|  |
| --- |
| ,,HTWG Konstanz – Fakultät Informatik |
| 2015-01-21 Softwaredokumentation JobShopNeu |
|  |

|  |
| --- |
| Dennis Klein, Wirtschaftsinformatik 7.Semester  Sebastian Stephan, Wirtschaftsinformatik 7.Semester |

Inhaltsverzeichnis

[Abbildungsverzeichnis ii](#_Toc409349997)

[1 Einführung und Ziele 1](#_Toc409349998)

[2 Randbedingungen 3](#_Toc409349999)

[2.1 Technische Einflussfaktoren 3](#_Toc409350000)

[2.2 Organisatorische Einflussfaktoren 3](#_Toc409350001)

[3 Projekt-Ist-Zustand bei Übernahme (22.10.2014) 4](#_Toc409350002)

[4 Probleme 5](#_Toc409350003)

[4.1 Lösungsansätze 5](#_Toc409350004)

[5 Kontextabgrenzung 6](#_Toc409350005)

[6 Softwarearchitektur 7](#_Toc409350006)

[6.1 Klassendiagramm: JobShopApp 7](#_Toc409350007)

[6.2 Klassendiagramm: JobShopView 8](#_Toc409350008)

[6.3 Klassendiagramm: JobShopCalculator 9](#_Toc409350009)

[6.4 Klassendiagramm: JobShopNumberFormatter 10](#_Toc409350010)

[6.5 Klassendiagramm: JobShopFileFilter 10](#_Toc409350011)

[6.6 Klassendiagramm: JobShopAboutBox.Java 11](#_Toc409350012)

[6.7 Klassendiagramm: JobShopResultBox 12](#_Toc409350013)

[6.8 Klassendiagramm: ResultList 13](#_Toc409350014)

[6.9 Klassendiagramm: Result 13](#_Toc409350015)

[7 Softwarequalität 14](#_Toc409350016)

[8 Begründung von Entwurfsentscheidungen 15](#_Toc409350017)

[8.1 Problemerörterung 16](#_Toc409350018)

[8.2 Lösungsansatz 16](#_Toc409350019)

[8.3 Vorgehensweise zur Erstellung der 64 Bit-Applikation: 17](#_Toc409350020)

[9 Weiterentwicklung JobShopNeu 19](#_Toc409350021)

[10 Beispielaufgabe 22](#_Toc409350022)

[10.1 Lösung traditionell mit lpSolve: 23](#_Toc409350023)

[10.2 Lösung mit JobShopNeu: 23](#_Toc409350024)

# Abbildungsverzeichnis

Abbildung 1: Klassendiagramm (22.10.2014) 4

Abbildung 2: Kontextdiagramm 6

Abbildung 3: Klassendiagramm JobShopApp 7

Abbildung 4: Klassendiagramm JobShopView 8

Abbildung 5: Klassendiagramm JobShopCalculator 9

Abbildung 6: Klassendiagramm JobShopNumberFormatter 10

Abbildung 7: Klassendiagramm JobShopFileFilter 10

Abbildung 8: Klassendiagramm JobShopAboutBox 11

Abbildung 9: Klassendiagramm JobShopResultBox 12

Abbildung 10: Ergebnisausgabe 12

Abbildung 11: Klassendiagramm ResultList 13

Abbildung 12: Klassendiagramm Result 13

Abbildung 13: DLL in Projekt importieren 17

Abbildung 14: Runnable JAR File Export 17

Abbildung 15: Projektinhalt 18

Abbildung 16: Aufbau Start\_Job\_Shop\_64Bit.bat-Datei 18

Abbildung 17: Programmaufruf 18

Abbildung 18: Eingabematrix 16/16 19

Abbildung 19: Methode initComponents() 19

Abbildung 20: Popup Beispiel bei Falscheingabe 21

Abbildung 21: Methode stringToValue(String s) 21

Abbildung 22: Eingabematrix JobShopNeu 22

Abbildung 23: Lösungsmatrix 22

Abbildung 24: FileWriter-Erweiterung 22

Abbildung 25: Lösungsvektor lpSolve 23

Abbildung 26: Eingabematrix JobShopNeu 24

Abbildung 27: Lösungsmatrix JobShopNeu 24

# Einführung und Ziele

Im Rahmen der betrieblichen Systemforschung pflegt die Hochschule für Technik, Wirtschaft und Gestaltung (HTWG) seit vielen Jahren eine Methodenbank. In der Methodenbank sind verschiedene Programme für die Lösung unterschiedlicher Problemstellungen der linearen Programmierung (LP) enthalten. Die verschiedenen Software-Anwendungen greifen dabei auf unterschiedliche Solver zur Lösung der erstellten LP-Modelle zurück. Die Methodenbank umfasst sowohl Open-Source-Solver (lpSolve, MOPS) als auch kommerzielle Solver-Produkte (IBM ILOG CPLEX).

Entwickelt und auch eingesetzt wird diese Methodenbank primär in der Vorlesung „Operations Research“. Das Ziel der Weiterentwicklung und Sicherstellung der Lauf- und Funktionsfähigkeit der einzelnen Methoden und Programmen und der Methodenbank selbst ist Gegenstand der Veranstaltungen „Teamprojekt“ und „ALO“ in den höheren Wirtschaftsinformatik-Semestern.

Bei JobShop handelt es sich um ein Programm dessen Aufgabe darin besteht, eine vorgegebene Zahl von Produkten hinsichtlich einer optimalen, kürzesten Gesamtdurchlaufzeit auf verschiedene Maschinen einzuplanen. Derzeit befindet sich sowohl JobShop als auch JobShopNeu[[1]](#footnote-1) weder in der Methodenbank OR\_ALPHA noch in ORWEB da die Programme trotz vollständiger Installation unter Windows 7 64-Bit nicht gestartet werden können. Gemäß dem Vorgängerprojekt lässt sich JobShopNeu jedoch ab und zu unter Windows 7 32-Bit starten, was für die Hochschule jedoch nicht ausreichend ist.

Primärziel des Projekts ist es, JobShopNeu auf Windows 7 SP1 (32-/64-Bit-Architektur) lauffähig zu bekommen sowie ggf. die „umfangreiche“ Installation des Programms zu verbessern. Ist die Software lauffähig, soll das System dem Benutzer die Möglichkeit bieten, neue Maschinen und Produkte anzulegen und die entsprechenden Durchlaufzeiten zu hinterlegen. Es soll die Möglichkeit bestehen, diese sowie gespeicherte LP-Modelle abzuspeichern bzw. zu laden (z.B. XML- oder TXT-Datei). Über eine Schnittstelle zu einem Solver, primär dem lpSolve, soll ein generiertes LP-Modell gelöst werden. Die Einbindung weiterer Solver ist durchaus denkbar (z.B. IBM ILOG CPLEX). Das Ergebnis wird dann dem Benutzer in einem neuen Fenster textuell oder graphisch angezeigt. Durch die Angabe eines Dateiverzeichnisses soll das Ergebnis als XML-, TXT-, CSV- oder ähnlichem Dateiformat dauerhaft gesichert werden können. Außerdem wird eine Eingabeüberprüfung insb. bei der Eingabe der Durchlaufzeiten angestrebt. Dadurch sollen Falscheingaben vorgebeugt und die Software benutzerfreundlicher gestaltet werden.

Parallel soll versucht werden, JobShop, die ältere Version von JobShopNeu, auf Windows 7 SP1 (32-/64-Bit-Architektur) oder über eine virtuelle Maschine wie z.B. DOSBox lauffähig zu bekommen, sodass den Studierenden übergangsweise ein Softwaretool für Optimierungsprobleme dieser Art zur Verfügung steht. Eine Weiterentwicklung dieses Tools ist nicht Bestandteil dieses Projekts.

