Implementierung
Graph von Ansicht

Nicolas Boltz
uweaw@student.kit.edu

Jonas Fehrenbach
urdtk@student.kit.edu

Sven Kummetz
kummetz.sven@gmail.com

Jonas Meier
Meierjonas96@web.de

Lucas Steinmann
ucemp@student.kit.edu

Inhaltsverzeichnis

1 Einleitung

Nachdem das Programm „Graph von Ansicht“ bereits in einem Pflichtenheft definiert und der Programmaufbau in einem Entwurf ausgearbeitet wurde, war der nächste Schritt das Implementieren anhand des Entwurfes. Dieses Dokument beschreibt die Planung und den Ablauf der Implementierungphase, welche vom 21.06.16 bis zum 19.07.16 dauerte. Es zeigt insbesondere die Probleme bei der Implementierung auf und dokumentiert die resultierende Änderungen an dem Entwurf.

Da sich manche Designentscheidungen und Strukturen aus dem Entwurf während der Implementierung als eher unpraktisch oder nicht umsetzbar herausstellten, mussten diese abgeändert werden. Betroffen von diesen Änderungen waren vor allem das generisch entworfene Graphmodel und der Layoutalgorithmus, welcher auf dem Sugiyama Framework basiert. Dies lag daran, dass der Entwurf von „Graph von Ansicht“ an manchen Stellen nicht genau definiert wurde und wichtige Implementierungsvorrausetzungen übersehen wurden.
Da auch viele kleine Änderungen gemacht wurden (z.B das Ändern von Parametern in einer Methode) werden in diesem Dokument nur die Änderungen dokumentiert, welche den Programmablauf oder den Sinn einer Klasse stark verändern.

Im Pflichtenheft wurden Anforderungen und Kriterien beschrieben, welche im Programm umgesetzt werden sollten. Dabei wurden jedoch einige kleine Änderungen an der Steuerung in der View (siehe Kapitel 3.1.3) und an dem Eingabeformat von Kommandozeilenparameter vorgenommen.

Zusätzlich wurden im Pflichtenheft auch Wunschkriterien beschrieben, die im Entwurf berücksichtigt wurden und welche bei Möglichkeit in das Programm übernommen werden sollten. Da jedoch die Zeit für die Implementierung von „Graph von Ansicht“ recht knapp bemessen war, und die Pflichtkriterien natürlich mit Priorität behandelt wurden, wurde keine der Wunschkriterien aus dem Pflichtenheft umgesetzt.

2 Änderungen am Entwurf

2.1 General

1. **Aktion:** Methode *getGraphBuilder()* von *IVertexBuilder* nach *IGraphBuilder* verschoben.
   Grund: Dadurch können hierachische Graphen einfacher erstellt werden, da der *IGraphBuilder* den *IGraphModelBuilder* kennt.

2. **Aktion:** Collabsable Graph Interface hinzugefügt.
   Grund: Zwischen einem zusammengeklappten Subgraphen und einer *CompoundVertex*(z.B. FieldAccess) war nicht zu unterscheiden. Durch die *CollapsedVertex* und ihre speziellen Subklassen ist die genaue Unterscheidung möglich.

3. **Aktion:** Abhängigkeiten zwischen *JoanaGraphModelBuilder*, *JoanaGraphBuilder*, *JoanaVertexBuilder* und *JoanaEdgeBuilder* hinzugefügt.
   Grund: Jede Builder Klasse muss wissen von welchem Builder sie erstellt wurde, um das erstellte Produkt (z.B. *JoanaVertex* oder *JoanaEdge*) bei seiner Elternklasse abspeichern zu können.

4. **Aktion:** *AbstractPluginBase* hinzugefügt.
   Grund: Die meisten Funktionen in den speziellen Plugins sind fast leer, müssen aber überschrieben werden. Die *AbstractPluginBase* reduziert diesen redundanten Code.

5. **Aktion:** *build()*-Funktionen aus *IGraphBuilder*, *IVertexBuilder* und *IEdgeBuilder* entfernt.
   Grund: *build()* wird nun nur auf den *IGraphModelBuilder* aufgerufen, welcher die speziellen build-Funktionen der konkreten Klassen rekursiv aufruft.

6. **Aktion:** *nodeKind*(String) aus *JoanaVertex* und *edgeKind*(String) aus *JoanaEdge* durch jeweils einen Enum ersetzt und die Interfaces entsprechend angepasst.
   Grund: Die genauen Werte die *nodeKind*/*edgeKind* annehmen können sind zur Compile-Zeit festgelegt. Dadurch können bestimmte Werte, wie Farbe und potentiell maximale Breite von Knoten nun durch den Typen festgelegt werden.

7. **Aktion:** Jede *GraphView* besitzt ihre eigene *GraphViewGraphFactory*.
   Grund: Die *GraphViewGraphFactory* bietet Zugriff mittels eines Mappings von GUI-Elementen auf die Modelelementen und auf den dargestellten Graph.

8. **Aktion:** *SerializedGraph* verschoben. Der angezeigte Graph wird nun von der *GraphViewGraphFactory* serialisiert und nicht im Model.

Grund: Die *GraphViewGraphFactory* besitzt mehr Information (z.B. Farbe, Gruppe usw.) über den Graphen als das Model. Diese Informationen sind vor allem für den *SvgExporter* wichtig.

9. **Aktion:** Klasse *DefaultDirectedEdge* hinzugefügt, Klasse *DirectedEdge* ist nun ein Interface. Die Vorkommen von *DirectedEdge* wurden, falls nötig, zu *DefaultDirectedEdge* geändert.

