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

Nach der Implementierung von Graph von Ansicht ist nun die Qualitätssicherung an der Reihe. Ziel ist es, Fehler im Programm durch verschiedene Vorgehensweisen zu finden und diese anschließend zu beheben. Ziel ist es auch, durch sowohl zufällige als auch wohl überlegte Testfälle und der Überprüfung dieser auf Korrektheit, die richtige Funktionsweise des Programmes ein Stück weit mehr garantieren zu können. Je mehr Testfälle funktionieren, desto mehr mögliche Fehlerquellen können wahrscheinlicher ausgeschlossen werden.
Da das Programm nach der Implementierung nicht komplett fertig war und es noch einige Stellen gab, an denen aus Zeitgründen nicht weiter gearbeitet werden konnte und diese somit auch teilweise nicht vollständig funktionsfähig waren, wurden diese zuallererst fertig implementiert. Noch zu erledigen waren in diesem Projekt: das korrekte Darstellen von Feldzugriffen (eine bestimmte Teilmenge von Knoten und Kanten des Graphen), dessen Kanten nicht richtig gezeichnet wurden, zeitliche Zusicherungen, welche versprochen wurden (1000 Knoten + 4000 Kanten unter 2 Minuten, 500 Knoten + 2000 Kanten unter einer Minute), ein Programmabsturz bei falschen Kommandozeilenargumenten soll verhindert werden.
Graph von Ansicht ist ein Graphviewer, seine Hauptaufgabe besteht darin, gegebene Graphen zu layouten. Deswegen liegt das Hauptaugenmerk des Testens auf dem Layoutalgorithmus. Die korrekte Funktionsweise des Algorithmus kann aufgrund der hohen Anzahl an möglichen Eingaben nicht garantiert werden, auch bietet das Programm durch eine eingebaute Filterfunktion für Knoten und Kanten eine Vielzahl an Möglichkeiten den Graphen nochmals zu modifizieren bevor der Algorithmus erneut ausgeführt werden kann. Deswegen wurden viele Tests erstellt, die vor allem Randfälle (zum Beispiel ein Graph ohne Knoten oder Kanten, mit einem Selbstzyklus (Kante in sich selbst), etc.) abdecken. Ebenso wird die Benutzbarkeit des Programmes getestet über das Testen des Programmes durch Personen, die sowohl von dem Thema als auch der Funktionsweise des Programmes unwissend sind. Auffälligkeiten hinsichtlich der Handhabung und Navigation in dem Programm werden angepasst, sodass sich auch fachfremde Personen in der Menüführung zurecht finden.

2 Bugs und Reparatur

#1	**Fehlersymptom:**	Graphische Hervorhebung von FieldAccessen durch eine farbige Box im Hintergrund wird nicht angezeigt.
	Fehlergrund:	Die Struktur im Hintergrund ist leicht fehlerhaft und in der Erstellung der Box gab es einen Bug.
	Fehlerbehebung:	Die Struktur im Hintergrund ist erneuert worden und der Bug beim Erstellen des Hintergrunds ist behoben.
#2	**Fehlersymptom:**	Einige Kanten werden übereinanderliegend gezeichnet.
	Fehlergrund:	Dummy Vertices werden übereinander platziert, da der Codeabschnitt zur Überprüfung von Kantenüberlagerungen fehlerhaft ist. Außerdem werden die Kanten die mehrere Ebenen überdecken nicht immer weit genug verschoben. Auch Mehrfachzuweisungen von Knoten in zu begradigenden Abschnitten treten auf.
	Fehlerbehebung:	Ein fehlerhafter Vergleich bei der Überprüfung von Kantenschnitten im VertexPositioner worde korrigiert. Er kann nun auch mit 1 breiten Kanten arbeiten. Die Verschiebung der Kanten wird nun so oft ausgeführt, bis sich Kanten nicht mehr überdecken. Der schwerwiegender Fehler, welcher zu den Mehrfachzuweisungen führte wurde auch behoben. Hier hat ein rekursiver Teilalgorithmus die Ergebnisse verdoppelt.
#3	**Fehlersymptom:**	Kanten, die von außen in einen FieldAccess hinein oder von einem FieldAccess nach außen gehen, werden nur bis an die Box des FieldAccesses gezeichnet, und nicht bis zu dem dazugehörigen Knoten.
	Fehlergrund:	Nach dem Layouten des kompletten Graphen werden die Kanten nicht von der Box nach innen weiter gezeichnet.
	Fehlerbehebung:	Alle Kanten innerhalb eines FieldAccesses werden nun nach dem Layouten des kompletten Graphen nochmals neu gezeichnet. Sowohl die alten, als auch die neuen. Die Positionen der Knoten und die außerhalb liegende Kanten bleiben dabei gleich.
#7	**Fehlersymptom:**	Kanten, welche keine Ebene überspannen werden nie begradigt
	Fehlergrund:	Diese Kanten werden beim Suchen nach Segmenten (Kantenketten, die im VertexPositioner später begradigt werden) nicht beachtet
	Fehlerbehebung:	Diese Kanten werden nun beim Starten des VertexPositioner ebenfalls hinzugefügt. Im Zuge dieser Korrektur wurde es außerdem ermöglicht die Endpunkte von Kanten, welche eine oder mehr Ebenen überspringen, zu den aus ihnen enstehenden Segmenten hinzuzufügen.