Welche Anforderung in welchem Umfang und Detail erbracht werden sollen, wurde zusammen mit Prof. Dr. Grütz und dem Projektteam mündlich besprochen und in einem Pflichtenheft festgehalten. Für die Bewertung des Projekterfolgs sowie die Benotung soll zusätzlich auf dieses Dokument verwiesen werden. Die in diesem Dokument erwähnten Funktionalitäten dienen der langfristigen Orientierung hinsichtlich der Weiterentwicklung von JobShopNeu bis hin zur Einbindung in OR\_ALPHA und ggf. ORWEB.

Ziel dieser Softwaredokumentation ist es, den aktuellen IST-Zustand des JobShopNeu zu erfassen. Da keine Softwaredokumentation von JobShop als auch JobShopNeu vorhanden war, soll der aktuelle IST-Zustand des Systems, das Zusammenspiel der einzelnen Klassen (und Komponenten) sowie die aktuellen Probleme analysiert und festgehalten werden. Auf dieser Basis sollen dann die Weiterentwicklungsmaßnahmen (Entwurfsentscheidungen) getroffen werden.

# Randbedingungen

Im Folgenden werden sämtliche (externe) Faktoren genannt, welche die (neue) Architektur beeinflussen oder einschränken können. Auf Basis dieser Faktoren sollen schließlich unter anderem alle anwendungs- und problembezogenen Entwurfsentscheidungen begründet werden.

## Technische Einflussfaktoren

**Betriebssystem:** Windows 7 SP1 (32-/64-Bit-Architektur)

**Programmiersprache:** Java

**Entwicklungsumgebung:** Eclipse Juno JDK1.8

**Komponenten/Bibliotheken:**

1. Sämtliche Java AWT-/Swing-Komponenten:
   1. AbsoluteLayout.jar
   2. appframework-1.0.3.jar
   3. swing-worker-1.1.jar
2. Schnittstelle für Solver
   1. lpsolve55j.jar

## Organisatorische Einflussfaktoren

**Vorgehensmodell:** Iteratives und inkrementelles Vorgehen, wöchentliche Teambesprechungen sowie Projektbesprechungen zusammen mit Prof. Dr. Grütz

**Projektzeitraum:** 22.10.2014 – 21.01.2015

**Personal:** 2 Java-Entwickler

**Qualitätsstandards:** Sollen der Messung der Softwarequalität dienen:

1. ISO 9126 bzw. ISO 25010 und
2. Oracle-Code-Convention

# Projekt-Ist-Zustand bei Übernahme (22.10.2014)

Das JobShopNeu-Javaprogramm wurde in der Entwicklungsumgebung „Netbeans IDE 6.8“ programmiert. Als Solver wurde lp\_solve 5.5 verwendet. Entwickelt und getestet wurde die Software unter Windows, wobei die Lauffähigkeit nicht dauerhaft gewährleitet werden konnte.

Momentan besteht die Software aus 10 Klassen.[[2]](#footnote-2)

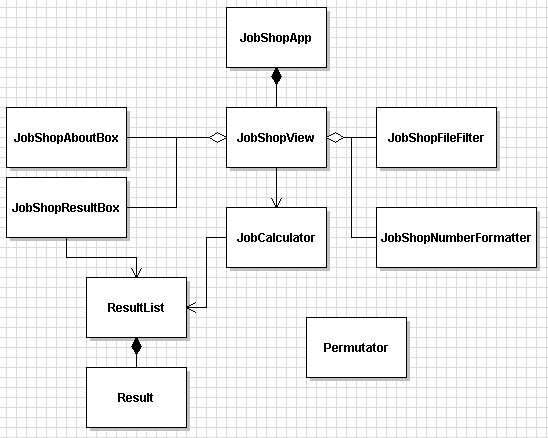


Abbildung : Klassendiagramm (22.10.2014)

Laut der „Dokumentation“ des Vorgängerprojekts lässt sich die Software potentiell auf einer 32-Bit-Architekur starten sodass sich der Programmablauf wie folgt beschreiben lassen kann:[[3]](#footnote-3)

Im Hauptfenster des JobShops (JobShopView) können die Bearbeitungsdauern der Produkte für jede Maschine eingetragen werden. Dabei kontrolliert der JobShopNumberFormatter, dass es sich bei den Einträgen um Zahlen zwischen 0 und 99 handelt.

Beim Klick auf den Berechnen-Button wird der JobCalculator aufgerufen. Der JobCalculator erstellt dann das LP-Modell und berechnet ein optimales Ergebnis mit dem lpSolve. Dieses Ergebnis speichert er dann in der ResultList ab, in welcher alle Punkte auf welcher Maschine welches Produkt an welchem Zeitpunkt gebaut wird (Result), gespeichert werden. Die ResultList wird dann an die JobShopResultbox übergeben, welche dann das Ergebnis anzeigt.

Der JobShopFileFilter kontrolliert, dass nur (Text-)Dateien gespeichert/geladen werden, deren Bezeichnung mit „.job“ endet.

Die JobShopAboutBox zeigt ein Fenster an in welchem Informationen wie Autor und Erstellungsdatum angegeben sind.

Die Permutator-Klasse ist ein erster Entwurf um das Job-Shop-Problem so zu lösen, wie in der alten JobShop-Version.

Aus diesem Grund werden zunächst die Klassen sowie ihre Attribute näher beschrieben und deren Beziehung zueinander.

# Probleme

Da dem Projektteam während des Projektzeitraums kein Rechner bzw. eine Virtual Machine (VM) mit Windows 7 SP1 32-Bit-Architektur zur Verfügung stand, konnte sich der soeben geschilderte Programmablauf zunächst nicht validieren lassen. Erst im späteren Verlauf konnte eine passende VM organisiert werden. Auf dieser wurde versucht den geschilderten Programmablauf zu validieren, jedoch vergebens.

Abbildung 2 skizziert grob wie die einzelnen Klassen miteinander in Beziehung stehen. Jedoch über die Funktionsweise sowie der Softwarequalität sagt dieses Klassendiagramm sowie die mitgelieferte Dokumentation nur wenig aus.

Die daraus resultierenden Folgen für das Projektteam waren

* eine aufwändige Codeanalyse,
* Bibliotheksabhängigkeiten aufgrund der zuvor verwendeten Entwicklungsplattform Netbeans IDE 6.8 sowie
* die Identifizierung möglicher Ansatzpunkte für Verbesserungen bzw. Erweiterungen.

## Lösungsansätze

Es wurden zu Beginn des Projekts drei mögliche Ansätze hinsichtlich der Ausführbarkeit des Programms diskutiert:

1. Dynamische Nutzung eines bzw. mehrerer externer Solver (lpSolve, IBM ILOG CPLEX, MOPS, …) über entsprechende Schnittstellen[[4]](#footnote-4)

* Der Programmcode nutzte bereits die Schnittstelle des lpSolve, jedoch war eine Ausführung auf Grund einiger Exceptions hinsichtlich des lpSolve nicht möglich. Daraufhin wurde eine aufwändige Fehleranalyse betrieben.

1. Statische Einbindung des lpSolve in das Softwareprojekt

* Eine weitere Möglichkeit war die statische Einbindung des lpSolve gewesen, dadurch müsste dieser jedoch fest ins Projekt eingebunden werden, ein fester Pfad gesetzt werden und der Code umgeschrieben werden.

1. Einbindung von Google’s linearen Programmierungssystems “GLOP”

* Der Google GLOP Solver kam insofern in Frage, dass dieser numerisch sehr stabil und schnell in der Berechnung ist und ressourcensparend arbeitet. Die Einbindung erfordert jedoch einen hohen Mehraufwand, da der Code komplett auf den neuen Solver umgeschrieben bzw. ergänzt werden müsste für die Erstellung eines Solver konformes LP-Modells.