Grund: Es war im Sinne von Erweiterungen nötig, ein Interface einer *DirectedEdge* zu haben.

10. **Aktion:** *getColor()* in *Edge* und *Vertex* hinzugefügt.

Grund: Die Farbe muss von der *Edge/ Vertex* vorgegeben werden, damit die GUI verschiedene Farben vergeben kann, *JoanaEdge* und *JoanaVertex* geben jeweils eine Farbe zurück, die von ihrem Typ bestimmt wird.

11. **Aktion:** Aktuelle *LayoutOption* wird in *GraphView* gespeichert.

Grund: Um zu wissen welcher Layoutalgorithmus zuletzt angewendet wurde und um diesen wiederholt anwenden zu können. Dies ist unter anderem bei der Collapse bzw. Expand Funktion und dem Ändern der Properties nötig.

12. **Aktion:** Die Realisierung von Gruppen wurde in die GUI verschoben und jede *VertexShape* speichert ihren VertexStyle.

Grund: Da Gruppen nicht direkt mit dem darunter liegenden Datenmodel zusammenhängen, werden sie komplett in der GUI realisiert. Dazu wird für eine bestimmte Untermenge der Knoten eine von JavaFX definierter Style, im CSS Format, gesetzt. Dieser wird in der *VertexShape* gespeichert und mittels *getVertexStyle()* und *setVertexStyle()* zugegriffen.

13. **Aktion:** Klasse *Point* zu *IntegerPoint* geändert, Klasse *DoublePoint* hinzugefügt.

Grund: In vielen Fällen war ein Punkt mit double Argumenten nötig, daher wurde einer für Integer und einer für double gemacht.

14. **Aktion:** Entfernen der generischen Parameter für *Vertex* und *Edge* in *Graph* sowie für *Vertex* in Edge.

Grund: Zwischen den verschiedenen Hierarchie-Ebenen von Graphen und Vertex/Edge bestand eine kovariante Beziehung (z.B. *Graph* kann alle Typen von *Vertex* und *Edge* enthalten, *DirectedGraph* kann jedoch nur Typen von *Vertex* und *DirectedEdge enthalten*). Um im Allgemeinen alle Möglichkeiten zu unterstützen, mussten diese Beziehungen in komplizierten und langen generischen Ausdrücken definiert werden. Zum Beispiel: `LayoutAlgorithm<G extends DirectedGraph<V,E>, V extends Vertex, E extends DirectedEdge<V>`.
An vielen Stellen war die Belegung der Parameter allerdings uninteressant, weshalb oft Raw-Types als Alternative gewählt wurden, anstatt der langen, (zu) allgemeinen Parametrisierung: `Graph<? extends Vertex, ? extends Edge<?>.`

15. **Aktion:** Entfernen der Vererbung *JoanaGraph* von /DefaultDirectedGraph.

Grund: Als Folge des Entfernens der Generics konnten von *DefaultDirectedGraph* erbende Graphen, welche auf die Implementation der Speicherung von Edges und Vertex in *DefaultDirectedGraph* zurückgreifen, dies nicht weiterhin ohne einen Verlust des genauen Typs der Edge/Vertex tun.

16. **Aktion:** *DefaultDirectedGraph<V,E>* und *DefaultLayering<V>* als Datenstruktur für andere Graphen.

Grund: Um nicht durch das Entfernen der Vererbung von *DefaultDirectedGraph* grundlegende Funktionen eines Graphen in jedem Graph neu implementieren zu müssen, wurde `DefaultDirectedGraph<V,E>` als Datenstruktur umfunktioniert, welcher in einer Komposition in anderen Graphen, wie JoanaGraph und SugiyamaGraph genutzt werden kann. Gleiches gilt bei *DefaultLayering* für das Speichern von relativen Positionen in Graphen.

2.2 Sugiyama

1. **Aktion:** Interface *ISugiyamaVertex* hinzugefügt. Außerdem implementieren *SugiyamaVertex* und *DummyVertex* dieses Interface.
   Grund: In vielen Schritten, vor allem im *CycleRemover* war es nötig, *SugiyamaVertex* und *DummyVertex* zusammen in einem Set zu haben, was aufgrund der Vererbungshierarchie bislang nicht möglich war. Weshalb beide nun das neue Interface implementieren, somit gemeinsam behandelt werden können und zusätzlich die Funktionalitäten anbieten können, die für die einzelnen Schritte von beiden Knotentypen notwendig sind.
2. **Aktion:** Interface *ISugiyamaEdge* hinzugefügt. Außerdem implementieren *SugiyamaEdge* und *SupplementEdge* dieses Interface.
   Grund: Aufgrund der Implementierungsentscheidung, die *SupplementEdge*s anstelle der *SupplementPath*s zu den nicht durch *Supplementpath*s ersetzten Kanten des Graphen zu speichern, war es nötig, *SugiyamaEdge*s und *SupplementEdge*s zusammen in einem Set zu haben. Deswegen implementieren beide nun dasselbe Interface. Zudem können so auch einfacher gemeinsame Methoden der beiden verschiedenen Kanten definiert werden, welche von diesen für weitere Schritte unterstützt werden müssen.
3. **Aktion:** Interface *ISugiyamaStepGraph* hinzugefügt, welches das Interface *LayeredGraph* erweitert. Die Interfaces *ILayerAssignerGraph*, *ICrossMinimizerGraph*, *IVertexPositionerGraph* und *IEdgeDrawerGraph* erweitern nun *ISugiyamaStepGraph* anstelle von *LayeredGraph*.
   Grund: Bietet allen Funktionen nötige zugriffe auf Informationen über den Graphen, wie zum Beispiel Eingangs- und Ausgangsgrad, die Layernummer eines Knoten, eingehende- und ausgehende Knoten eines Knoten ... Somit werden Duplikate von Methoden mit dem gleichen Zweck in den einzelnen Graph-Interfaces für verschiedene Schritte vermieden.
4. **Aktion:** Klasse *Point* aus dem sugiyama-Paket in das Paket *shared/src/main/java/edu.kit.student.util* verschoben.
   Grund: Nicht nur der Sugiyama braucht Punkte, daher wurde es in einen allgemeinnütziges Paket, der Übersichtlichkeit wegen, verschoben.

