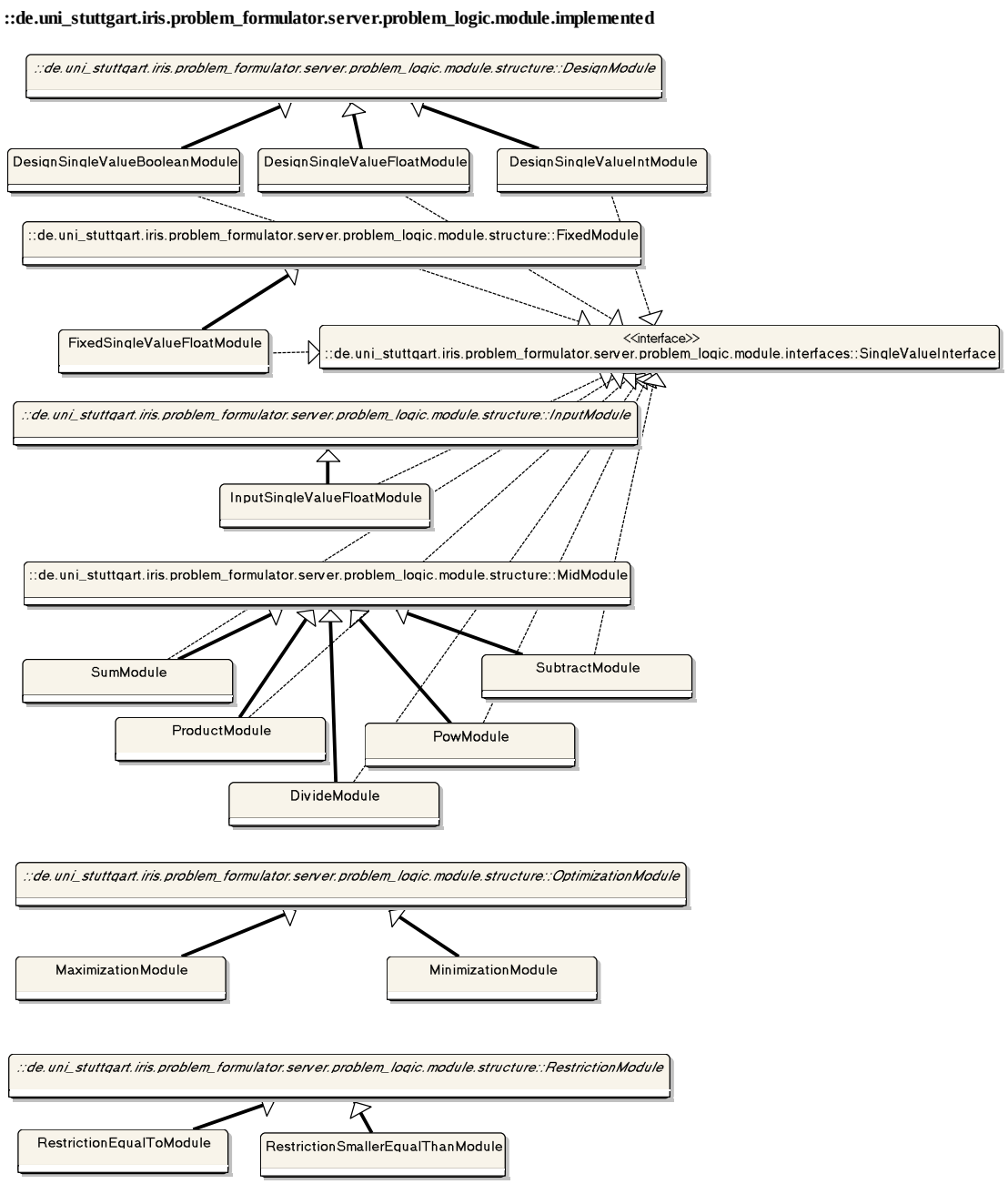


Abbildung 26: Klassendiagramm der Klassen im Package server.problem_logic.module.implemented mit Zuordnungen zu den structure-Klassen

Genau wie die Module ist auch der Berechnungsteil in `server.problem_calculator` stark hierarchisch aufgebaut. Die Klasse `ProblemCalculator` implementiert bereits alle Funktionalitäten, die für die Berechnung benötigt werden. Lediglich die Bereitstellung und Konfiguration eines konkreten Optimierungsalgorithmus wird den Subklassen überlassen.

::de.uni_stuttgart.iris.problem_formulator.server.problem_calculator

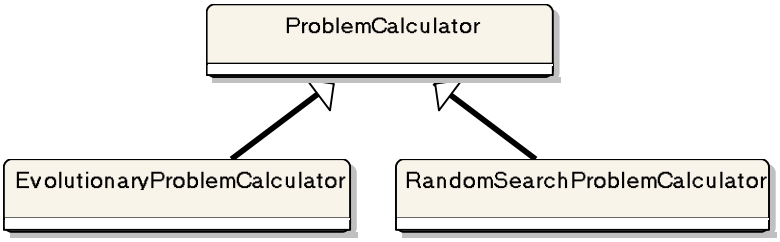


Abbildung 27: Klassendiagramm der Klassen zur Berechnung der optimalen Lösung für das Optimierungsproblem

7.2.2 Kernelemente der Implementierung

Um den Rahmen dieser Arbeit nicht zu sprengen, kann bei weitem nicht auf alle Details des Quellcodes eingegangen werden. Um einen detaillierten Einblick in die Implementierungsdetails zu bekommen, sei deshalb auf den gut dokumentierten Quellcode selbst verwiesen. Trotzdem sollen in diesem Abschnitt einige Schlüsselstellen im Quellcode beschrieben werden.

Einen wichtigen Teil der Entwicklung stellt die Abfrage der Wissensdatenbank in Form von SPARQL-Abfragen dar. Eine solche Abfrage stellt die Untersuchung der Modulhierarchie-Beziehungen dar. Die Modulhierarchie muss wie in der Problemdatei definiert nach der Instanziierung der Java-Module auch auf die Java-Module übertragen werden. Die dazu nötige SPARQL-Abfrage ist in Listing 7 aufgeführt. Die Abfrage wird in der Funktion `addInputModuleDependencies()` der Klasse `ProblemManager` verwendet. Die Schwierigkeit in der Abfrage besteht darin, stets das spezifischste Object Property vom Typ `InputModule` abzufragen. Dies wird durch die 2 ineinander geschachtelten Filter erreicht.

```
PREFIX schema:
<http://www.semanticweb.org/flo/ontologies/2014/6/problemformulator_s
chema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT DISTINCT ?module ?inputmoduletransition ?inputmodule
WHERE {
  ?module rdf:type ?moduletype
. ?moduletype rdfs:subClassOf* schema:Module
. ?module ?inputmoduletransition ?inputmodule
. ?inputmoduletransition rdfs:subPropertyOf+ schema:hasInputModule
. FILTER NOT EXISTS {
  ?module ?inputmoduletransition2 ?inputmodule
. ?inputmoduletransition2 rdfs:subPropertyOf ?inputmoduletransition
. FILTER(?inputmoduletransition != ?inputmoduletransition2)
                    }
      }
```

Listing 7: SPARQL-Abfrage der InputModule-Beziehungen zwischen allen Modulen

Ein weiteres Kernelement in der Implementierung stellt die Definition von Creator, Decoder und Evaluator (siehe Abbildung 7 auf Seite 33) für den Optimierungsprozess mit Opt4J dar.

Zunächst einmal muss der Genotype definiert werden:

```
public static class ProblemGenotype
      extends CompositeGenotype<String, Genotype>
{
      public ProblemGenotype()
      {
            super();
      }

      public void addGenotypesOfModules(List<Module> designModules)
            throws CanNotMapOntologyRelationException
      {
            for(Module module : designModules)
            {
                  DesignModule dm = (DesignModule) module;
                  this.put(dm.getModuleName(),
                        dm.createNewGenotype());
            }
      }

      public Genotype getGenotypeForKey(String key)
      {
            return this.get(key);
      }
}
```

Listing 8: Definition des Genotypes des Problems im ProblemCalculator

Der Genotype ist abgeleitet vom `CompositeGenotype`, der selbst eine Zusammenstellung von weiteren Genotypes repräsentiert. Dieser bekommt zusätzlich noch eine Funktion zum Auslesen einer Liste von Design-Modulen. Die Design-Module sind in der Lage selbst ihren Genotype zu erzeugen und zurück zu geben. Diese erzeugten Genotypes werden alle zum `CompositeGenotype` hinzugefügt.