# Kontextabgrenzung

Beschreibung des Systems aus Black-Box-Sicht. Sämtliche Interaktion von außen mit dem System sowie einzelner System-Komponenten soll übersichtlich dargestellt werden.

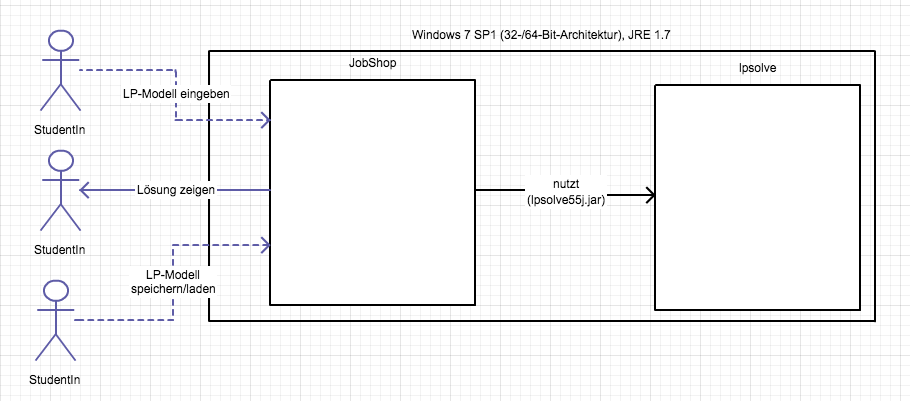


Abbildung : Kontextdiagramm

# Softwarearchitektur

Im folgendem wird die Ist-Architektur in Form von Klassendiagrammen sowie einer kurzen Beschreibung dieser festgehalten. Der Hintergrund dieses Projektschritts war wie bereits erwähnt die unzureichende Softwaredokumentation des Entwicklerteams.

## Klassendiagramm: JobShopApp

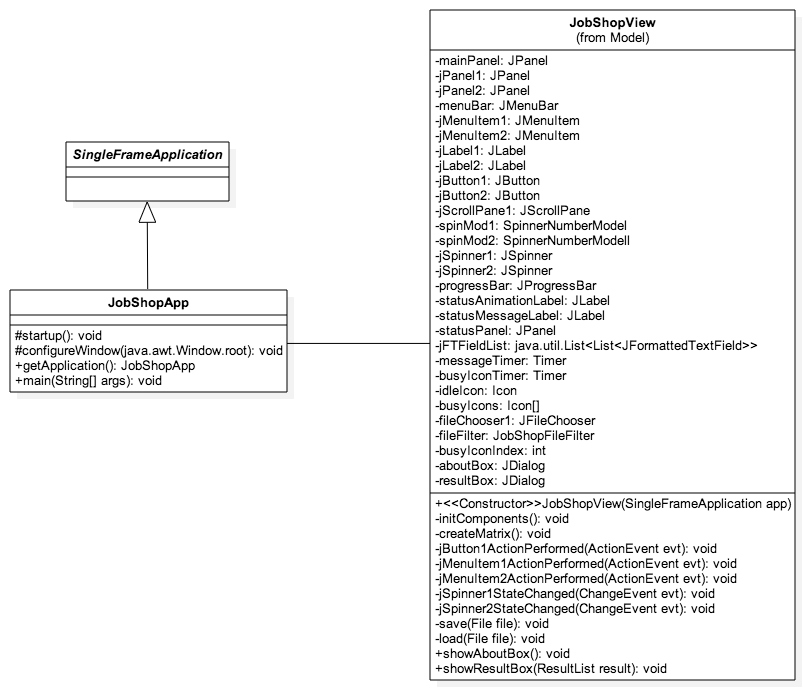


Abbildung : Klassendiagramm JobShopApp

Diese Klasse dient der Instanziierung und Verwaltung der GUI. Durch die Verwendung des Singleton-Patterns soll sichergestellt werden, dass von einer Klasse nur ein Objekt (Instanz) existiert. Dadurch soll eine bessere Zugriffskontrolle bzw. eine bessere Wartbarkeit der einzelnen Objekte gewährleistet werden. JobShopApp erbt dabei von der abstrakten Klasse *SingleFrameApplication*, die Methoden zur Verwaltung (WindowListener, Daten laden/speichern, Anwendung öffnen/schließen usw.) der instanziierten Session bereitstellt. Über die Methode *launch()* wird die Klasse instanziiert. Diese ruft die Methode *getApplication()* auf, die eine bestehende Instanz zurückgibt bzw. falls noch keine existiert, eine Application-Instanz zurückliefert. Über *startup()* wird die GUI erzeugt und angezeigt. Die Erstellung und Abbildung der Eingabelogik für das LP-Modell wurde in eine weitere Klasse ausgelagert, der JobShopView-Klasse.