3 Implementierte Muss- und Wunschkriterien

3.1 Pflichtkriterien

3.1.1 Allgemein

- Hierarchisches Layout mit dem Sugiyama-Framework (/FA030/)
- Ein Callgraph-Layout, welches übersichtlich die Abhängigkeiten der Methoden darstellt (/FA030/)
- Ein Methodgraph-Layout, welches die Abhängigkeiten innerhalb einer Methode - mithilfe von vorgegebenen Constraints darstellt (/FA030/)
- Kollabieren und Ausklappen von Subgraphen (/FA060/ und /FA070/)
- Informationsanzeige zu einzelnen Knoten und Kanten (/FA270/)
- Statistiken über den Graphen und Subgraphen (/FA280/)
- Filter für Knoten- und Kantentypen aus JOANA (/FA040/)
- Tabs für geöffnete Graphen (/FA260/)
- Akzeptieren von Kommandozeilenargumenten zur Angabe von Graphdatei und Layoutalgorithmus für ein schnelles Starten
  - Das Eingabeformat der Kommandozeilenargumente hat sich zum Pflichtenheft geändert, um ein einfacheres parsen der Parameter zu ermöglichen. Argumente werden nun im folgenden Format eingegeben: „–in=<datei>“
- Das Produkt wird unter einer freien Lizenz veröffentlicht
- Die Anzeigesprache der GUI ist englisch. Ein Sprachwechsel soll aber leicht zu implementieren sein. (/NFA100/)
  - JavaFX bietet bereits eine Möglichkeit verschiedene Sprachen zu verwenden und Oberflächenstrings anhand einer ID zu vergeben.

3.1.2 Input/Output

- Import von generischen Graphen im GraphML-Format (/FA100/)
- Export der visualisierten Graphen im SVG-Format (/FA110/)

3.1.3 Steuerung

- Navigation mittels Zoom (/FA210/) und Verschieben (/FA200/)
  - Die im Pflichtenheft definierte Maustastenbelegung kann von Java leider nicht unter jedem Betriebssystem realisiert werden. Im Fall von Windows 8.1 gibt es eine Funktionalität, die auf den Klick des Mausrads reagiert. Aus diesem Grund wurde die Tastenbelegung leicht abgeändert:
  - Das Sichtfeld verschiebt man nun durch das Drücken von STRG und klicken und ziehen der rechten Maustaste.
- Selektieren (/FA220/) und Deselektieren (/FA230/) von einzelnen oder mehreren Knoten
  - Um das Selektieren und Deselektieren von Knoten benutzerfreundlicher zu gestalten, wurde die Tastenbelegung an die der gängigen Dateisystemexplorer angepasst:
  - Zum Hinzufügen oder Entfernen eines einzelnen Knotens aus der Selektion von mehreren Knoten muss beim Klicken STRG gehalten werden. Ansonsten wird die komplette Selektion aufgehoben und nur der angeklickte Knoten selektiert.
  - Ist eine Menge von Knoten selektiert, STRG gedrückt und man selektiert durch ziehen der Maus mehrere Knoten, werden bereits selektierte Knoten aus der Menge entfernt und nicht selektierte Knoten hinzugefügt.

3.1.4 Plugins

- Schnittstellen für Plugins in den Bereichen Import, Export, Layoutalgorithmen, Filter für Knoten- und Kantentypen und weitere Operationen auf einzelne Knoten und Kanten (Pflichtenheft Kapitel 7)
- Es gibt ein Pluginmanagement-System, welches externe Plugins laden und verwalten kann. (Pflichtenheft Kapitel 7)

3.2 Wunschkriterien

Aus den Wunschkriterien die im Pflichtenheft definiert wurden, wurden keine Implementiert.

4 Implementierungsplan

Um einen Implementierungsplan aufzustellen, wurde das Projekt zuerst anhand der einzelnen Plugins/Pakete aufgeteilt. Danach wurden die Plugins/Pakete noch feiner unterteilt und es wurde versucht einzelne, möglichst voneinander unabhängige, Aufgaben zu finden. Der nötige Arbeitsaufwand zum Bearbeiten der einzelnen Aufgaben wurden von allen Projektmitgliedern abgeschätzt und der Mittelwert als geplante Bearbeitungsdauer festgesetzt. Anhand von den bestehenden Abhängigkeiten der Aufgaben untereinander und der vorausgesetzten wöchentlichen Arbeitsleistung, wurden die Aufgaben auf die einzelnen Wochen und Personen verteilt.