Der Creator muss nun nur noch diesen Genotype erzeugen, dazu wird zunächst eine Instanz des obigen Genotypes erzeugt. Danach werden erst alle Design-Module abgefragt um dann die oben angesprochene Funktion `addGenotypesOfModules(…)` zu nutzen. Die Definition des Creators zeigt Listing 9.

```
public static class ProblemCreator
      implements Creator<ProblemGenotype>
{
      public ProblemGenotype create()
      {
            ProblemGenotype genotype = new ProblemGenotype();

            List<Module> designModules
                  = problemManager.getDesignModuleList();

            try
            {
                  genotype.addGenotypesOfModules(designModules);
            }
            catch (CanNotMapOntologyRelationException e)
            {
                  e.printStackTrace();
            }

            return genotype;
      }
}
```

Listing 9: Definition des Creators aus den Problem-Modulen für Opt4J

Schließlich muss noch der Decoder definiert werden. Dieser extrahiert die Werte der Optimierungs- und Restriktionsmodule aus dem Problem und speichert sie in der Klasse `OptimizationResult`. Dazu müssen erst mit der Funktion `pushOptimizationResultInProblem(genotype)` die Werte der Design-Variablen der Genotype-Instanz in die Designmodule geschrieben werden. Nun sind die Werte aller Start-Module gesetzt und das Problem kann ausgewertet werden. Die Aufgabe, die Optimierungsergebnisse in ein Objekt vom Typ `OptimizationResult` zu schrieben, übernimmt die Funktion `getOptimizationResultFromProblem()`. Die Implementierung des Decoders ist in Listing 10 aufgeführt.

```
public static class ProblemDecoder
      implements Decoder<ProblemGenotype, OptimizationResult>
{
      public OptimizationResult decode(ProblemGenotype genotype)
      {

      ProblemCalculator.pushOptimizationResultInProblem(genotype);

            OptimizationResult result = null;

            try
            {
                  result = ProblemCalculator
                        .getOptimizationResultFromProblem();
            }
            catch(ValueNotAvailableException e)
            {
                  e.printStackTrace();
            }
            return result;
      }
}
```

Listing 10: Definition des Decoders aus den Problem-Modulen für Opt4J

Der letzte Teil in der Definition des Optimierungsproblems für Opt4J ist die Definition des Evaluators. Dieser erstellt aus dem Phonotype (`OptimizationResult`) `Objectives` die Opt4J verstehen und auswerten kann. Auf die Listung des Quellcodes des deutlich umfangreicheren Evaluators wurde an dieser Stelle verzichtet, um die Code-Listings übersichtlich zu halten.

Wenn diese drei Kernelemente der Problemdefinition in Opt4J definiert sind, kann daraus eine `OptimizationProblemModule`-Klasse erzeugt werden, die dann mit einem entsprechend konfigurierten Optimierungsalgorithmus gelöst werden.

7.3 Vorgehen zum Erstellen neuer Probleme

In diesem Abschnitt wird das Vorgehen zum Erstellen neuer Probleme aus den bestehenden Modulen beschrieben. Ergebnis dieses Prozesses ist eine OWL-Datei, die das Problem computerlesbar beschreibt.

7.3.1 Modellierungsbeispiel

Das Vorgehen soll anhand eines einfachen Beispiels veranschaulicht werden. Das Beispiel handelt von der effizienten Einzäunung einer Weidefläche. Problemstellung ist die Folgende:

Es soll vom Benutzer die maximal verfügbare Zaunlänge abgefragt werden. Mit diesem Zaun gilt es eine rechteckige Fläche einzuzäunen, wobei auf der linken Seite ein Fluss verläuft (wodurch auch die Versorgung des Weideviehs mit Wasser sichergestellt wird). Deshalb wird auf dieser Seite kein Zaun benötigt. Es sind also nur drei Seiten des Rechtecks (Oben, Rechts und Links) einzuzäunen. Die Optimierung gilt der Maximierung des Flächeninhalts der Weidefläche.

Mathematisch ist das Problem in Formel 12 abgebildet. Dass das Problem auch durch simple Differenzierung der Optimierungsfunktion gelöst werden kann, zeigt Formel 13. Trotzdem ist das Problem als Beispiel an dieser Stelle sehr anschaulich.

$$\max(x * y)$$

$$s.t.$$
$$(2x + 1y) \leq length_{max}$$

Formel 12: Mathematische Formulierung des Weideflächenproblems

$$(2x + 1y) \leq length_{max}$$
$$\Leftrightarrow y \leq length_{max} - 2x \ (1)$$

$$\max(x * y)$$
$$\Leftrightarrow \max(x * (length_{max} - 2x)$$
$$\Leftrightarrow \max(x * length_{max} - 2x^2)$$

$$f(x) = x * length_{max} - 2x^2$$
$$f'(x) = length_{max} - 4x$$

$$f'(x) = 0$$
$$0 = length_{max} - 4x$$
$$\Leftrightarrow \boldsymbol{x = \frac{length_{max}}{4}} \ (2)$$

$$mit\ (2) in\ 1\ eingesetzt\ ergibt\ sich;$$
$$y \leq length_{max} - 2\frac{length_{max}}{4}$$
$$\boldsymbol{y \leq \frac{length_{max}}{2}}$$

Formel 13: Lösung des Weideflächenproblems durch Differenzierung

Wie bereits beschrieben wird die OWL-Modellierung in dieser Arbeit mit dem Modellierungstool Protégé vorgenommen. So werden auch die folgenden Schritte anhand dieses Modellierungstools erklärt. Die Ontologie-Datei kann jedoch manuell im XML-Format erstellt werden.

7.3.2 Erstellen der Ontologie

Der erste Schritt ist das Erstellen der Ontologie. Dazu gehört der Ontologie einen Namen zu geben, als Import die Schema-Ontologie festzulegen und die Annotation-Properties, die direkt an die Ontologie gehören, festzulegen. Zu den Annotation-Properties, die auf jeden Fall gesetzt sein müssen, gehören `problemName` und `problemCharacterizations`, sowie ein Sub-Property vom Typ `hasOptimizationAlgorithm`, damit klar ist mit welchem Algorithmus gearbeitet werden soll. Die Parameter des verwendeten Optimierungsalgorithmus müssen nicht zwangsläufig alle gesetzt werden. Ist ein Parameter nicht gesetzt wird automatisch ein Standardparameter verwendet.

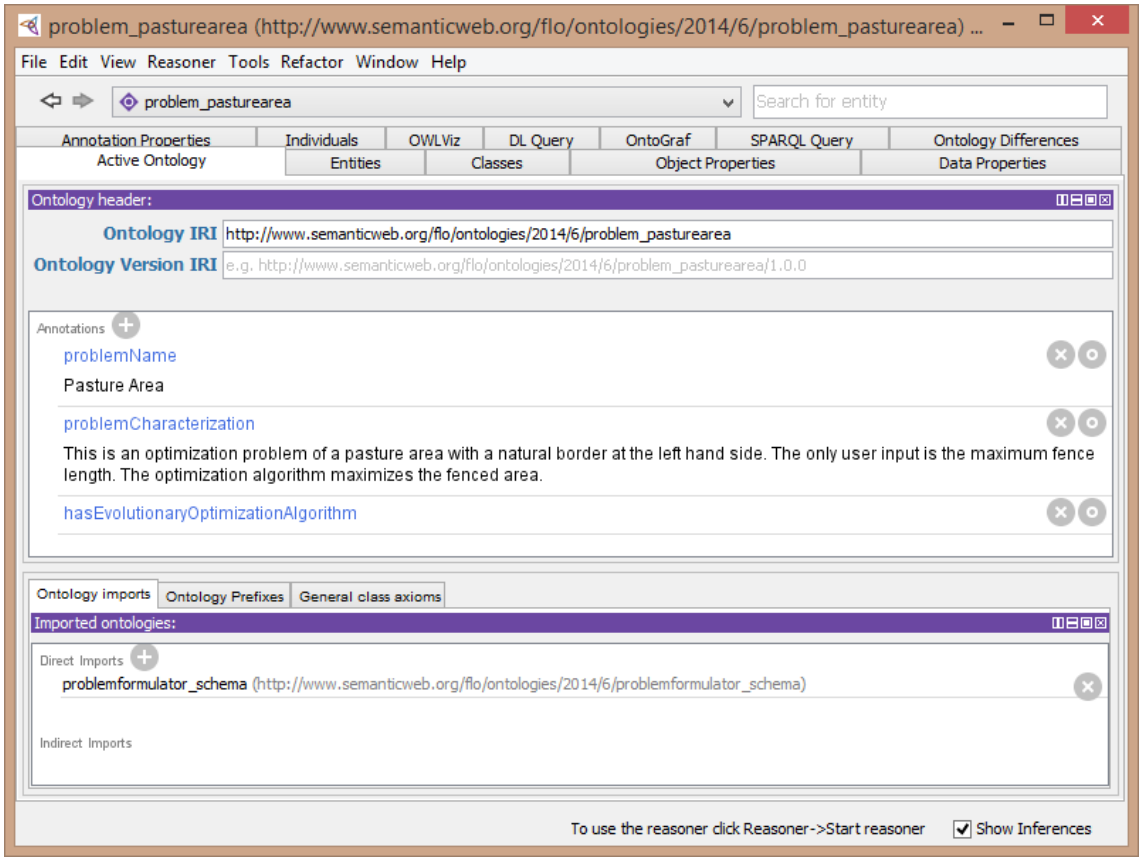


Abbildung 28: Screenshot vom Erstellen der Ontologie und Setzen der direkt an der Ontologie hängenden Properties

7.3.3 Erstellen der Modulinstanzen

Um die benötigten Modulinstanzen zu erstellen werden entsprechend viele Individuals erstellt. Der Name jedes Individuals kann hierbei beliebig gewählt werden und dient nur der Lesbarkeit für den Problemdesigner. Um den Typ eines jeden Individuals festzulegen muss unter Types die entsprechende Modulklasse eingetragen werden.