## Klassendiagramm: JobShopView

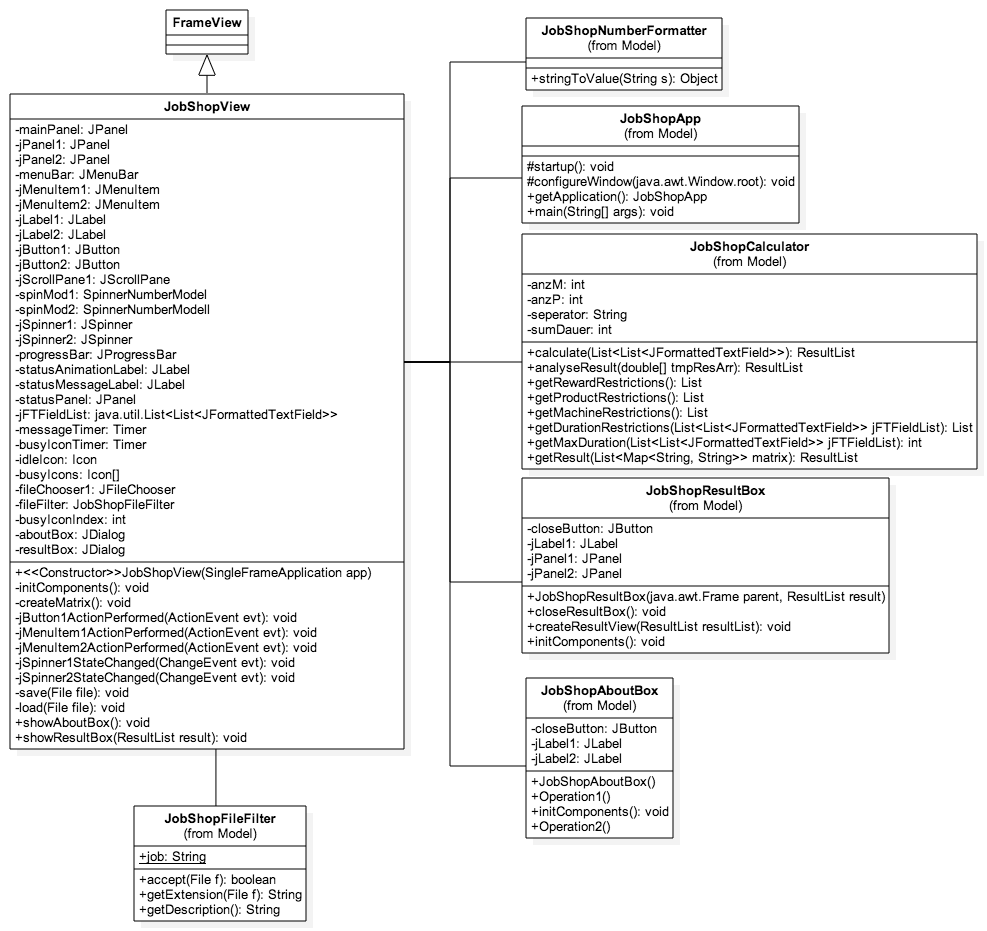


Abbildung : Klassendiagramm JobShopView

In dieser Klasse sind sämtliche Attribute und Methoden für die Erzeugung der GUI implementiert. Die Methode *initComponents()* kümmert sich um die Initialisierung der entsprechenden GUI-Attribute wie der Menüleiste(*JMenu()*) oder der Eingabematrix (*createMatrix()*), die dem Benutzer angezeigt werden soll. In einer ResourceMap werden die Namen und Bezeichnungen der jeweiligen Swing-Komponenten verwaltet und gesetzt. Diese ResourceMap befindet sich in einem separaten Ordner mit den entsprechenden Properties-Dateien. Bei einer Java Properties-Datei handelt es sich um eine einfache Textdatei, die in Java als einfache Konfigurationsdatei verwendet wird. Des Weiteren werden noch passende EventListener implementiert um zum Beispiel die Maschinen- bzw. Produktanzahl der Eingabematrix zur Laufzeit anzupassen. Diese Triggern dann die entsprechenden Funktionen wie z.B. speichern und laden der Eingabematrix, Erstellung des LP-Modells und dessen Berechnung.

## Klassendiagramm: JobShopCalculator

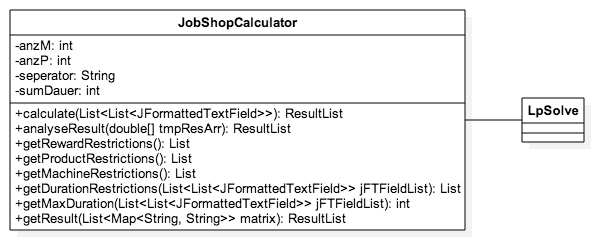


Abbildung : Klassendiagramm JobShopCalculator

Die Klasse JobShopCalculator überführt nun die Benutzereingabe in ein LP-Modell. Dabei werden folgende Restriktionen durch entsprechende Methodenaufrufe erstellt:

* *getDurationRestriction(List<List<JFormattedTextField>>jFTFieldList)*:Jedes Produkt muss innerhalb der Gesamtdurchlaufzeit abgearbeitet werden. Jedes Produkt hat so viel Variablen wie die Gesamtdurchlaufzeit beträgt. Natürlich wird aufgrund zeitlicher Grenzen des Produkts nicht der gesamte Variablenbereich benutzt.
* *getMachineRestriction()*:  
  Jede Maschine kann pro Zeiteinheit immer nur 0 oder 1 Produkt bearbeiten.
* *getMaxDuration()*:  
  Ermittelt die Gesamtdurchlaufzeit, in welcher die Produkte abgearbeitet werden müssen und gibt diese als Summe zurück. Diese ist notwendig um die Gesamtanzahl der benötigten Strukturvariablen als auch der benötigten Restriktionen zu ermitteln.
* *getProductRestriction()*:  
  Ein Produkt kann pro Zeiteinheit immer nur von 1 Maschine bearbeitet werden.
* *getRewardRestriction()*:  
  Die Produkte sollen möglichst an einem Stückbearbeitet werden. Dadurch soll ein ständiger Wechsel auf den Maschinen in Hinblick auf potenziell anfallende Rüstkosten vermieden werden.

Anschließend wird das zu lösende LP-Modell aus den einzelnen Restriktionslisten durch *calculate(List<JFormattedTextField>> jFTFieldList)* zusammengebaut, an den Solver übergeben, gelöst und das Ergebnis als Result zurückgegeben.

## Klassendiagramm: JobShopNumberFormatter

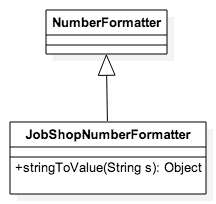


Abbildung : Klassendiagramm JobShopNumberFormatter

In dieser Klasse werden die Eingaben des Benutzers in der Eingabematrix entsprechend den Anforderungen, dass die Durchlaufzeiten nur ganzzahlig sein dürfen sowie die Durchlaufzeit nicht 99 Zeiteinheiten überschreiten darf, validiert.

## Klassendiagramm: JobShopFileFilter

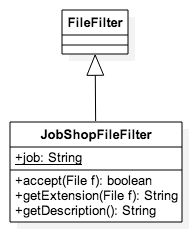


Abbildung : Klassendiagramm JobShopFileFilter

Klasse, die für das Erstellen und Verarbeiten von job-Dateien zuständig ist. Ermöglicht das Speichern der Eingabematrix automatisch als job-Datei. Im Prinzip handelt es sich dabei um eine einfache txt-Datei. Des Weiteren wird beim Laden überprüft, dass nur job-Dateien in die Anwendung geladen werden können.

## Klassendiagramm: JobShopAboutBox

Die Klasse JobShopAboutBox zeigt grundlegende Metainformationen über die JobShopNeu Applikation:



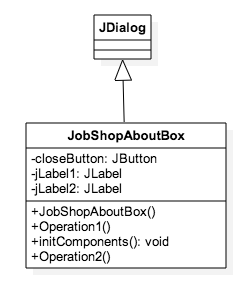
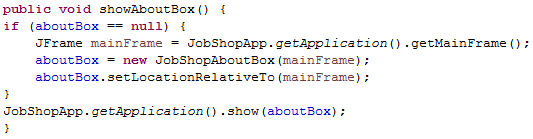


Abbildung : Klassendiagramm JobShopAboutBox

Die Klasse JobShopView ruft die Methode *showAboutBox()* auf, woraufhin sofern noch nicht vorhanden, ein Frame erstellt wird und der Klasse JobShopAboutBox als Parameter übergeben wird. Andernfalls wird die AboutBox direkt angezeigt.



Die Klasse JobShopAboutBox selbst erbt von der Klasse *javax.swing.JDialog*. Der *JDialog* ähnelt sehr stark einem *JFrame* mit der Ausnahme, dass der *JDialog* modal gesetzt werden kann. Modal bedeutet, dass, während der betreffende *JDialog* angezeigt wird, kein anderes Fenster verwendbar bzw. aktivierbar ist. Über die Methode *initComponents()* werden verschiedene JLabel sowie ein JButton initialisert. Der JButton dient zum Schließen des Fensters, die JLabel für die Angaben zur Beschreibung, der Version, des Autors, sowie dem Stand, was in obiger Abbildung sehr gut zu erkennen ist. Mit Hilfe des *dispose* Befehls, werden die zugeordneten Windows-Ressourcen wieder freigegeben.

## Klassendiagramm: JobShopResultBox

Nachdem das gefundene Optimum berechnet wurde, muss dieses anschließend dargestellt werden. Die Klasse JobShopResultBox erbt daher von der Klasse javax.swing.JDialog und initialisiert alle nötigen Komponenten um das Ergebnis, wie in nachfolgender Abbildung darzustellen:

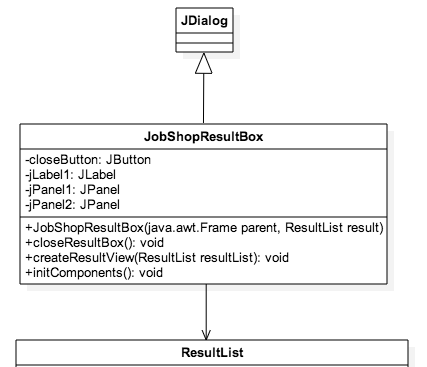


Abbildung : Klassendiagramm JobShopResultBox

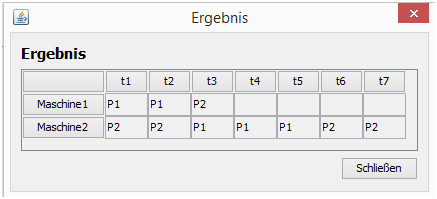


Abbildung : Ergebnisausgabe

## Klassendiagramm: ResultList

Die Klasse ResultList deklariert eine List vom Typ Result sowie die Variablen anzM (Anzahl Maschinen), anzP (Anzahl Produkte) und maxDauer (maximale Dauer), welche zur Sammlung gefundener Ergebnisse nötig sind.