		Geschätzter Aufwand	Bearbeitung in Woche	Zugewiesen an
objectproperties		4	1	Nicolas
parameter		5	1	Nicolas
graphmodel				
	Interfaces schreiben	2	1	Jonas F.
	Default-Klassen implementieren	8	1	Jonas F.
	Layout-Register schreiben	4	1	Jonas F.
	Serialized-Klassen	2	1	Jonas F.
sugiyama				
	Graph + Impl Interfaces + Vertex/Edge	9	1	Jonas M + Sven
	Phase 1	7	1	Jonas M + Sven
	Phase 2	10	1	Jonas M + Sven
	Phase 3	10	2	Jonas M + Sven
	Phase 4	10	2	Jonas M + Sven
	Phase 5	10	2	Jonas M + Sven
	Constraints implementieren	4	3	
	LayoutAlgorithm	2	3	
joana				
	Callgraph	5	2	Lucas
	Methodengraph	5	2	Lucas
	FieldAccess	3	2	Lucas
	GraphModel	4	2	Lucas
	CallgraphLayout/Register/Option	13	3	
	MethodengraphLayout/Register/Option	13	3	
	GraphBuilder-Klassen	12	3	
	JoanaPlugin	6	3	
	Joana Vertex/Edge	4	2	Nicolas
	Workspace	5	3	
	JoanaGraph	2	2	Nicolas
GUI				
	GraphFactory	5	3	
	GraphView(Navigation und Zoom)	10	3	
	StrukturView	4	4	
	InformationView	4	4	
	Generelle GUI	11	1	Nicolas
	Laden der Plugins	6	2	Nicolas + Lucas
	Import und Export anstoßen + Dialoge	5	2	Nicolas
	ParameterDialogCreator	6	3	
Plugin				
	Manager	11	1	Lucas
	Interfaces schreiben	3	1	Lucas
	Filter	7	3	
	Workspace-/EntrypointOption	4	3	
Import		11	2	Jonas F.
Export		7	4	

Abbildung 4.1: Erster Implementierungsplan

4.1 Woche 1

In der ersten Woche wurde der zuerst erstellte Implementierungsplan jedoch komplett überarbeitet. Dies war nötig da im ersten Plan viele Abhängigkeiten zwischen den Aufgaben nicht berücksichtigt wurden. Zum Beispiel hing das restliche Projekt komplett von der Fertigstellung des Sugiyama ab und hätte ohne diesen nicht getestet werden können. Um dieses Problem zu lösen, wurden vorläufige Mockups für die einzelnen Phasen eingeplant. Die anderen Aufgaben wurden zusätzlich in der Abarbeitungsreihenfolge umsortiert, verfeinert und für die ersten zwei Wochen bereits einzelnen Personen zugewiesen. Außerdem wurde eine Excel Tabelle mit dem bereits existierenden Plan erstellt und durch eine Fortschrittsspalte erweitert. In der Fortschrittspalte sollten die Entwickler den prozentualen Fortschritt an den ihnen zugeteilten Aufgaben eintragen. Dadurch konnte ein Fortschrittsdiagram erstellt werden, welchen den aktuellen Soll- und Ist-Zustand des gesamten Projekt visualisiert.

Zusätzlich zur Änderung des Plans wurde eine grobe Projektstruktur in Gradle aufgebaut und die bereits im Entwurf erstellte Interfaces in das Projekt überführt. Um die Gradle Projektkonfiguration zu testen wurde eine grobe GUI implementiert und mit der Implementierung der Allgemein genutzten Klassen aus dem Paket ”graphmodel”begonnen.

Paket	Aufgabe	Aufwand(h)	Woche	Zugewiesen an	Fortschritt in %	Datum der Fertigstellung
Allgemein	Entscheidung Namespace	1	1	Alle	100	
Allgemein	Erstellen der Gradle-Konfiguration	6	1-3	Lucas	75	
objectproperties	Implementieren	4	1	Nicolas	100	23.06.16
parameter	Implementieren	5	1	Nicolas	100	23.06.16
graphmodel	Interfaces schreiben	2	1	Jonas F + Lucas	50	
graphmodel	Default-Klassen implementieren	8	1	Jonas F (+ Sven)	60	
graphmodel	Layout-Register schreiben	4	1	Jonas F	20	
graphmodel	Serialized-Klassen	2	1	Jonas F (+ Sven)	70	
sugiyama	Graph + Impl Interfaces + Vertex/Edge	9	1	Jonas M + Sven	10	
sugiyama	Mokups für einzelne Phasen	8	1	Jonas M + Sven	0	
sugiyama	Constraints implementieren	4	1	Jonas M + Sven	0	
sugiyama	LayoutAlgorithm	2	1	Jonas M + Sven	0	
sugiyama	CycleRemover (Phase 1) + Interface	7	1	Jonas M + Sven	0	
gui	GraphFactory	5	1	Nicolas	75	
gui	ParameterDialogCreator	6	1	Nicolas	100	23.06.16
plugin	Manager	11	1	Lucas	50	
plugin	Interfaces schreiben	3	1	Lucas	0	
gui	Generelle GUI(Incl. Basis-GraphView)	11	1-3	Nicolas	30	

Abbildung 4.2: Überarbeiteter Plan nach Woche 1 mit Fortschritt

4.1.1 Verzögerungen

Wie man in 4.3 sehen kann, hing das Projekt nach der ersten Woche bereits ca. 35 Stunden hinter dem geplanten Zustand hinterher. Dies lag neben der Tatsache, dass die erste Hälfte der Woche aufgrund von Erschöpfung aus der Entwurfsphase, weniger gearbeitet wurde, als auch daran, dass das Team, welches für die Implementierung des Sugiyama-Layoutalgorithmus eingeteilt wurde, relativ schnell auf konzeptionelle Probleme mit dem im Entwurf definierten Aufbau des SugiyamaGraphen und dessen Kanten/Knoten stieß(siehe 2.2).