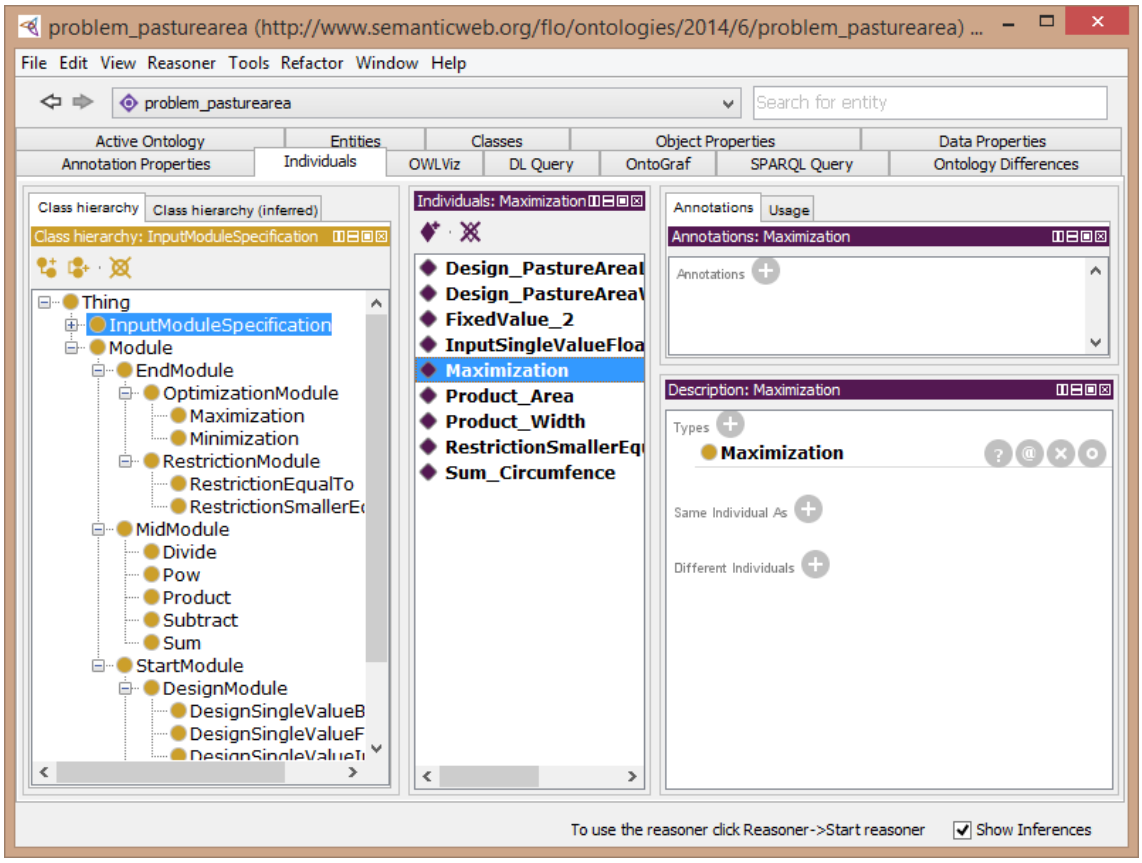


Abbildung 29: Screenshot vom Erstellen der Modulinstanzen

7.3.4 Festlegen der Input-Module-Beziehungen

Um die Modulhierarchie festzulegen, muss bei jedem Modul mittels eines Object-Properties vom Typ hasInputModule festgelegt werden, was die Input-Module sind. Der der Modellierung zugrundeliegende Modulbaum ist in Abbildung 30 dargestellt.

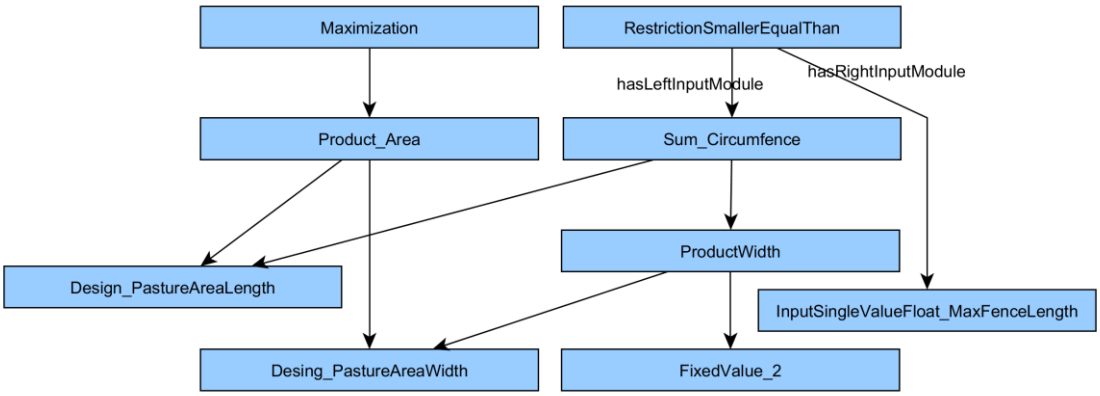


Abbildung 30: Modulbaum des Weideflächenproblems

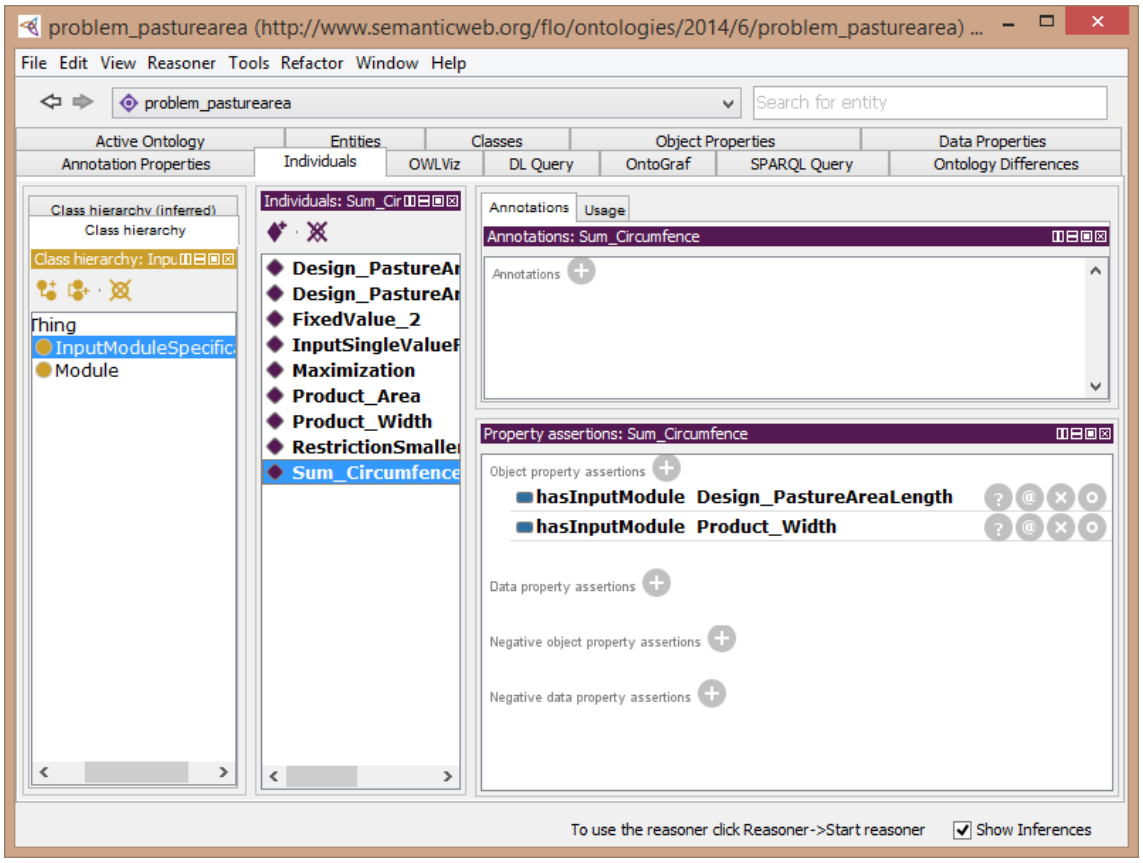


Abbildung 31: Screenshot vom Erstellen der Input-Module-Beziehungen

7.3.5 Setzen der Modulspezifischen Properties

Zusätzlich gibt es Module die spezifische Data Properties verlangen. Diese müssen in diesem Fall für jede Instanz gesetzt werden.