#8	**Fehlersymptom:**	Kollabierte Knoten in einem FieldAccessGraph verschwinden bei Ihrer Erstellung.
	Fehlergrund:	Bei dem Layouten von MethodGraphen werden alle FieldAccesse betrachtet. Unter anderem auch FieldAccesse die teilweise kollabiert wurden. Das führt dazu, dass Annahmen an den FieldAccess gemacht werden, welche nicht für teilweise kollabierte Knoten gilt.
	Fehlerbehebung:	Es werden nun nur noch FAs welche vollständig ausgeklappt sind im MethodGraphLayout betrachtet.
#9	**Fehlersymptom:**	Beim Öffnen eines Graphens über die Kommandozeile wird nicht das Default Layout des ausgewählten Workspace übernommen.
	Fehlergrund:	Es wird immer der LayoutSelectionDialog aufgerufen, obwohl ein Workspace als Kommandozeilenargument angegeben wurde.
	Fehlerbehebung:	Der Aufruf des Dialogs wurde gelöscht und das Default Layout aus dem Workspace wird angewendet.
#10	**Fehlersymptom:**	Keine Informationen oder Warnung über falsch eingegebene Kommandozeilenparameter.
	Fehlergrund:	Keine explizite Überprüfung der Eingaben. Es wurde angenommen, dass der User alle Eingaben richtig macht.
	Fehlerbehebung:	Hinzufügen von expliziten Überprüfungen und Anzeigen von Fehlermeldungen mit hilfreicher Information.
#11	**Fehlersymptom:**	Falls das Joana-Workspace für eine generische GraphML-Datei ausgewählt wird, kommt keine Warnung und es passiert nichts.
	Fehlergrund:	Keine Überprüfung ob ein GraphBuilder für diesen Graphentyp existiert.
	Fehlerbehebung:	Der Importer überprüft ob der GraphBuilder gesetzt wird. Falls nicht wird eine Fehlermeldung angezeigt.
#12	**Fehlersymptom:**	Der Graph springt beim verschieben mit der Maus in der Graphview zu Beginn ein Stück nach unten.
	Fehlergrund:	Für die Berechnung der Mausposition werden die Koordinaten des Fensters genutzt.
	Fehlerbehebung:	Die Mausposition wird nun direkt aus dem Mausevent, welches durch das Ziehen aufgerufen wird, berechnet.
#13	**Fehlersymptom:**	Es gibt keine Option zur Einstellung ob eine Abbruchgrenze beim Kreuzungsminimieren in Abhängigkeit von den vorhandenen Kreuzungen gewählt werden soll.
	Fehlergrund:	Diese Option wird in den CrossMinimizer Einstellungen nicht bereitgestellt.
	Fehlerbehebung:	Die Option wurde als Checkbox hinzugefügt und wird während der Kreuzungsminimierung beachtet.
#17	**Fehlersymptom:**	Das Filtern von Knoten bestimmten Types (z.B. EXPR) im MethodGraphLayout führt zu einer NullPointerException.