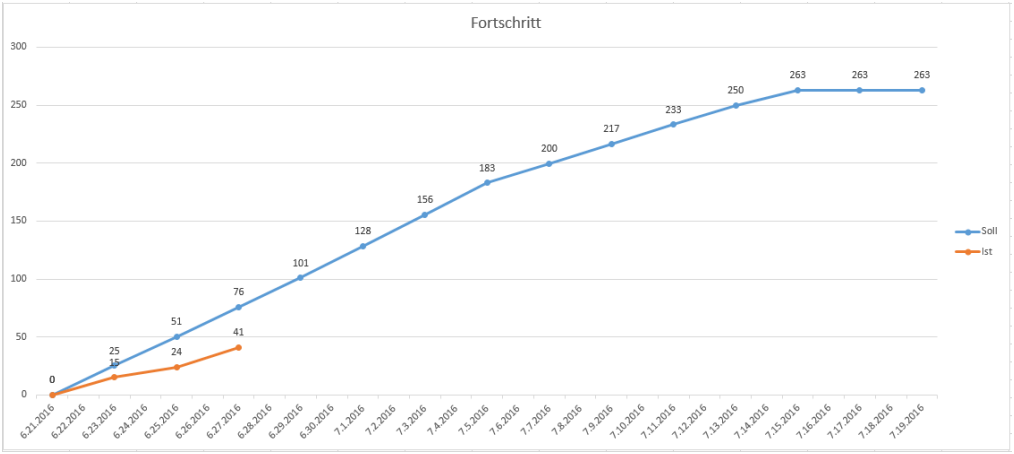

Abbildung 4.3: Fortschrittsdiagramm nach Woche 1

4.2 Woche 2

Das Ziel der Woche 2 war es, einen Graphen zu importieren und diesen, mithilfe von Mockups im Sugiyama-Algorithmus, in der Graphansicht anzuzeigen. Dazu wurde besonders auf die zeitnahe Fertigstellung des Imports und auf die grobe Implementierung des JOANA-Graph-Plugins mit allen dazugehörigen Klassen geachtet. Zudem wurde auch in den einzelnen Phasen des Sugiyama-Algorithmus große Fortschritte gemacht und einige wurden sogar abgeschlossen. Dadurch entstand der, wie in 4.5 zu sehen ist, sprunghafte Anstieg des Fortschritts, welcher fast den Soll-Wert erreichte.

Paket	Aufgabe	Aufwand(h)	Woche	Zugewiesen an	Fortschritt in %	Datum der Fertigstellung
Allgemein	Erstellen der Gradle-Konfiguration	6	1-3	Lucas	75	
graphmodel	Default-Klassen implementieren	8	1	Jonas F (+ Sven, Nicolas)	90	
graphmodel	Serialized-Klassen	2	1	Jonas F (+ Sven)	70	
sugiyama	Graph + Impl Interfaces + Vertex/Edge	9	1	Jonas M + Sven	70	
sugiyama	Mokups für einzelne Phasen	8	1	Jonas M + Sven	90	
sugiyama	CycleRemover (Phase 1) + Interface	10	1	Jonas M + Sven	70	
gui	GraphFactory	5	1	Nicolas	80	
plugin	Manager	11	1-3	Lucas	50	
gui	Generelle GUI(Incl. Basis-GraphView)	11	1-3	Nicolas	70	
sugiyama	LayerAssigner (Phase 2) + Interface	10	2	Jonas M + Sven	100	04.07.16
sugiyama	CrossMinimizer (Phase 3) + Interface	10	2	Jonas M + Sven	100	04.07.16
joana	Callgraph	5	2	Lucas	70	
joana	Methodengraph	5	2	Lucas	70	
joana	FieldAccess	3	2	Lucas	70	
joana	GraphModel	4	2	Lucas	100	
joana	Joana Vertex/Edge	4	2	Nicolas	100	
joana	Workspace	5	2	Nicolas(+ Lucas)	75	
joana	JoanaGraph	2	2	Nicolas	80	
joana	GraphBuilder-Klassen	12	2	Nicolas + Lucas (+ Jonas F)	50	
gui	Import und Export anstoßen + Dialoge	5	2	Nicolas + Jonas F	100	
gui	Laden der Plugins	6	2	Lucas	100	
import	Implementieren	11	2	Jonas F(+ Nicolas)	100	
sugiyama	VertexPositioner (Phase 4) + Interface	10	3	Jonas M + Sven	40	
sugiyama	EdgeDrawer (Phase 5) + Interface	10	3	Jonas M + Sven	10	

Abbildung 4.4: Implementierungsplan nach Woche 1 und 2

4.2.1 Verzögerungen

Aufgrund der im Entwurf definierten Generics im GraphModel entstanden häufig sehr unschöne Stellen im Code und Inkompatibilitäten. Diese zu reparieren oder funktionsfähig zu machen war häufig der Grund für mehr oder weniger unschöne Workarounds, die teilweise viel Arbeit in Anspruch nahmen. Deshalb wurde für die folgende Woche ein großes Refactoring angesetzt, welches die Generic-Struktur im GraphModel stark vereinfachen soll.