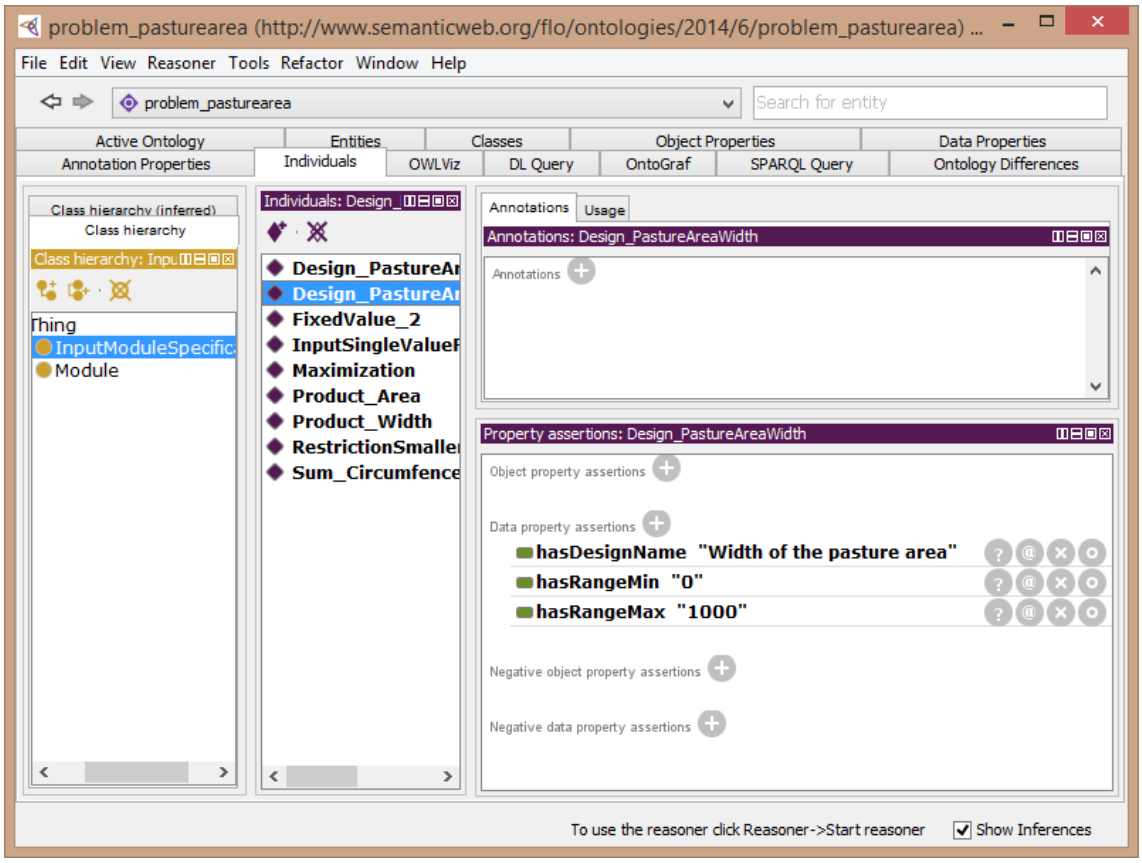


Abbildung 32: Screenshot vom Hinzufügen modulspezifischer Eigenschaften zu den Modul-Instanzen

7.3.6 Laden der Ontologie mit dem Problem Formulator

Da die Ontologie nun vollständig definiert wurde, muss die Ontologie nur noch gespeichert und dem Problem Formulator zum Laden bereitgestellt werden. Dazu muss die Ontologie im Format RDF/XML gespeichert und dann die *.owl-Datei im Ordner `\war\WEB-INF\ontology\problems` abgelegt werden. Nun steht das neue Problem dem Problem Formulator zur Verfügung.

7.4 Vorgehen zum Erstellen neuer Module

In diesem Abschnitt soll die Entwicklung neuer Module an einem Beispiel erläutert werden. Als Beispiel dient die Entwicklung eines `MidModule`, das den Modulo-Operator (%) umsetzt. Das Modul selbst soll die `InputModuleSpecification SingleValue` implementieren.

7.4.1 Implementierung eines neuen Moduls

Zur Implementierung muss zunächst eine neue Klasse im Package `server.problem_logic.module.implemented` erstellt werden. Diese muss eine Sub-Klasse von `MidModule` sein und das Interface `SingleValueInterface` unterstützen. Nun sind drei Methoden zu erstellen. Zunächst muss der Konstruktor erstellt werden, um den Super-Konstruktor aufzurufen. Zum anderen müssen die Methoden `getSingleValue` und `addInputModuleConcreteModuleSpecific` erstellt werden. Die fertig implementierte Klasse ist in Listing 11 aufgeführt.

```
...
public class ModuloModule extends MidModule implements
SingleValueInterface
{
    private SingleValueInterface leftSingleValueInterfaceInputModule;
    private SingleValueInterface
rightSingleValueInterfaceInputModule;

    public ModuloModule(String moduleName)
    {
        super(moduleName);
    }

    @Override
    public double getSingleValue() throws ValueNotAvailableException
    {
        return leftSingleValueInterfaceInputModule.getSingleValue()
            % rightSingleValueInterfaceInputModule.getSingleValue();
    }

    @Override
    protected void addInputModuleConcreteModuleSpecific(
            String moduleTransitionType, Module inputModule)
            throws CanNotMapOntologyRelationException
    {
        if(moduleTransitionType.equals("hasLeftInputModule"))
        {
            this.leftSingleValueInterfaceInputModule
                        = (SingleValueInterface) inputModule;
        }
        else if(moduleTransitionType.equals("hasRightInputModule"))
        {
            this.rightSingleValueInterfaceInputModule
                        = (SingleValueInterface) inputModule;
        }
        else
        {
            throw new CanNotMapOntologyRelationException(
                        "no such input module possible");
        }
    }
}
```

Listing 11: Implementierung des Modulo-Modules

7.4.2 Einpflegen eines neuen Moduls in die Schema-Ontologie

Um das Modul in die Schema-Ontologie einzupflegen, muss lediglich eine neue Klasse mit den folgenden Eigenschaften erstellt werden:

- SubClass of: `MidModule`
- SubClass of: `SingleValue`
- SubClass of: `hasLeftInputModule` exactly 1 `SingleValue`
- SubClass of: `hasRightInputModule` exactly 1 `SingleValue`
- Annotation: isJavaClass de.uni_stuttgart.iris.problem_formulator.server.problem_logic.module.implemented.ModuloModule

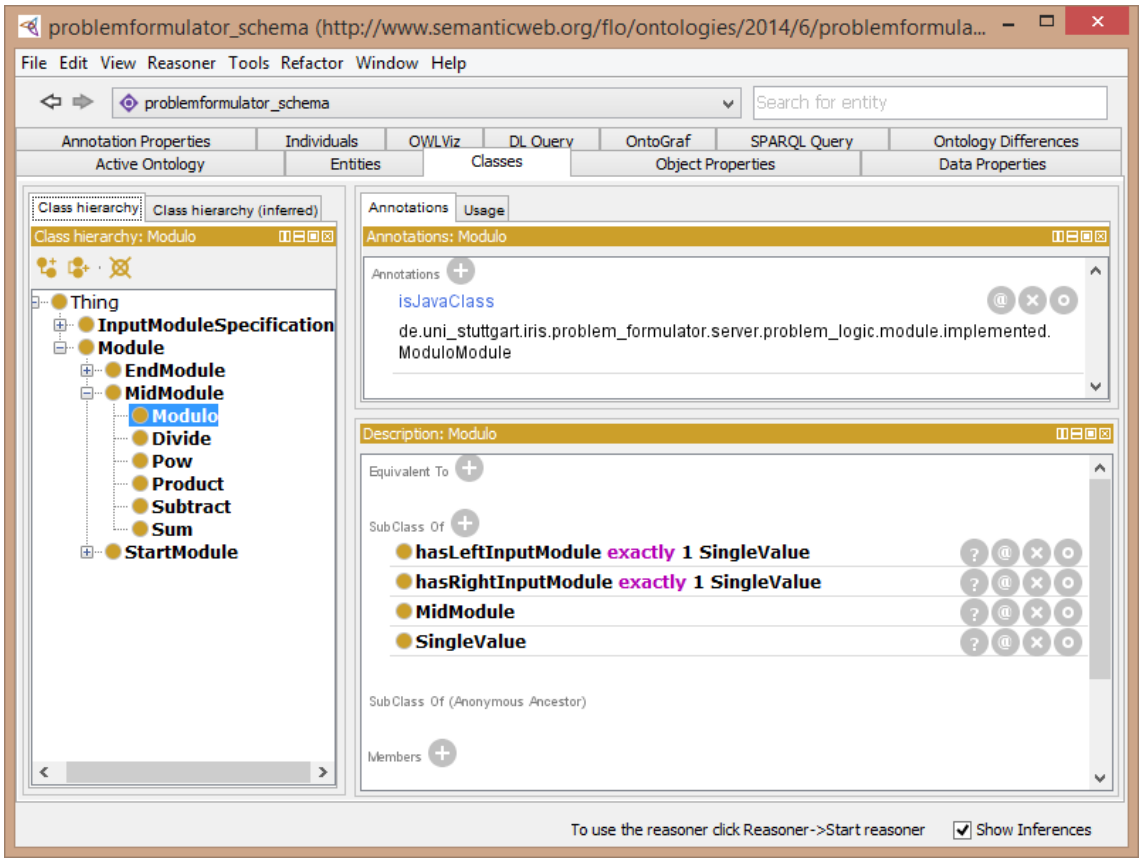


Abbildung 33: Screenshot vom Einpflegen des Modulo-Moduls in die Schema-Ontologie

7.5 Mögliche Erweiterungen

Im folgenden Abschnitt wird auf Erweiterungen eingegangen, die eine sinnvolle Erweiterung des Problem Formulators darstellen. Damit stellt dieser Abschnitt eine Hilfestellung für die Themenauswahl weiterführender Arbeiten dar.