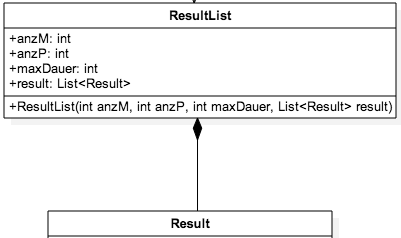


Abbildung : Klassendiagramm ResultList

## Klassendiagramm: Result

Um ein gefundenes Ergebnis darstellen zu können, werden in der Klasse Result die Integer Variablen zeitpunkt, maschine und produkt deklariert.

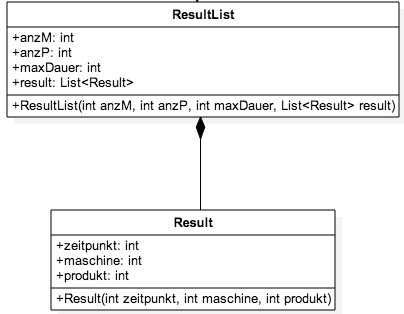


Abbildung : Klassendiagramm Result

# Softwarequalität

Da keine Systemspezifikation vorhanden war, wurde anhand der internationalen Norm für System und Software-Engineering – Qualitätskriterien und Bewertung von System und Softwareprodukten (ISO 25010) versucht die Software hinsichtlich folgender Qualitätskriterien zu beurteilen:

* Functional suitability  
  Zu Beginn konnte sich nur wenig über die funktionale Vollständigkeit sagen, da sich die Software nicht hat ausführen lassen. Jedoch zeichnete sich durch den Modellierungsprozess der einzelnen Klassen ab, dass ein fachlich korrektes LP-Modell gemäß den Anforderungen, die im Commitment spezifiziert wurden, erstellt wird (Konformität). Die Erstellung, Berechnung und Ergebnisanzeigen wurden in entsprechende Methoden ausgelagert. Durch die Verwendung der von lpSolve spezifizierten Klassenbibliothek lpsolve55.jar wurde die Anwendung vom Solver entkoppelt, was die Interoperabilität erhöhte. Ein spezifisches Exception-Handling wurde in Form von try-catch-Statements konsequent durchgeführt sodass eine entsprechende Programmsicherheit gewährleistet werden konnte.
* Reliability  
  Es findet eine Überprüfung der Benutzereingaben statt. So lassen sich beliebig viele Maschinen und Produkte anlegen, jedoch nur bis zu einem Maximum von jeweils 16 Maschinen und Produkten. Außerdem lassen sich nur ganzzahlige Durchlaufzeiten im Intervall von [0,99] angeben. Jedoch steigt der Ressourcenverbrauch und somit die Berechnungsgeschwindigkeit mit steigender Größe des LP-Modells. Das liegt zum einen an den vielen Anforderungen in Form von Restriktionen[[5]](#footnote-5) aber auch am lpSolve. Da es sich um ein Open Source Produkt handelt ist dieser nun mal nicht so leistungsstark wie z.B. ein kommerzieller IBM ILOG CPLEX.
* Usability  
  Die GUI ist einfach gehalten und bietet alle Grundfunktionalitäten an, so dass die Software schnell erlernbar ist trotz keiner Benutzeranleitung. Durch eine implementierte Eingabevalidierung werden entsprechende Nutzungshinweise dem Benutzer angezeigt.
* Portability  
  Da es sich um eine Neuimplementierung in Java handelt lässt sich die Software prinzipiell auf jedem Betriebssystem ausführen. Da jedoch auf ein externer Solver zur Lösung der LP-Modelle zurückgegriffen wird, muss neben der entsprechenden JRE auch eine vom Betriebssystem unterstützte Version des lpSolve installiert sein, was die Portabilität des Softwaretools in der Hinsicht einschränkt, dass zusätzlich der lpSolve installiert sein muss. So muss z.B. unter Windows die entsprechende DLL-Datei (32-/64-Bit-Architektur) im Projekt eingebunden sein. Dadurch entstanden zwei Softwareprodukte, ein JobShopNeu für Win7 32-Bit-Architektur und ein JobShopNeu für Win7 64-Bit-Architektur.
* Maintainability  
  NetBeans nutzt keine proprietären Klassen, sodass der Programmcode 1:1 auch in ein Eclipse-Projekt kopiert werden kann und dort ohne Anpassung läuft. Jedoch mussten einzelne Klassenbibliotheken nachgeladen werden da diese ansonsten nicht exportiert werden konnten.

# Begründung von Entwurfsentscheidungen

Wie bereits in Kapitel 4.1 beschrieben, wurde primär der erste Lösungsansatz verfolgt. Hintergründe für diese Entscheidung waren vor allem

* Wiederverwendbarkeit des bestehenden Softwarecodes
* Entkopplung zwischen Java-Anwendung und konkreten Solver
* Wartbarkeit

Gleichwohl wurden auch Ansätze in Bezug auf die Lösungsansätze 2 und 3 ausgearbeitet, da selbst nach 2 Monaten sich kaum Projektfortschritte in Hinblick auf Lösungsansatz 1 ergaben. Für Lösungsansatz 2 wurde ähnlich wie im Softwareprodukt „Diätplaner“ versucht, den lpSolve direkt in das Projekt einzubinden. Dabei müsste wie bereits in unserem Java-Projekt das LP-Modell z.B. in Form einer Matrix dargestellt werden. Diese müsste zusätzlich noch um die Zielfunktion (MIN/MAX!) ergänzt werden. Anschließend müsste in einer neuen Klasse (z.B. LpSolve.java) der in das Projekt eingebundene lpSolve statisch über eine (oder mehrere) Pathvariable(n) gesetzt werden. In einer entsprechenden Methode müsste dann ein *Process[[6]](#footnote-6)* gestartet werden über welchen der Solver angesprochen werden wurde. Über entsprechende Input/Output-Streams würde das LP-Modell eingelesen bzw. das Ergebnis zurückgegeben werden. Über die genau Funktionsweise dieses Ansatzes soll auf das Projektteam „Diätplaner“ verwiesen werden.

Auch wurde „GLOP“ näher analysiert. Eine detailiertere Beschreibung sowie Einbindung in ein Java-Projekt kann unter[[7]](#footnote-7)

Einführung : https://developers.google.com/optimization/docs/lp/glop

API : https://developers.google.com/apps-script/reference/optimization/

Installation : https://developers.google.com/optimization/installing

nachgelesen werden. Wie bereits erwähnt wurde jedoch dieser Ansatz als Notfallplan

ausgearbeitet und direkt wieder verworfen, nachdem Ansatz 1 funktionierte.