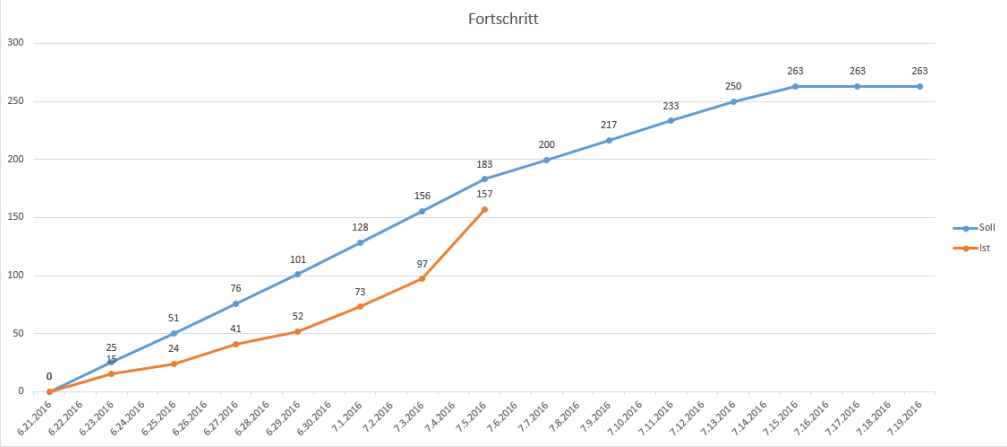

Abbildung 4.5: Fortschrittsdiagramm nach Woche 2

4.3 Woche 3

Da in Woche 3 die Oberfläche benutzbar war und man Graphen importieren und anzeigen konnte wurde neben dem in 4.2.1 beschriebenen Refactoring, viele Bugfixes und Verbesserungen vorgenommen, die ohne eine Anzeige nicht sichtbar waren. Genau deshalb wurden in 4.6 zwar weniger Aufgaben vollständig abgeschlossen, dafür bei vielen aber ein deutlicher Fortschritt erzielt.
In dieser Woche wurde auch die strikte Unterteilung in die Wochen 3 und 4 aufgehoben da es für das händische Testen von anderen Aufgabenteilen an der Oberfläche anbot diese Vorzuziehen.

Paket	Aufgabe	Aufwand(h)	Woche	Zugewiesen an	Fortschritt in %	Datum der Fertigstellung
Allgemein	Erstellen der Gradle-Konfiguration	6	1-3	Lucas	75	
graphmodel	Default-Klassen implementieren	8	1	Jonas F (+ Sven, Nicolas)	95	
sugiyama	Graph + Impl Interfaces + Vertex/Edge	9	1	Jonas M + Sven	85	
gui	GraphFactory	5	1	Nicolas	90	
plugin	Manager	11	1-3	Lucas	70	
gui	Generelle GUI(Incl. Basis-GraphView)	11	1-3	Nicolas	80	
joana	Methodengraph	5	2	Lucas	70	
joana	FieldAccess	3	2	Lucas	70	
joana	Workspace	5	2	Nicolas(+ Lucas)	75	
joana	JoanaGraph	2	2	Nicolas	90	
sugiyama	VertexPositioner (Phase 4) + Interface	10	3	Jonas M + Sven	50	
sugiyama	EdgeDrawer (Phase 5) + Interface	10	3	Jonas M + Sven	10	
joana	JoanaPlugin	6	3	Lucas	75	
joana	CallgraphLayout/Register/Option	7	3	Nicolas + Jonas F	80	
joana	MethodengraphLayout/Register/Option	7	3	Nicolas + Jonas F	40	
gui	GraphView(Navigation, Selection und Zoom)	10	3	Nicolas + Jonas F	90	
plugin	Filter	7	3	Lucas	0	
plugin	Workspace-/EntrypointOption	4	3	Lucas	100	
gui	StrukturView	4	4	Nicolas	90	
gui	InformationView	4	4	Nicolas	70	
gui	Kommandozeilenparameter akzeptieren	4	4	Jonas F	0	
export	Implementieren	7	4	Jonas F	70	

Abbildung 4.6: Implementierungsplan nach Woche 3

4.3.1 Verzögerungen

Nach dem Sprint zum Ende der zweiten Woche wurde, wie in 4.7 zu sehen ist, zunächst weniger gearbeitet. Ab Mitte der Woche wurde aber wieder der normale Arbeitsrythmus aufgenommen und es kam zu keinen weiteren Verzögerungen.

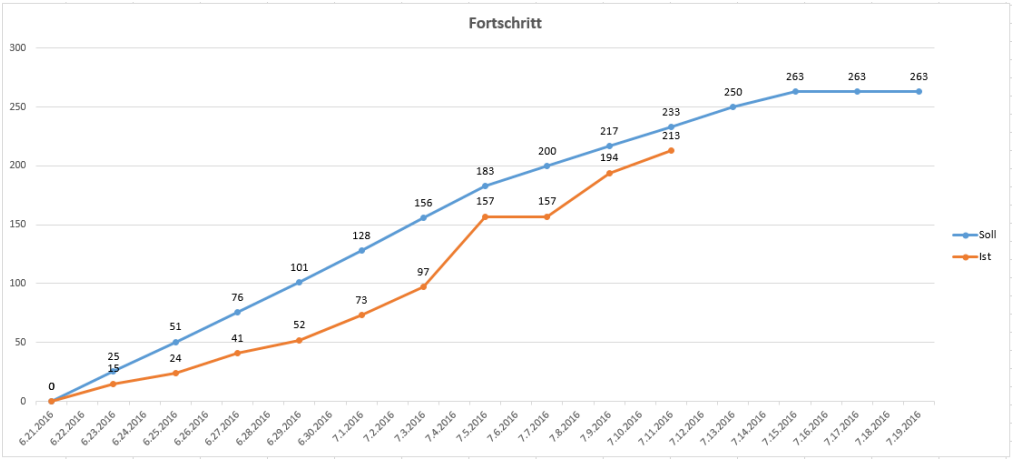

Abbildung 4.7: Fortschrittsdiagramm nach Woche 3

4.4 Woche 4

In Woche 4 wurden die vielen in Woche 3 begonnenen Aufgaben fertiggestellt. Des weiteren wurden Aufgaben die noch nicht begonnen waren, wie Filter und Kommandozeilenparameter akzeptieren, schnellstmöglich eingebaut. Die Arbeit an den Filtern und dem Sugiyama-Algorithmus zog sich bis zum letzten Tag vor Fertigstellungstermin.