Sinnvolle Erweiterungen des Problem Formulators sind:

- Entwicklung eines speziellen Editors für den Problemdesigner, der dem Benutzer erlaubt den Problembaum grafisch aufzubauen und modulspezifische Properties automatisch hinzufügt.
- Entwicklung eines umfangreicheren Modulkatalogs. Dieser kann abhängig von den Modellierungszielsetzungen (siehe Abschnitt 7.1.3 auf Seite 76) verschiedener Gestalt sein. So bietet sich beispielsweise bei der Zielsetzung einer guten Kapselung an einen Modulkatalog mit einer Art Vektorlogik zu implementieren, der es erlaubt skalierbare Probleme zu modellieren, deren Dimension vom Problemdesigner offen gelassen werden kann.
- Umsetzung weiterer Problemstellungen.
- Ausgabemöglichkeit der mathematischen Formulierung des Problems schaffen. Hierzu muss eine abstrakte Funktion zu der Klasse Module hinzugefügt werden mit der die Formel dann rekursiv erzeugt wird.
- Vereinfachung der Problemauswahl durch eine Volltextsuche, unter Einbeziehung der Problembeschreibung und durch Kategorisierungen der vorhandenen Optimierungsprobleme.
- Durchführung einer Nutzerstudie, die den Zeitgewinn und die Vereinfachung in der Durchführung von Optimierungsaufgaben zeigt.

8 Evaluierung

Im folgenden Kapitel wird die Funktionalität des erstellten Softwaretools zur Unterstützung des Problemformulierungsprozesses dargelegt.

Der Problemformulierungsprozess wird durch den Problem Formulator auf ein Niveau vereinfacht damit er auch für Laien durchführbar ist. Abbildung 34 zeigt den Problemformulierungsprozess bei Auswahl des Weideflächenproblems, das in Abschnitt 7.3.1 eingeführt wurde.

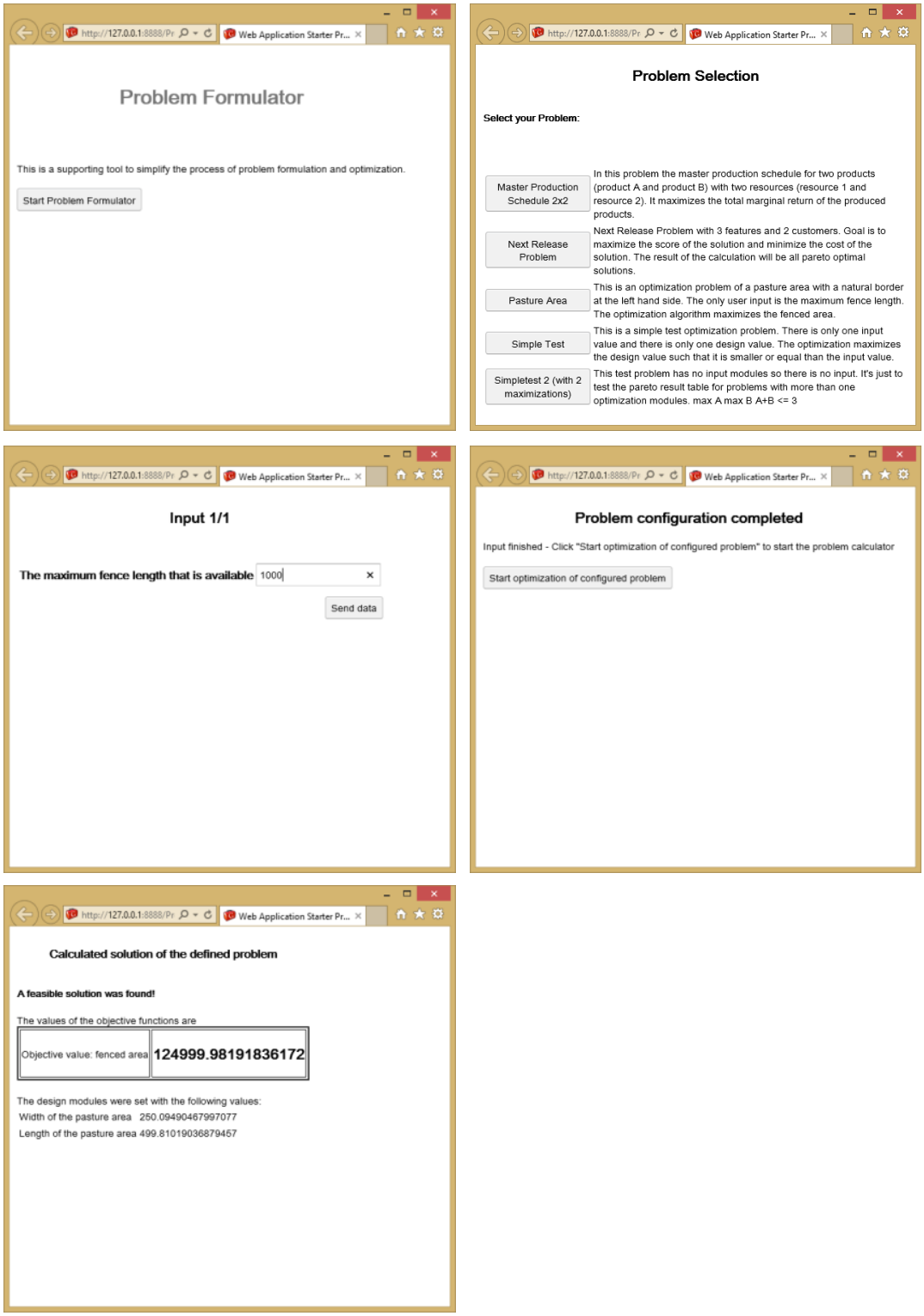


Abbildung 34: Screenshots des Problemformulierungsprozesses des Problem Formulators

Voraussetzung für diesen vereinfachten Problemformulierungsprozess ist jedoch, dass das gesuchte Problem in der Wissensdatenbank vorhanden ist. Das Vorhandensein jedes

Problems von Bedeutung in der Wissensdatenbank kann im Rahmen dieser Arbeit natürlich nicht erreicht werden. Deshalb soll generell die Umsetzbarkeit von Optimierungsproblemen dargelegt werden. Aufgrund des eingeschränkten Modulkatalogs müssen die Probleme ohne skalierbare Dimensionalität umgesetzt werden. Die Umsetzung von zwei ausgewählten Optimierungsproblemen aus Anwendungen rechnergestützter Entwurfsoptimierung wird im Folgenden exemplarisch durchgeführt.

Das erste umgesetzte Problem ist das in Abschnitt 4.1.1 eingeführte Next Release Problem. Der Modulbaum zu den verwendeten Modulinstanzen ist in Abbildung 35 dargestellt. Dieser wurde Analog zu Abschnitt 7.3 in der Wissensdatenbank modelliert.