	Fehlergrund:	Für das Layouten der FieldAccesse und des ganzen MethodGraphen wurde der selbe Layoutalgorithmus, mit den selben Constraints gewählt. Diese Constraints enthielten auch die gefilterte Knoten, welche dann nicht für den SugiyamaAlgorithmus erreichbar waren.
	Fehlerbehebung:	Separate Instanzen des SugiyamaAlgorithm für FieldAccesse und MethodGraphen werden benutzt. Die Instanzen für die FieldAccesse enthalten nicht die Constraints des MethodGraphen.
#22	**Fehlersymptom:**	Die alte Strukturansicht kann nach dem Import einer neuen Datei nicht zurückgebracht werden.
	Fehlergrund:	Alte Graphen werden nach einem neuen Import nicht geschlossen, obwohl die Anwendung nur eine einzelne Datei unterstützt.
	Fehlerbehebung:	Bereits offene Graphen werden nach erfolgreichem Import nun geschlossen.
#23	**Fehlersymptom:**	Ähnlich wie in Bug 22 wird versucht nach einem Import über den noch offenen alten Callgraphen einen Methodengraphen zu öffnen.
	Fehlergrund:	Siehe Bug 22
	Fehlerbehebung:	Siehe Bug 22
#24	**Fehlersymptom:**	Programm stürzt ab, falls der eingegebene Dateipfad über den Kommandozeilenparameter - -in keinen Punkt enthält.
	Fehlergrund:	Falls Dateipfad keinen Punkt erhält gibt Java Methode lastIndexOf('.') als Ergebnis -1 zurück. Dieses Ergebniss wird bei substring() als ungültiger Index verwendet was zum Abstürz führt.
	Fehlerbehebung:	Vor Aufruf der Methode substring() wird überprüft ob der Dateipfad einen Punkt enthält. Falls nicht wird eine Fehlermeldung ausgegeben.
#25	**Fehlersymptom:**	Es wird ein Fehler geworfen, wenn man die ENTR Knoten filtert.
	Fehlergrund:	Der VertexPositioner benötigt mindestens ein Vertex pro Graph.
	Fehlerbehebung:	Der VertexPositioner überprüft nun am Anfang, ob der Graph leer ist und läuft in diesem Fall nicht durch.
#26	**Fehlersymptom:**	Das Layout beim erneuten Layouten des selben Graphen ist nicht stabil. Das heißt Knoten nehmen andere Positionen an und Kanten haben unterschiedliche Pfade, ohne dass es eine Änderung am Graphen vorgenommen wird. Dieses Ergebnis kann auf zwei verschiedene Weisen erzeugt werden: Entweder durch mehrmaliges Öffnen eines Graphen, oder durch Öffnen des Filterdialoges und Schließen ohne die Auswahl der Filter zu ändern.
	Fehlergrund:	Das Programm stützt sich in den meisten Bereichen bei der Abspeicherung von Daten bzgl. der Graphen auf unsortierte Sets. Daher kann ein gleiches Layout bei mehrmaligem Layouten nicht garantiert werden, ohne ein größeres Refactoring vorzunehmen, welches eine Sortierung der Daten, wie z.B. Knoten nach deren ID vornimmt.

	Fehlerbehebung:	Der Graph wird nicht erneut gelayoutet nachdem der Filterdialog geöffnet und wieder geschlossen wurde, ohne dass Änderungen an der Auswahl der Filter gemacht wurden.
#27	**Fehlersymptom:**	Text in exportierter SVG-Datei ist größer als die Textboxen der Knoten.
	Fehlergrund:	Default Textgröße in SVG ist größer als Default Textgröße innerhalb der GUI. Die Größe der Textboxen richten sich jedoch nach der Textgröße der GUI.
	Fehlerbehebung:	Hinzufügen einer expliziten Textgröße zur SVG-Datei, welche der Textgröße der GUI entspricht.
#28	**Fehlersymptom:**	Kanten innerhalb eines FieldAccess laufen ggf. aus der Box hinaus.
	Fehlergrund:	Kantenläufe werden bei der Berechnung der Größe der Box nicht berücksichtigt.
	Fehlerbehebung:	Kantenläufe werden nun bei der Berechnung der Größe der Box berücksichtigt.
#30	**Fehlersymptom:**	Kanten laufen durch Knoten durch und manchmal von unten rein und verbinden sich dann oben auf dem Knoten.
	Fehlergrund:	Der Kantenzeichner geht davon aus, dass Kanten immer von oben nach unten verlaufen, also der source-Knoten sich immer oberhalb des target-Knoten befindet. Durch relative- und absolute-layer-constraints kann es aber vorkommen, dass miteinander verbundene Knoten auf dem selben layer liegen, weswegen die Kanten teilweise unvorhersehbar durch Knoten gezeichnet werden.
	Fehlerbehebung:	Kanten zwischen Knoten auf dem selben layer werden nun gesondert gezeichnet, und zwar unterhalb des layers.
#31	**Fehlersymptom:**	Beim Zoomen wird nicht vom Mauszeiger weg oder zum Mauszeiger hin gezoomt. Der Fokus und Sichtbereich springt herum.
	Fehlergrund:	Die GraphView wird durch das Zoomen vergrößert, dadurch gibt es von den äußeren Oberflächenelementen, die um die GraphView gelegt sind, Interferenzen, welche das Springen und unfokussierte Zoomen ausgelöst haben.
	Fehlerbehebung:	Die Ansicht des Graphen besteht nun aus 4 ineinander verschachtelten Panes, wobei das innerste die GraphView und das äußerste ein ScrollPane ist. Vergrößert wird beim Zoomen nun das Pane welches direkt um der GraphView liegt. Das Pane darum passt sich per Listener immer an die Größe der beiden inneren Panes an. Dadurch entstehen keinen Interferenzen mehr und anderes ungewolltes Verhalten tritt auch nicht mehr auf.
#33	**Fehlersymptom:**	Wenn Knoten aus einem FieldAccess kollabiert wird, wird eine Exception geworfen. Das Programm stürzt nicht ab, aber der Graph wird nicht neu gelayoutet.