4.4.1 Verzögerungen

Die vielen weit fortgeschrittenen Aufgaben in Woche 3 die bereits in der Oberfläche benutzbar waren, zeigten verschiedene kleinere Fehler auf, die zuvor nicht sichtbar waren. Diese Fehler zu reparieren und die Tatsache das die Arbeit am Implementierungsheft begonnen wurde, sorgte für leichte Verzögerungen im Implementierungsfortschritt.

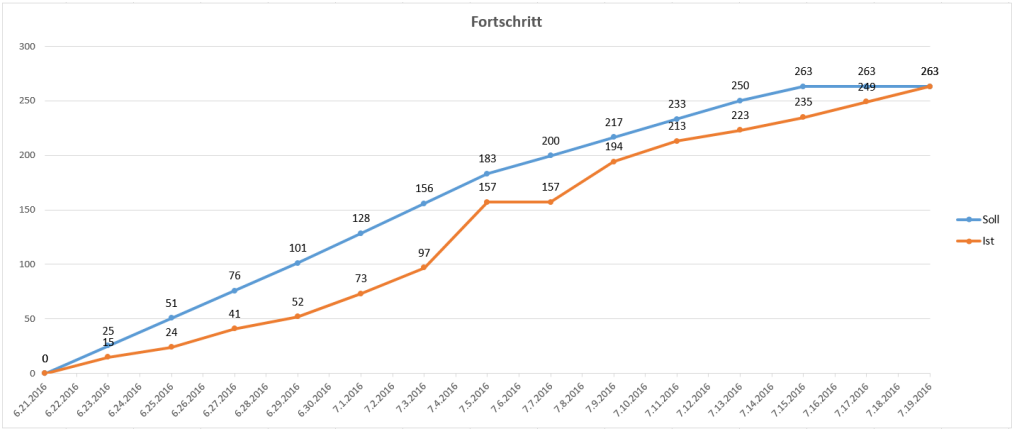

Abbildung 4.8: Fortschrittsdiagramm nach Woche 4

4.5 Rückblick

Am Ende der 4. Woche steht rückblickend fest, dass der Implementierungsplan zwar von seiner Struktur und den generellen zeitlichen Abschätzungen gut war, der Arbeitsaufwand aber in vielen Aufgabenbereichen stark unterschätzt wurde. Der Grund war häufig, das im Entwurf gemachte Designentscheidungen nicht tief genug durchdacht waren und im Laufe der Entwicklung angepasst oder komplett neu erdacht werden mussten. Solche Anpassungen haben an verschiedenen Stellen sehr viel Zeit in Anspruch genommen, welche nicht im Implementierungsplan vermerkt wurde. Diese Verzögerungen spiegeln sich besonders in der flachen Fortschrittskurve in den ersten zwei Wochen dar und haben die gesamte Entwicklung zurückgeworfen.
Man erkennt in 4.8 das die Ist-Kurve nach der ersten Woche fast genau so stark, manchmal sogar stärker, steigt als die Soll-Kurve. Trotzdem wurde durch die nötigen Anpassungen bedeutend mehr Zeit gebraucht als angegeben wurde. Deshalb waren auch die 16 wöchentlichen Arbeitsstunden pro Person, mit welchen die Soll-Kurve berechnet wurde, bei weitem nicht ausreichend um die Entwicklung in den vier Wochen möglichst vollständig abzuschließen. Es wurden auch die fünf Tage am Ende, die als zeitlicher Puffer eingeplant waren vollständig benötigt.

5 Unit Tests

5.1 Plugin

5.1.1 PluginManagerTest

1. **Test:** testPluginLoad()
   Aufgabe: Testet ob der *PluginManager* alle Plugins läd.

5.2 Joana

5.2.1 JoanaGraphTest

Importiert eine größere Anzahl von GraphML Dateien und führt mehrere Tests auf die Menge von GraphModels aus.

1. **Test:** callGraphSizeTest()
   Aufgabe: Testet ob in jedem Model, die Anzahl der Knoten im CallGraph der Anzahl der MethodenGraphen entspricht.
2. **Test:** collapseTest()
   Aufgabe: Testet für einen zufälligen MethodenGraphen das Kollabieren und Ausklappen von einer Knotenmenge.
3. **Test:** randomSymmetricCollapseTest()
   Aufgabe: Kollabiert in einem zufälligen MethodenGraphen mehrfach Knoten, klappt sie in umgekehrter Reihenfolge wieder aus und testet auf ob die Struktur (mithilfe Adjezenzmatrizen) mit der Struktur zu Beginn des Tests übereinstimmt.
4. **Test:** randomAssymmetricCollapseTest()
   Aufgabe: Kollabiert zuerst mehrfach Knoten, klappt sie dann in zufälliger Reihenfolge wieder aus. Testet am Ende wieder auf Gleichheit der Struktur.
5. **Test:** randomMixedCollapseTest()
   Aufgabe: Kollabiert und klappt Knoten in zufälliger Reihenfolge aus. Klappt nach einer festgelegten Anzahl von Kollapsen alle Knoten wieder aus. Testet wieder auf Gleichheit der Struktur.