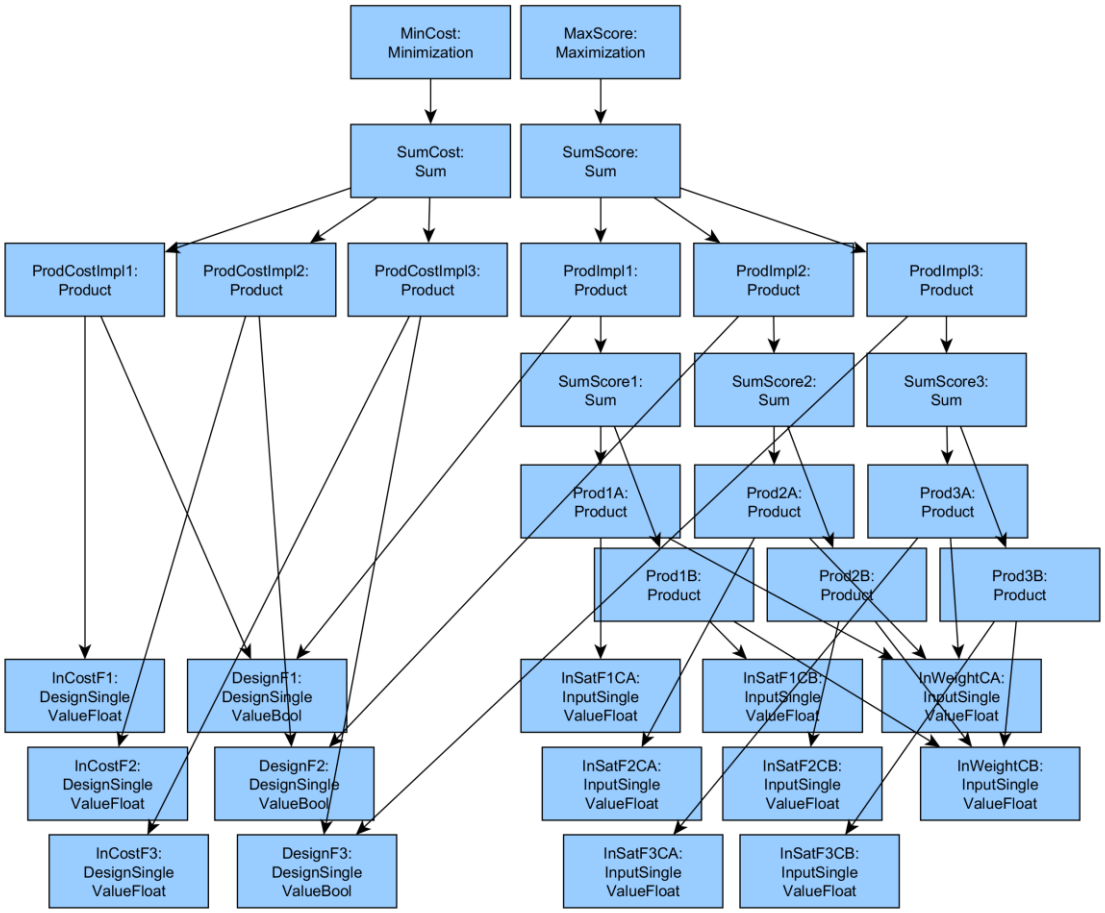


Abbildung 35: Modulbaum des modellierten Next Release Problems

Aufgrund der binären Designvariablen ist der Suchraum relativ klein, was dazu führt, dass eine Optimierung in 100 Generationen mit dem evolutionären Algorithmus (selbst bei einer Erhöhung der Zahl an Features) gut ausreicht.

Eine Beispiel-Problemformulierung hierzu wurde mit den folgenden Parametern durchgeführt:

- Kosten Feature 1 (Modul: InCostF1): 3,5
- Kosten Feature 2 (Modul: InCostF1): 5,0
- Kosten Feature 3 (Modul: InCostF1): 6,0
- Verbraucherbefriedigung von Feature 1 für Kunde A (Modul: InSatF1CA): 0,5
- Verbraucherbefriedigung von Feature 1 für Kunde B (Modul: InSatF1CB): 0,0
- Verbraucherbefriedigung von Feature 2 für Kunde A (Modul: InSatF2CA): 0,5
- Verbraucherbefriedigung von Feature 2 für Kunde B (Modul: InSatF2CB): 0,5
- Verbraucherbefriedigung von Feature 3 für Kunde A (Modul: InSatF3CA): 0,0
- Verbraucherbefriedigung von Feature 3 für Kunde B (Modul: InSatF3CB): 0,5
- Kundenrelevanz von Kunde A (InWeightCA): 1
- Kundenrelevanz von Kunde B (InWeightCB): 2

Als Ergebnis gibt der Problem Formulator die folgende Pareto-Optimalen-Lösungen zurück:

Objective value: Cost of the solution	Objective value: Score of the solution	Implement feature 3 in next release?	Implement feature 2 in next release?	Implement feature 1 in next release?
3.5	0.5	false	false	true
0.0	0.0	false	false	false
11.0	2.5	true	true	false
14.5	3.0	true	true	true
8.5	2.0	false	true	true
5.0	1.5	false	true	false

Tabelle 2: Pareto-Optimale Lösungen des Problem Formulators auf das Next Release Problem

Zu beachten ist, dass die Lösung nur Feature 3 umzusetzen von der Lösung nur Feature 2 umzusetzen dominiert wird (cost: 6 zu 5 score: 1 zu 1,5). Die Lösung Feature 1 und Feature 3 umzusetzen wird ebenfalls von gleich mehreren anderen Lösungen dominiert. Deshalb tauchen diese richtigerweise nicht in der Tabelle der Pareto-Optimalen Lösungen auf.

Es ist leicht zu erkennen, dass es keine Schwierigkeit darstellt, das Problem in größerer Dimensionalität analog umzusetzen. Das Ergebnis wäre nur ein stark vergrößerter Formelbaum und eine Vergrößerung des Suchraums (nur wenn die Anzahl der Features

vergrößert wird), weshalb die Parameter des Optimierungsalgorithmus angepasst werden sollten.

Das zweite umgesetzte Problem ist die in Abschnitt 4.1.3 eingeführte Produktionsprogrammplanungs-Optimierung. Das Optimierungsproblem wurde in der Dimensionalität von zwei Produkten und zwei Ressourcen umgesetzt. Auch hier ist eine Erhöhung der Dimensionalität in analoger Vorgehensweise möglich.

Der der Modellierung in OWL zugrundeliegende Modulbaum ist in Abbildung 36 abgebildet.

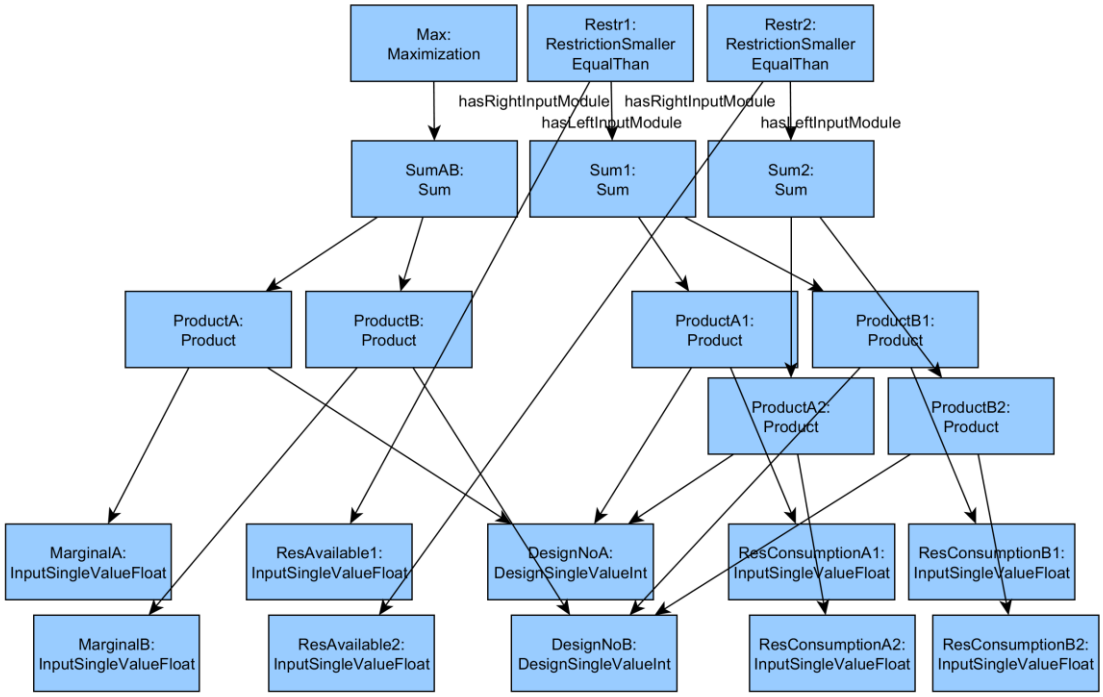


Abbildung 36: Modulbaum der modellierten Produktionsprogrammplanungs-Optimierung

Eine Beispiel-Problemformulierung hierzu wurde mit den folgenden Werten durchgeführt:

- Deckungsbeitrag für Produkt A (Modul: MarginalA): 12000
- Deckungsbeitrag für Produkt B (Modul: MarginalB): 10000
- Verfügbare Menge der Ressource 1 (Modul: ResAvailable1): 18
- Verfügbare Menge der Ressource 2 (Modul: ResAvailable2): 15
- Ressourcenverbrauch eines Produkts A an Ressource 1 (Modul: ResConsumptionA1): 5

- Ressourcenverbrauch eines Produkts B an Ressource 1 (Modul: ResConsumptionB1): 5
- Ressourcenverbrauch eines Produkts A an Ressource 2 (Modul: ResConsumptionA2): 5
- Ressourcenverbrauch eines Produkts B an Ressource 2 (Modul: ResConsumptionB2): 5

Als Ergebnis gibt der Problem Formulator die folgende optimale Lösung zurück:

- Produktionsmenge Produkt A: 2
- Produktionsmenge Produkt B: 2
- Gesamt-Deckungsbeitrag des gefundenen Produktionsprogramms: 44000

Da das modellierte Problem nur ein Optimierungsmodul besitzt, wird nur eine optimale Lösung angezeigt. Es kann zwar verschiedene Lösungen geben, die den gleichen Wert der Bewertungsfunktion haben, jedoch sind diese Lösungen nach der Problemdefinition als gleich gut zu bewerten und werden deshalb von Opt4J nicht separat aufgeführt.