	Fehlergrund:	Beim Neu-Layouten der FieldAccesse wurden alle FieldAccesse und alle von ihnen enthaltenen Knoten betrachtet. Damit auch die FieldAccesse, die kollabiert wurden.
	Fehlerbehebung:	FieldAccesse, die nicht vollständig im Graphen enthalten sind, werden ignoriert.
#34	**Fehlersymptom:**	Wenn ein Knoten, der zu einem Constraint gehört, kollabiert wird, wird eine Exception geworfen. Das Programm stürzt nicht ab, aber der Graph wird nicht neu gelayoutet.
	Fehlergrund:	Die Constraints mit den kollabierten Knoten wurden dennoch erstellt. Wenn der SugiyamaAlgorithmus versucht den Knoten aus dem Graphen zu holen wird dieser nicht zurück gegeben, da er nicht vorhanden ist.
	Fehlerbehebung:	Es werden keine Constraints erstellt, welche Knoten enthalten die nicht im Graph enthalten sind. (Kollabiert oder gefiltert)
#40	**Fehlersymptom:**	Die Richtung mancher Kanten zwischen zwei Knoten ändert sich manchmal nach neuem Importieren oder neuem Layouten des Graphen.
	Fehlergrund:	Im Sugiyama werden notwendigerweise zuallererst Kanten gedreht, die Zykel im Graph verursachen. Der Kantenzeichner im letzten Schritt des Sugiyama hat nun nur die gedrehten Kanten, die sich über ein Layer erstrecken in ihrer Richtung angepasst. Kanten, die über mehrere Layer gingen wurden ignoriert, da diese in einem anderen Set lagen.
	Fehlerbehebung:	Der Kantenzeichner passt nun durch Berücksichtigung des Sets der sich über mehrere Layer erstreckenden Kanten, von diesen ehemals gedrehte Kanten in ihrer Richtung an.
#41	**Fehlersymptom:**	In den Javadoc Kommentare existieren veraltete Parameter.
	Fehlergrund:	Beim Ändern der Methodenparameter wurden die Javadoc Kommentare vernachlässigt.
	Fehlerbehebung:	Es wurde nach fehlerhaften Javadocs gesucht und diese aktualisiert.
#42	**Fehlersymptom:**	ArrayIndexOutOfBoundsException im VertexPositioner nach dem Filtern von NORM Knoten und dann EXPR Knoten
	Fehlergrund:	Nach einem commit verschwand dieser Fehler und äußert sich durch die selben Merkmale wie Bug 44, siehe daher 44.
	Fehlerbehebung:	siehe Bug 44
#44	**Fehlersymptom:**	Nach dem Filtern von NORM Knoten aus einem Graphen, tritt eine NullPointerException im EdgeDrawer auf.
	Fehlergrund:	Der Graph enthält nach dem Filtern einen isolierten Knoten.
	Fehlerbehebung:	Der EdgeDrawer schaut nun nach isolierten Knoten, bevor er auf Knoten zugreift.
#45	**Fehlersymptom:**	FieldAccess Boxen werden in der exportierten SVG-Datei nicht angezeigt.