5.3 Sugiyama

5.3.1 CycleRemoverTest

1. **Test:** testSimpleCycle()
   Aufgabe: Erstellt einen Testgraphen aus drei Knoten die jeweils mit einem anderen Knoten über eine gerichtete Kante verbunden sind, sodass ein Kreis entsteht. Diesen übergibt es an einen *SugiyamaGraph*. Dann ruft es auf dem Cycle remover *removeCycles()* mit dem *SugiyamaGraph* als Parameter und testet danach ob der Graph azyklisch ist.
2. **Test:** testDoubleCycle()
   Aufgabe: Erstellt einen Testgraphen aus vier Knoten und fünf gerichteten Kanten. Die Kanten sind so miteinander verknüpft, dass zwei Zykel entstehen. Ein kleiner Zykel drei Knoten beinhaltender und ein großer Zykel, der alle vier Knoten und damit auch den kleineren Zykel enthaltenden. Diesen übergibt es an einen *SugiyamaGraph*. Dann ruft es auf dem Cycle remover *removeCycles()* mit dem *SugiyamaGraph* als Parameter und testet danach ob der Graph azyklisch ist.
3. **Test:** RandomGraphsTest()
   Aufgabe: Erstellt zwanzig zufällige, zyklische *SugiyamaGraphen*. Diese bestehen beim n-ten Graphen aus aus 2n Knoten und haben eine Kantendichte von 0.95^n. Dann ruft es auf dem Cycle remover *removeCycles()* mit dem *SugiyamaGraph* als Parameter und testet danach ob der Graph azyklisch ist.
4. **Test:** SelfLoopTest()
   Aufgabe: Erstellt einen Testgraphen aus einem Knoten mit einer gerichteten Kante als Selbstschleife. Diesen übergibt es an einen *SugiyamaGraph*. Dann ruft es auf dem Cycle remover *removeCycles()* mit dem *SugiyamaGraph* als Parameter und testet danach ob diese Kante umgedreht wurde.

5.3.2 LayerAssignerTest

1. **Test:** assignLayers()
   Aufgabe: Erstellt einen azyklischen Testgraphen aus fünf Knoten, welche die korrekte Ebene für die Knoten angeben. Diese werden mit fünf Kanten verbunden. Diesen übergibt es an einen *SugiyamaGraph*. Dann ruft es auf dem LayerAssigner *assignLayers()* mit dem *SugiyamaGraph* als Parameter und testet mithilfe der Labels, ob die Knoten den richtigen Schichten zugewiesen wurden.
2. **Test:** LayerAssignerTest2()

Aufgabe: Erstellt einen azyklischen Testgraphen aus sieben Knoten, welche die korrekte Ebene für die Knoten angeben. Diese werden mit zehn Kanten verbunden. Diesen übergibt es an einen *SugiyamaGraph*. Dann ruft es auf dem LayerAssigner *assignLayers()* mit dem *SugiyamaGraph* als Parameter und testet mithilfe der Labels, ob die Knoten den richtigen Schichten zugewiesen wurden.

5.3.3 CrossMinimizerTest

1. **Test:** singleRandomTest()
   Aufgabe: Erstellt einen zufälligen, zyklischen, topologisch gelayerten *SugiyamaGraphen*. Diese bestehen aus 20 Knoten, die jeweils 2-8 Kanten haben. Dann ruft es auf dem CrossMinimizer *minimizeCrossings()* mit dem *SugiyamaGraph* als Parameter und testet danach ob die Anzahl der Kreuzungen verringert wurde oder zumindest gleich bleibt.
2. **Test:** randomTests()
   Aufgabe: Erstellt 20 zufälligen, zyklischen, topologisch gelayerten *SugiyamaGraphen*. Diese bestehen beim n-ten durchlauf aus $n + 10$ Knoten, die jeweils 3-4 Kanten haben. Dann ruft es auf dem CrossMinimizer *minimizeCrossings()* mit dem *SugiyamaGraph* als Parameter und testet danach ob die Anzahl der Kreuzungen verringert wurde oder zumindest gleich bleibt.
3. **Test:** performanceTest()
   Aufgabe: Erstellt 20 zufälligen, zyklischen, topologisch gelayerten *SugiyamaGraphen*. Diese bestehen aus 75 Knoten, die jeweils 2-8 Kanten haben. Dann ruft es auf dem CrossMinimizer *minimizeCrossings()* mit dem *SugiyamaGraph* als Parameter und testet danach ob die Anzahl der Kreuzungen verringert wurde oder zumindest gleich bleibt. Dabei geht es besonders um die Zeit die der Test benötigt.
4. **Test:** hugeTest()
   Aufgabe: Erstellt einen zufälligen, zyklischen, topologisch gelayerten *SugiyamaGraphen*. Dieser besteht aus 250 Knoten, die jeweils 2-6 Kanten haben. Dann ruft es auf dem CrossMinimizer *minimizeCrossings()* mit dem *SugiyamaGraph* als Parameter und testet danach ob die Anzahl der Kreuzungen verringert wurde oder zumindest gleich bleibt. Dabei geht es besonders um die Zeit die der Test benötigt.

5.3.4 VertexPositionerTest

1. **Test:** positionVertices()
   Aufgabe: Erstellt einen Graphen, den VertexPositioner auf diesen aus und prüft ob dabei Exceptions geworfen werden.

5.3.5 EdgeDrawerTest

1. **Test:** compileTest()
   Aufgabe: Erstellt einen Graphen, führt alle Algorithmus-Schritte auf diesen aus und prüft ob dabei Exceptions geworfen werden.

5.3.6 SugiyamaLayoutAlgorithmTest

Für jeden Test wird ein kompletter Sugiyama-Layout-Algorithmus angewendet.

1. **Test:** testSmallGraph()
   Aufgabe: Testet für einen Graphen mit vier Knoten und fünf Kanten ob der gesamte Algorithmus ohne Fehler durchläuft.
2. **Test:** testRandomGraph()
   Aufgabe: Testet für drei zufällige Graphen ob der gesamte Algorithmus ohne Fehler durchläuft.