Die Umsetzbarkeit dieser zwei Optimierungsprobleme zeigt bereits, dass mit dem bestehenden schmalen Modulkatalog eine gute Kapselung erzielt werden konnte und bereits ein breites Problemspektrum umsetzbar ist. Während die Problemformulierung mit dem Problem Formulator, bei Vorhandensein einer geeigneten Problemmodellierung in der Wissensdatenbank, eine massive Vereinfachung bringt, ist mit dem exemplarisch entwickelten Modulkatalog nach wie vor die Fähigkeit zur Umsetzung der Problemstellung in eine mathematische Formel vom Problemdesigner gefordert. Doch auch wenn der Gesamtprozess bei leerer Wissensdatenbank nicht an Komplexität eingebüßt hat, ist trotzdem ein Mehrwert geschaffen, da der Prozess nun dreigeteilt ist und die Schritte unabhängig voneinander ausgeführt werden können. So ist es möglich die Komplexität auf verschieden Nutzer bzw. Nutzergruppen zu verteilen.

Auch aus dem Problemdesign kann die Komplexität genommen werden, indem der Modulkatalog, wie in Abschnitt 7.1.3 beschrieben, weniger mit der Modellierungszielsetzung der Kapselung entwickelt wird. Dann lassen sich eng gefasste Module entwickeln, die zwar nur für ein kleines Anwendungsgebiet geeignet sind dort jedoch umfangreichere Aufgaben übernehmen können. Ein solcher Ansatz ist mit dem in Abbildung 37 dargestellten Modulbaum für das in Abschnitt 7.3.1 vorgestellte Modellie-

rungsproblem einer Weidefläche angedeutet. Mit solchen Modulen kann ein Problem in Baukastenform modelliert werden, ohne die Kenntnisse zur Umformung in eine mathematische Formulierung, zu besitzen. Eine Entwicklung solcher Module ist ohne Einschränkungen möglich, da das in Abschnitt 6.5 vorgestellte Konzept Module so allgemein fasst, dass diesbezüglich nahezu keine Einschränkungen bestehen. Ein Modul kann vom Umfang eines kleinen Operator-Moduls bis hin zu weit spezifischeren Modulen, mit denen sich umfangreiche Probleme aus nur einem Input-Modul und einem End-Modul modellieren lassen, alles darstellen.

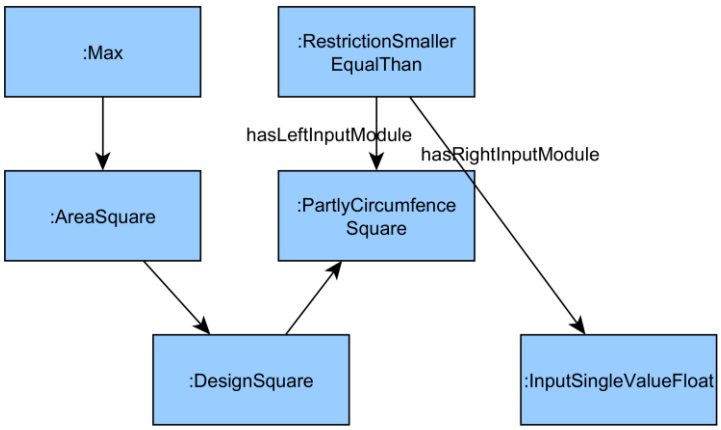


Abbildung 37: Modulbaum für das Weideflächen-Optimierungsproblem mit enger gefassten Modulen

Über den reinen Optimierungsprozess hinaus kann eine Modellierung in Form einer Ontologie auch noch weitere Vorteile bieten. Das Konzept der Ontologie eignet sich hervorragend um weitere Informationen an die Modellierung anzuhängen. Diese Form der Darstellung gibt z.B. die Möglichkeit umfangreiche Metadaten an das Problem und seine Module anzuhängen und erlaubt so eine weitere Facette der Unterstützung des Problemformulierungsprozesses analog zu der Arbeit [WKG07]. Solche Metainformation können z.B. domänenspezifische Zuständigkeiten, oder auch Kommentare zum modellierten Problem zur besseren Wartbarkeit desselben sein.

9 Fazit

Ziel dieser Arbeit ist die möglichst einfache Definition eines Optimierungsproblems über eine Webschnittstelle zu ermöglichen. Dieser Problemstellung wurde durch die Entwicklung des Problemformulierungstools Problem Formulator entgegengetreten. Dieses Expertensystem erlaubt, nach vorheriger Modellierung in Form von Ontologien, eine denkbar einfache Problemformulierung.

Den Hauptteil dieser Arbeit stellten der Entwurf und die Entwicklung des modularen Problemformulierungstools zum Problemdesign und der anschließenden Problemformulierung dar. Das entwickelte Framework erlaubt es neue Module zu entwickeln ohne den sonstigen Quellcode des Frameworks verändern zu müssen. Das Konzept des verwendeten Ansatzes wurde so allgemein gehalten, dass in der Modulentwicklung umfangreiche Freiheiten bleiben. Jedes denkbare Optimierungsproblem lässt sich, die Möglichkeit der Modulentwicklung miteinbezogen, mit diesem Tool umsetzen. Das Spektrum möglicher Module geht sogar noch weiter, so lassen sich, wie in Abschnitt 7.1.3 erörtert, verschiedene Modellierungszielsetzungen beim Entwurf eines Modulkatalogs verfolgen. Trotz dieser Offenheit in der Modulentwicklung konnte der Implementierungsaufwand für neue Module so klein wie möglich gehalten werden. Dies wird durch den objektorientierten Ansatz, die Module in baumartigen Ableitungsbeziehungen zu strukturieren, erreicht. Des Weiteren lassen sich so Quellcoderedundanzen im Modulcode weitgehend vermeiden.

Der Problemformulierungsprozess hängt ausschließlich von den in der Modellierung verwendeten Modulen und deren Verschachtelung ab. Dieser Prozess lässt sich also durch unterschiedliche Modellierungen und der Möglichkeit beliebige Module selbst zum Modulkatalog hinzuzufügen, nahezu beliebig gestalten. Der in dieser Arbeit erstellte Modulkatalog zielt zunächst stark auf das Modellierungsziel einer guten Kapselung ab, damit mit möglichst vielseitigen Modulen unterschiedlichste Optimierungsprobleme umgesetzt werden können. Dies wurde anhand von exemplarischen Umsetzungen von verschiedensten Optimierungsproblemen gezeigt.

10 Zusammenfassung und Ausblick

Diese Arbeit beginnt mit einer Einführung in das Thema und führt anschließend in relevante Hintergrundthemen ein. Hierbei spielen, dem Titel der Arbeit entsprechend, Themen rund um den industriellen Entwicklungsprozess eine Rolle. Auf technischer Seite werden Technologien vorgestellt, die im späteren Verlauf der Arbeit Anwendung finden. Mit der Aufgabenstellung lässt sich die Arbeit erstmals in die Hintergrundthemen einordnen.

In verwandten Arbeiten konnte eine Vielzahl von Anwendungen rechnergestützter Entwurfsoptimierungen recherchiert werden. Die Recherche nach Arbeiten, die sich mit der Problemformulierung selbst befassen, ergab hingegen weniger Ergebnisse. Die gefundenen Arbeiten beschränken sich auf Systeme, die durch das Anhängen von Metadaten einen kollaborativen Problemformulierungsprozess möglich machen und auf Ansätze, die versuchen das Optimierungsproblem effizient lösbar umzuformen um die Berechnungszeit zu verkürzen. Arbeiten, die den Ansatz verfolgen die mathematische Problemformulierung an sich zu vereinfachen, konnten keine gefunden werden. Deshalb stellte die Entwicklung eines geeigneten Ansatzes für die Arbeit bereits einen wesentlichen und zudem komplexen Teil dieser Arbeit dar.

Als Vorgehen zum Lösen der Aufgabenstellung wurde ein lineares Vorgehensmodell, das eine Reihe teils überlappender Phasen vorsieht, gewählt.

Das erstellte Problemformulierungstool wurde in der Form eines Wissenssystems erstellt. Die hierzu notwendige Wissensdatenbank wurde in Form von Ontologien aufgebaut. Die Ontologien wurden mit OWL modelliert und im XML-Format gespeichert. Basis aller Modellierungen in der Wissensdatenbank ist eine Schema-Ontologie, die unter anderem alle verwendbaren Modulklassen enthält. Die Modellierung weiterer Probleme sowie die Erstellung neuer Module wurde anhand von Beispielen beschrieben und so auch deren Unabhängigkeit vom Programmkern dargelegt.

Die Evaluierung wurde in Form einer Problemmodellierung von zwei praktisch relevanten Optimierungsproblemen durchgeführt. Die umgesetzten Optimierungsprobleme wurden mit einem zuvor implementierten Basis-Modulkatalog modelliert. Bei der Erstellung des Basismodulkatalogs wurde das Modellierungsziel einer guten Kapselung

verfolgt, wodurch sich bereits ein breites Spektrum an Modellierungsproblemen umsetzen lässt.