## Problemerörterung

Die Applikation JobShopNeu ließ sich bisher über eine Jar-Datei starten, jedoch konnte keine Berechnung durchgeführt werden. Wurde das Projekt in Eclipse eingebunden und gestartet, gab es unter anderem Laufzeitfehler bezüglich der Schnittstelle zu lpSolve (z.B.: no lpsolve55j in java.library.path oder lpSolve55j.dll konnte nicht gefunden werden). Nach einiger Recherche haben wir herausgefunden, dass einige DLL-Dateien für verschiedene Betriebssysteme zwar im Projekt abgelegt wurden, jedoch zur Lauffähigkeit noch weitere fehlten. Im folgenden Abschnitt wird der Zusammenhang zur DLL-Dateien näher erläutert:

DLL ("Dynamic Link Library")-Dateien wie z.B. lpsolve55.dll sind kleine Programme, ähnlich EXE ("ausführbare")-Dateien, die es verschiedenen Sotftwareprogrammen ermöglichen, dieselbe Funktion zu nutzen (z.B. drucken). Die lpsolve55.dll ist eine DLL-Datei, die im Zusammenhang mit lpSolve steht, dass von Mixed Integer Linear Program Solver für das Windows-Betriebssystem entwickelt wurde. Die aktuellste Version von Lpsolve55.dll ist momentan 5.5.0.10.

DLL Fehler können auftreten, weil sie eben gemeinsam genutzte Dateien sind und so Windows beim Laden dieser auf Fehler und somit auf einen Abbruch der Applikation stoßen kann. Außerdem können DLL-Dateien beschädigt sein.

## Lösungsansatz

Um die vorher angedeuteten Probleme zu beheben, haben wir die aktuellsten DLL-Dateien lpsolve55.dll und lpsolve55j.dll heruntergeladen. Diese können von der Bit-Rechnerarchitektur anschließend direkt in das Projekt eingebunden werden. Somit wurde der Fehler behoben und die Berechnung lief in Eclipse. Anschließend wird das gesamte Projekt in einer JAR-Datei gespeichert. Diese kann nun auf Grund der DLL Abhängigkeit jedoch nicht direkt per Doppelklick ausgeführt werden. In nachfolgendem Beispiel für eine 64Bit-Rechnerarchitektur werden die bisherigen Schritte näher erläutert[[8]](#footnote-8):

## Vorgehensweise zur Erstellung der 64 Bit-Applikation:

Die beiden DLL-Dateien für eine 64Bit-Architektur direkt in das Projekt einbinden, wenn nötig überschreiben:

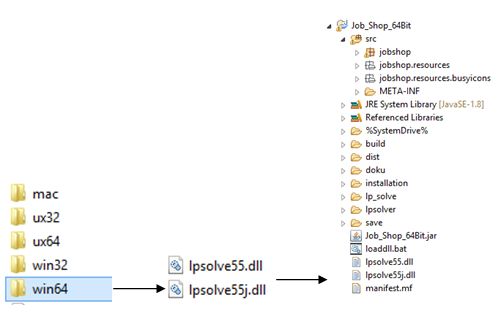


Abbildung : DLL in Projekt importieren

Als runnable Jarfile speichern:

* In der Launch configuration die Startklasse auswählen
* In der Export destination den Projektordner auswählen und JAR-Datei benennen (Hier: Job\_Shop\_64Bit.jar)

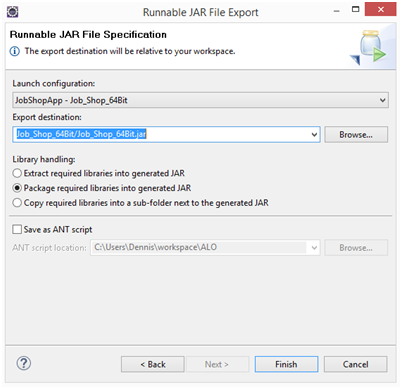


Abbildung : Runnable JAR File Export

Damit die JAR-Datei sich starten lässt, wurde direkt im Projektordner eine BATCH-Datei erstellt, welche diese aufruft.

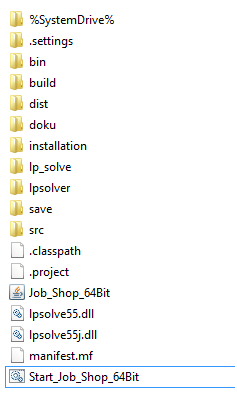


Abbildung : Projektinhalt

Inhalt der Batch-Datei Start\_Job\_Shop\_64Bit:

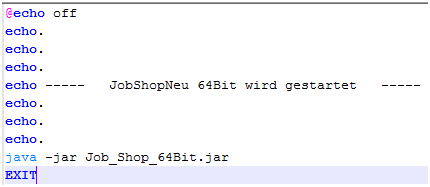


Abbildung : Aufbau Start\_Job\_Shop\_64Bit.bat-Datei

Somit lässt sich die Applikation JobShopNeu über die Batch-Datei starten:

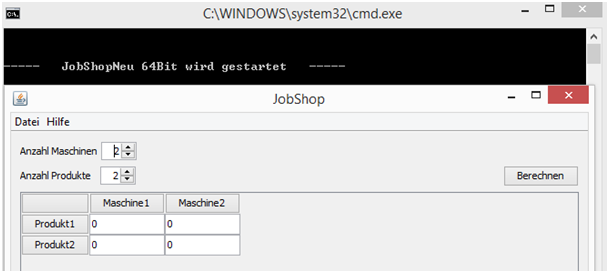


Abbildung : Programmaufruf

# Weiterentwicklung JobShopNeu

Im Folgenden werden alle Aktivitäten zur Erstellung der Lauffähigen JobShopNeu (32-/64-Bit) näher erläutert.[[9]](#footnote-9)

Die Anwendung wurde des Weiteren wie folgt weiterentwickelt:

* Eine Erweiterte Eingabematrix von 16 Maschinen und 16 Produkten

Die Code wurde insofern erweitert, dass nun die Möglichkeit besteht einen Maschinen- und Produktanzahl von 16 auszuwählen und diese berechnen zu lassen. Davor lag die Anzahl bei lediglich 10.

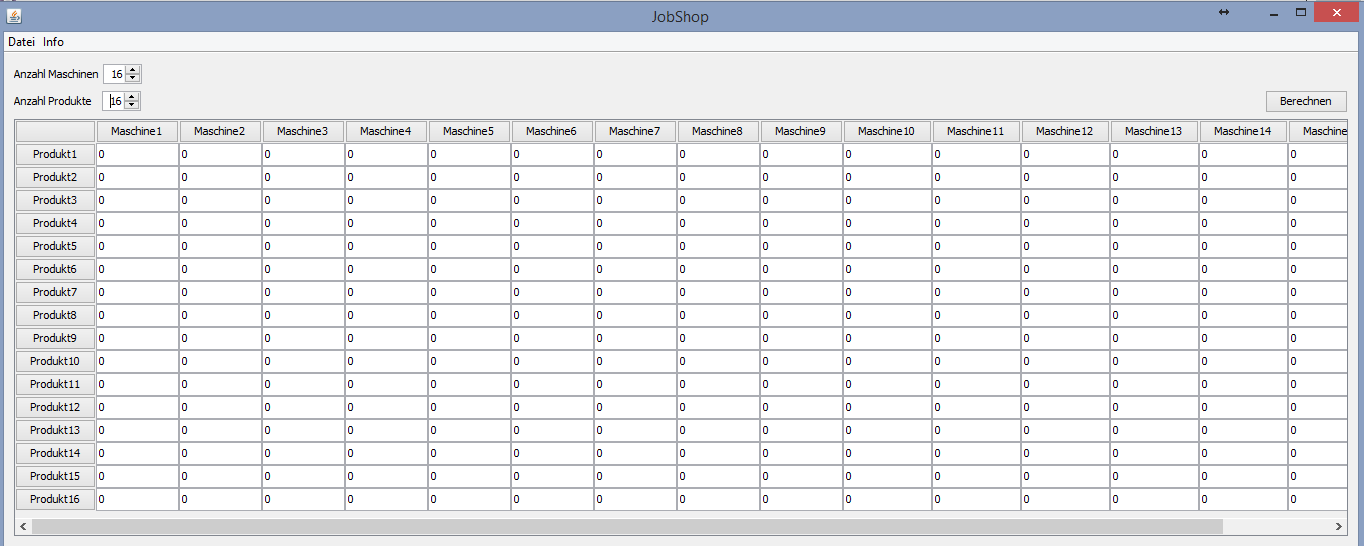


Abbildung : Eingabematrix 16/16

Dabei mussten die Maximalwerte der Methode *initComponents()* in der Klasse *JobShopView* auf 16 gesetzt werden:

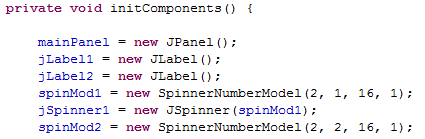


Abbildung : Methode initComponents()

* Erstellung eines PopUp Fensters bei Falscheingabe (Bsp.: String)

Um Falscheingaben hinsichtlich der Durchlaufzeit zu vermeiden, wurde die Methode *stringToValue(String s)* aus der Klasse *JobShopNumberFormatter* insofern erweitert, dass bei Eingabe eines Strings ein PopUp Fenster mit der Meldung „Sie müssen eine Ganzzahl eingeben“ erscheint.

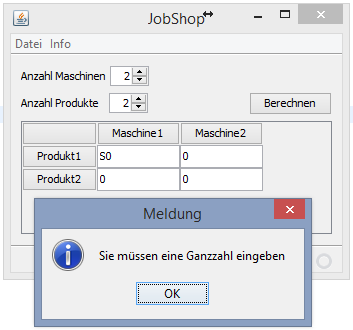


Abbildung : Popup Beispiel bei Falscheingabe

Sofern eine Ganzzahl über 99 eingegeben wird, springt diese automatisch auf 0, da nur Werte bis 99 akzeptiert werden.

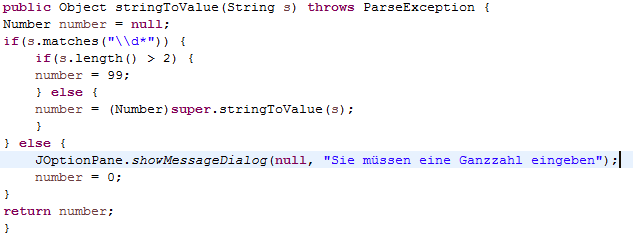


Abbildung 21: Methode stringToValue(String s)

* Das zu berechnende LP Modell (Eingabematrix) wird gespeichert/ geladen

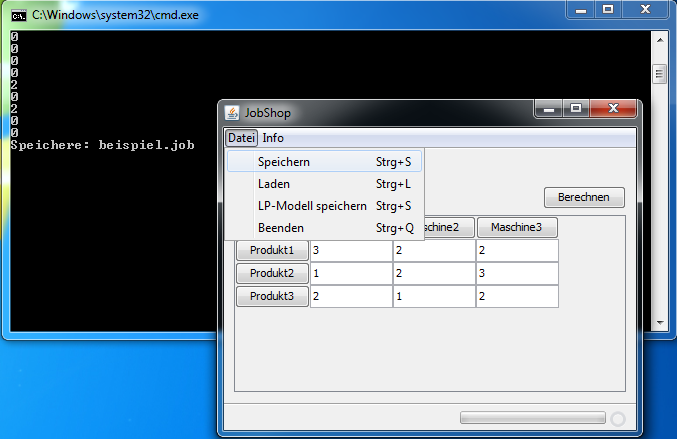


Abbildung 22: Eingabematrix speichern

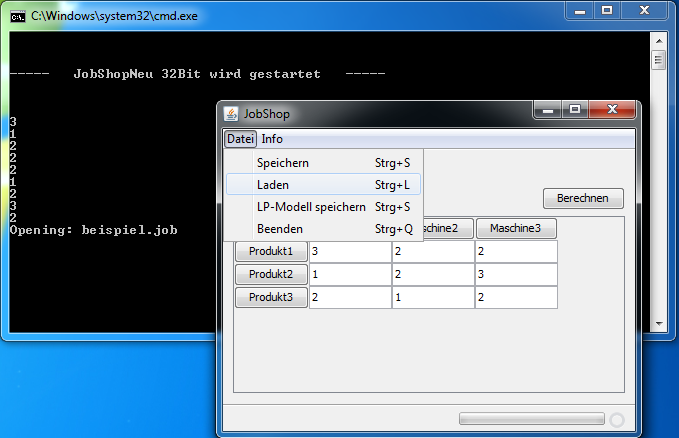


Abbildung 23: Eingabematrix laden

* Speichern des aufgestellten LP-Modells sowie den Lösungsvektor

Zur Überprüfung der Fachlichkeit wurde der Code um einen FileWriter ergänzt, welcher das erstellte LP-Modell in einer temporären TXT-Datei abspeichert. Ebenfalls wird eine temporäre TXT-Datei für die Speicherung der berechneten Lösung im entsprechenden Projektverzeichnis erstellt.

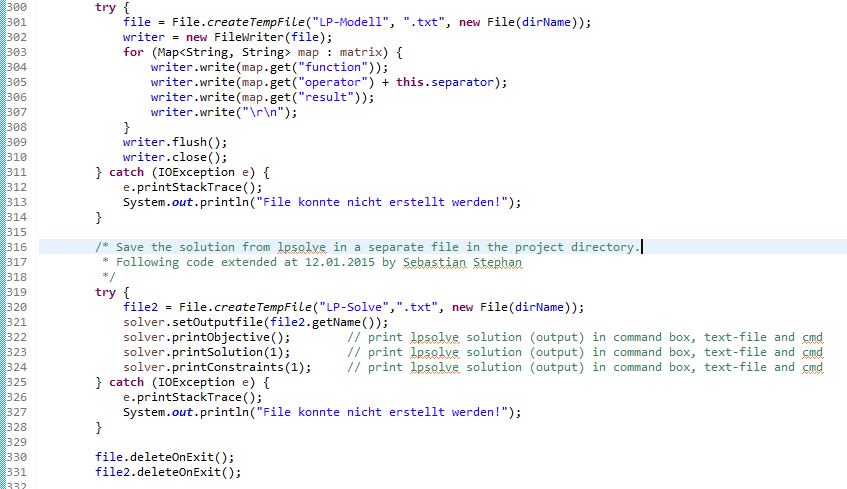


Abbildung 24: FileWriter-Erweiterung

Der Lösungsvektor für das mitgelieferte Beispiel-LP-Modell sieht dann z.B. wie folgt aus:

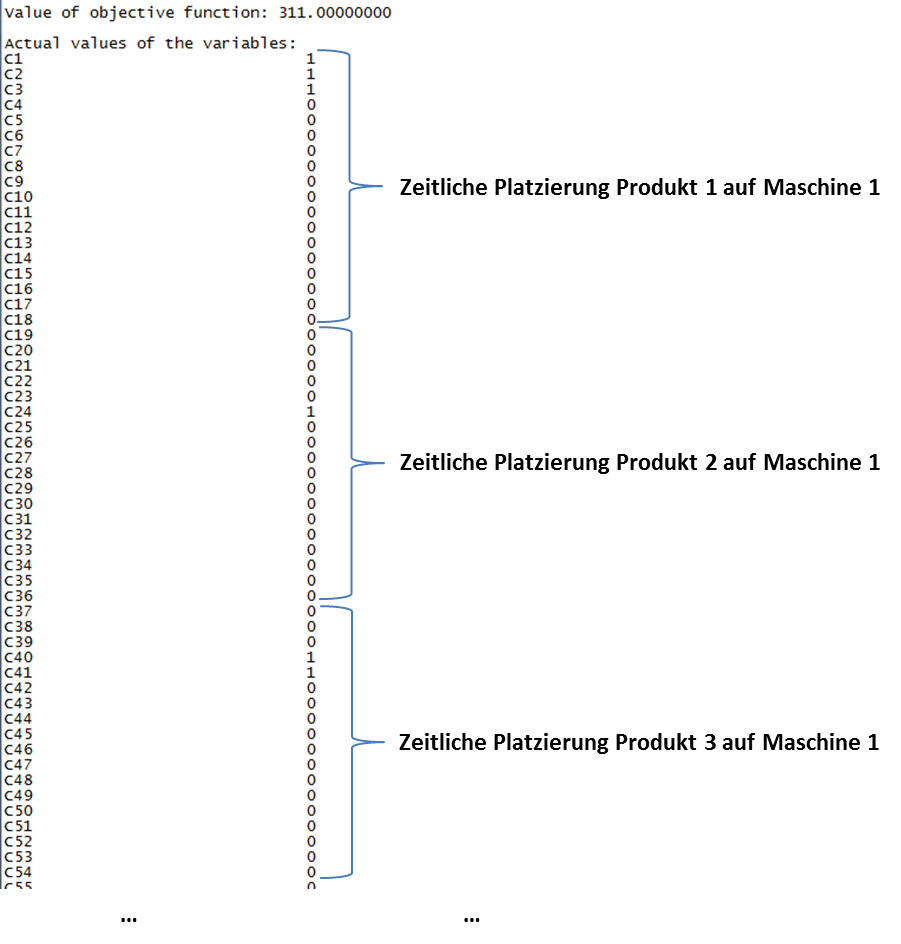


Abbildung 25: Lösungsvektor lpSolve

# Beispielaufgabe

Im Folgenden soll die Funktionsweise der Software anhand einer kleinen Beispielaufgabe veranschaulicht werden.

Auf 3 Maschinen können jeweils 3 Produkte bearbeitet werden. Jedes Produkt kann pro Zeiteinheit (ZE) nur von genau einer Maschine bearbeitet werden. Ebenso kann eine Maschine immer nur ein Produkt pro ZE bearbeiten. Die jeweiligen Durchlaufzeiten der Produkte auf den Maschinen sind in nachfolgender Tabelle aufgezeigt:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Maschine1 | Maschine2 | Maschine3 |
| Produkt1 | 3 | 2 | 2 |
| Produkt2 | 1 | 2 | 3 |
| Produkt3 | 2 | 1 | 2 |

**Aufgabe:** Erstellen Sie einen Maschinenbelegungsplan bei dem die Durchlaufzeit möglichst minimal ist (optimal).

## Lösung traditionell mit lpSolve:

Zunächst müsste ein LP-Modell erstellt werden. Dieses Modell würde dabei unter Berücksichtigung der entsprechenden Anforderungen

* Ein Produkt kann pro Zeitintervall nur auf einer Maschine bearbeitet werden
* Eine Maschine kann nur an einem Produkt pro Zeitintervall arbeiten
* Produkte sollen möglichst am Stück bearbeitet werden (Zusatz)

über >300 Variablen sowie über >500 Restriktionen umfassen. Dieses LP-Modell aufzustellen würde sehr aufwändig sein. Aus diesem Grund wurde die Software "JobShop“ entwickelt.

## Lösung mit JobShopNeu:

Die Eingabe erfolgt in Form einer Matrix.

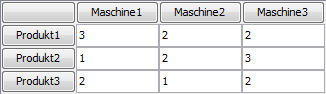


Abbildung 26: Eingabematrix JobShopNeu

Entsprechend den genannten Anforderungen wird folgender Lösungsvektor berechnet:

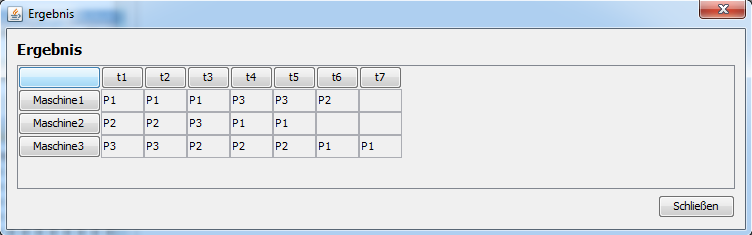


Abbildung : Lösungsmatrix JobShopNeu

**Hinweis:**

Fertigstellungszeiträume einzelner Produkte werden nicht im LP-Modell mit berücksichtigt.

**Bug:**

Wird ein gespeichertes LP-Modell geladen, müssen zuerst alle Texteingabefelder „aktiviert“ werden, d.h. durch einen Mausklick in entsprechendes Feld oder Ähnlichem. Grund hierfür liegt in der verwendeten Softwarekomponente „SingleFrameApplication“

# Fazit

Die Lehrveranstaltung Anwendung der Linearen Optimierung (ALO) ermöglichte uns erste Einblicke in die Planung, Modellierung und Implementierung von anwenderfreundlichen Problemlösungen auf der Grundlage von Modellen der Linearen Programmierung (Operations Research). Dabei lag der Fokus nicht nur ausschließlich auf der fachlichen Ebene, der Entwicklung mathematisches LP-Modell, sondern auch deren Lösung softwaretechnisch durch Eigenimplementierung sowie unterschiedlicher LP-Solver-Produkte (z.B. lpSolve, GLOP). Durch die Bearbeitung verschiedener Problemstellungen in Form von Projekten konnten wir bereits gelerntes direkt anwenden und so weitere praktische Erfahrungen sammeln. Im Team aber auch durch Projektbesprechungen zusammen mit Prof. Grütz lernten wir für auftauchende Problemstellungen Lösungsskizzen und –alternativen auszuarbeiten und ggf. umzusetzen.

Einziger Kritikpunkt sind die zum Teil hohen projektabhängigen softwaretechnischen Anforderungen.

Zusammenfassend lässt sich festhalten, dass die Projektarbeit im Rahmen der Veranstaltung ALO sehr interessant war. Es wurden Einblicke in die praktische Anwendung der Linearen Optimierung ermöglicht sowie die OR-Kenntnisse gefestigt und erweitert. Außerdem konnten wir, da es sich um ein Java-Projekt handelte, zusätzlich unsere Programmier- und Softwaremodellierungskenntnisse erweitern.

1. Bei JobShopNeu handelt es sich um die Neuimplementierung von JobShop in Java. [↑](#footnote-ref-1)
2. vgl. Dokumentation – JobShop2.docx (WS 2010/11) [↑](#footnote-ref-2)
3. vgl. Dokumentation – JobShop2.docx (WS 2010/11) [↑](#footnote-ref-3)
4. Wobei primär der Fokus auf dem Open Source Produkt lpSolve lag (auch in Hinblick auf die Veröffentlichung in ORWEB). [↑](#footnote-ref-4)
5. So besitzt z.B. eine 2x3 Matrix 132 Variablen + 127 Restriktionen [↑](#footnote-ref-5)
6. Vgl. <http://docs.oracle.com/javase/7/docs/api/java/lang/Process.html>, Abruf: 19.01.2015 [↑](#footnote-ref-6)
7. Abgerufen am 19.01.2015 [↑](#footnote-ref-7)
8. Gilt analog für die Erstellung der 32 Bit-Applikation [↑](#footnote-ref-8)
9. Zu berücksichtigen sind ebenfalls die im Code enthaltenen Kommentare. [↑](#footnote-ref-9)