Trotz diesen erfolgreichen Umsetzungen von Optimierungsproblemen bietet sich eine große Vielfalt an Weiterentwicklungsmöglichkeiten des erstellten Problemformulierungstools. Ein erster Ansatz kann die Erweiterung der Wissensdatenbank in Form von neuen Optimierungsproblemumsetzungen sein.

Möglichkeiten für umfassendere aufbauende Arbeiten bietet dagegen die Erweiterung des Modulkatalogs. Ein erster Schritt hierbei muss die Auswahl einer geeigneten Modellierungszielsetzung sein. Die Auswahl muss stets unter Abwägung der der jeweiligen Zielsetzung zugrundeliegenden Zielkonflikte getroffen werden. In Abhängigkeit der getroffenen Entscheidung kann sich so eine weitere Verbesserung der Benutzbarkeit dieses Tools ergeben.

Da das Modellierungstool Protégé für die Problemformulierung eine zu allgemeine Modellierungsoberfläche besitzt, bietet es sich an, ein eigenes spezialisiertes Modellierungstool zu entwickeln. Kernfunktionalität dieses Tools sollte eine grafische Modellierbarkeit des Modulbaums sein.

Literaturverzeichnis

[AHEC97] Alexandrov N.M.; Hussaini Y.; Engineering I.C.A.S. et al. (1997): *Multidisciplinary Design Optimization: State of the Art*. Society for Industrial and Applied Mathematics, ISBN: 9780898713596 LCCN: 96072047.

[And00] Andersson J. (2000): A survey of multiobjective optimization in engineering design. *Department of Mechanical Engineering, Linktjping University. Sweden*.

[Apa14] (2014): *Apache Jena*. The Apache Software Foundation. https://jena.apache.org/getting_started/, zuletzt geprüft am 09.Oktober.2014.

[Aut12] Automotive Simulation Center Stuttgart (2012): *Aktuelles aus der Wissenschaft zum Thema Simulation*. Automotive Simulation Center Stuttgart. http://www.kunde.arh-design.de/ascs/newsletter/2012-08/simulation.htm, zuletzt geprüft am 09.Oktober.2014.

[BFGP13] Beitz W.; Feldhusen J.; Grote K. et al. (2013): *Pahl/Beitz Konstruktionslehre: Grundlagen erfolgreicher Produktentwicklung. Methoden und Anwendung*. 8. Auflage. Aufl. Heidelberg: Springer Vieweg.

[BHM04] Booth D.; Haas H.; McCabe F. et al. (2004): *Web Services Architecture*. World Wide Web Consortium. http://www.w3.org/TR/ws-arch/, zuletzt geprüft am 39.September.2014.

[BSW12] Burr W.; Stephan M. und Werkmeister C. (2012): *Unternehmensführung*. Vahlen, ISBN: 9783800639397.

[BvHH04] Bechhofer S.; van Hermelen F.; Hendler J. et al. (2004): *OWL Web Ontology Language*. W3C. http://www.w3.org/TR/owl-ref, zuletzt geprüft am 09.Oktober.2014.

[CBH+11] Cao K.; Batty M.; Huang B. et al. (2011): Spatial multi-objective land use optimization: extensions to the non-dominated sorting genetic algorithm-II. *International Journal of Geographical Information Science*, 25, 12, 1949-1969.

[Cla06] Claus P.D.V. und Schwill P.D.A. (2006): *Duden Informatik*. Mannheim: Dudenverlag.

[Don14] Computer Science University of Maryland: *Donald Norman*. Computer Science University of Maryland. http://www.cs.umd.edu/hcil/muiseum/norman/norman_page.htm, zuletzt geprüft am 09.Oktober.2014.

[EKGW11] Eddy D.; Krishnamurty S.; Grosse I. et al. (2011): Support of product innovation with a modular framework for knowledge management: a case study. In: *ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*.. S. 1223-1235. Zitiert Witherhel.

[Fei13] Feigenbaum L. und Prud'hommeaux E. (2013): *SPARQL by Example*. Cambridge Semantics. http://www.cambridgesemantics.com/de/semantic-university/sparql-by-example, zuletzt geprüft am 09.Oktober.2014.

[GO94] Gruber T.R. und Olsen G.R. (1994): An Ontology for Engineering Mathematics. *KR*, 94, 258-269.

[Goo14] (2014): *Java Runtime Environment*. Google. https://developers.google.com/appengine/docs/java/#Java_The_sandbox, zuletzt geprüft am 09.Oktober.2014.

[Goo14a] Google Inc.: *GWT Project*. http://www.gwtproject.org/overview.html, zuletzt geprüft am 09.Oktober.2014.

[GR04] Greer D. und Ruhe G. (2004): Software release planning: an evolutionary and iterative approach. *Information and Software Technology*, 46, 4, 243-253.

[Gru93] Gruber T.R. (1993): A translation approach to portable ontology specifications. *Knowledge acquisition*, 5, 2, 199-220.

[gua14] (2014): *guava-libraries*. Google Project Hosting. https://code.google.com/p/guava-libraries/, zuletzt geprüft am 09.Oktober.2014.

[HK06] Homburg C. und Krohmer H. (2012): Grundlagen des Marketingmanagements. *Einfhrung in Strategie, Instrumente, Umsetzung und Unternehmensfhrung, Wiesbaden*, 3.

[HKRS08] Hitzler P.; Krotzsch M.; Rudolph S. et al. (2008): *Semantic Web: Grundlagen*. Spring-Verlag.

[HMRSZ11] Haslhofer B.; Momeni Roochi E.; Schandl B. et al. (2011): Europeana rdf store report.

[LGRT11] Lukasiewycz M.; Glaß M.; Reimann F. et al. (2011):

[Mo65] Moore G.E. und others (1965): *Cramming more components onto integrated circuits*. McGraw-Hill New York, NY, USA.

[NSD01] Noy N.F.; Sintek M.; Decker S. et al. (2001): Creating semantic web contents with protege-2000. *IEEE intelligent systems*, 16, 2, 60-71.

[Pau11] Paulheim H. (2011): *TU Darmstadt*. https://www.ke.tu-darmstadt.de/lehre/archiv/ws-11-12/./folien-teil-13, zuletzt geprüft am 09.Oktober.2014.

[PwC13] (2013): *PwC Strategy& Pressemitteilungen*. PwC Strategy&. http://www.strategyand.pwc.com/de/home/Presse/Pressemitteilungen/details/2013-global-innovation-1000-de, zuletzt geprüft am 09.Oktober.2014.

[RGKW09] Rockwell J.; Grosse I.R.; Krishnamurty S. et al. (2009): A Decision Support Ontology for collaborative decision making in engineering design. In: *Collaborative Technologies and Systems, 2009. CTS'09. International Symposium on*.. S. 1-9. Zitiert Witherel.

[SF09] Stepan A. und Fischer E.O. (2009): *Betriebswirtschaftliche Optimierung: Einführung in die quantitative Betriebswirtschaftslehre*. Oldenbourg, ISBN: 9783486587814.

[Shn03] Shneiderman B. (2003): *Leonardo's laptop: human needs and the new computing technologies*. Mit Press.

[sou14] sourceforge.net (2014): *A Modular Framework for Meta-heuristic Optimization - Tutorial*. sourceforge.net. http://opt4j.sourceforge.net/documentation/3.1.2/tutorial.xhtml, zuletzt geprüft am 09.Oktober.2014.

[TER09] Tosserams S.; Etman L.P. und Rooda J. (2009): A classification of methods for distributed system optimization based on formulation structure. *Structural and Multidisciplinary Optimization*, 39, 5, 503-517.

[VDI03] VDI (2003): 2211 (2003): Informationsverarbeitung in der Produktentwicklung Berechnungen in der Konstruktion. *VDI-Verlag, Düsseldorf*.

[VDI93] VDI (1993): 2221 (1993): Methodik zum Entwickeln und Konstruieren technischer Systeme und Produkte. *VDI-Verlag, Düsseldorf*.

[WCL+05] Weerawarana S.; Curbera F.; Leymann F. et al. (2005): *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Upper Saddle River, NJ, USA: Prentice Hall PTR, ISBN: 0131488740.

[WKG07] Witherell P.; Krishnamurty S. und Grosse I.R. (2007): Ontologies for supporting engineering design optimization. *Journal of Computing and Information Science in Engineering*, 7, 2, 141-150.

[WS07] Wang G.G. und Shan S. (2007): Review of metamodeling techniques in support of engineering design optimization. *Journal of Mechanical Design*, 129, 4, 370-380.

[Yeg09] Y. und Bonfanti V. (2009): *mail-archive*. https://www.mail-archive.com/google-appengine-java@googlegroups.com/msg00983.html, zuletzt geprüft am 09.Oktober.2014.

[ZHM07] Zhang Y.; Harman M. und Mansouri S.A. (2007): The multi-objective next release problem. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*.. S. 1129-1137.

Abbildungsverzeichnis

Verzeichnis für Code-Listings

Formelverzeichnis

Tabellenverzeichnis

Stichwortverzeichnis

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.

Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.

Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.

Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.

Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

(Ort, Datum, Unterschrift)