	Fehlergrund:	Die FieldAccess Boxen werden nicht zum serialisierten Graphen hinzugefügt.
	Fehlerbehebung:	Die FieldAccess Boxen werden nun als zusätzliche Knoten im serialisierten Graphen gespeichert.
#46	**Fehlersymptom:**	Exportierte SVG-Datei wird in manchen SVG-Viewer nicht komplett angezeigt.
	Fehlergrund:	Einige SVG-Viewer können die Standard Größe 100% einer SVG-Datei nicht korrekt interpretieren.
	Fehlerbehebung:	Die komplette Größe eines Graphens wird berechnet und als feste Werte in die SVG-Datei geschrieben.
#47	**Fehlersymptom:**	Große Graphen werden sehr langsam gelayotet
	Fehlergrund:	Es fehlt eine Möglichkeit vom Benutzer Einfluss auf die Parameter des Layoutvorgangs zu nehmen.
	Fehlerbehebung:	Einige dieser Einstellungen sind nun vorhanden. So kann der Benutzer auswählen ob er eine prozentuale Verbesserungsgrenze verwenden möchte. Diese Einstellung kostet viel Berechnungszeit, kann aber zu besseren Ergebnissen führen. Diese Möglichkeit Einstellungen am Algorithmus vorzunehmen sollte nach dem Release noch erweitert werden.
#51	**Fehlersymptom:**	Vertices werden hinter anderen Vertices positioniert
	Fehlergrund:	Der VertexPositioner positioniert Vertices, die auf einer Ebene liegen aber durch eine Kante verbunden sind fehlerhaft
	Fehlerbehebung:	Im VertexPositioner werden Vertices, welche auf einer Ebene liegen und durch eine Kante verbunden sind nun nicht mehr versucht auf eine Linie zu bringen.
#52	**Fehlersymptom:**	Im Callgraph finden sich Kantenkreuzungen
	Fehlergrund:	Die Anzahl der CrossMinimizer durchläufe beim callgraph war zu gering. Dies liegt vor allem an einer prozentualen Verbesserungsgrenze für Callgraphen, diese macht es wahrscheinlich, dass nur ein einzelner Durchlauf des CrossMinimizers stattfindet.
	Fehlerbehebung:	Der Callgraph wird nun per Default mindestens 10 mal durchgeführt. Einer prozentualen Grenze für die Verbesserung wird nun nicht mehr verwendet.
#53	**Fehlersymptom:**	Das Programm stürzt ab falls beim Import eines Joana Graphen ein falscher Knotentyp eingelesen wird.
	Fehlergrund:	Alle mögliche Joana Knotentypen werden in einem Enum gespeichert. Falls der Typ nicht im Enum existiert wird eine IllegalArgumentException geworfen.
	Fehlerbehebung:	Die Exception wird nun im Importer abgefangen und als Fehlermeldungen dem Nutzer angezeigt.
#57	**Fehlersymptom:**	Im Callgraph werden keine Schleifen (Kante, welche einen Knoten mit sich selbst verbindet) angezeigt.

Fehlergrund: Beim Import-Vorgang werden, beim Erstellen des Callgraphen, Schleifen ignoriert. Dies liegt daran, dass Source- und Target-Knoten einer Schleife im selben MethodGraph liegen und somit diese Kante zum MethodGraph hinzugefügt wird und nicht zum CallGraph.

Fehlerbehebung: Falls eine Kante vom „EdgeKind CL“ ist (welche Call-Kanten definieren), wird diese zum CallGraph hinzugefügt.

3 Durchgeführte Tests

3.1 Globale Testfälle aus dem Pflichtenheft

/T010/: Import und Darstellung von einem JOANA Graphen
Anmerkungen: -

/T020/: Öffnen eines JOANA-Methodengraphen
Anmerkungen: -

/T030/: Selektieren mehrerer Knoten und Kanten
Anmerkungen: Da es sich während der Implementierung als wenig nützlich erwiesen hat, wurde das selektieren von Kanten nicht implementiert. Deswegen ist es in diesem Test auch nicht möglich, Kanten zu selektieren.
Es werden außerdem bei mehreren ausgewählten Knoten weder eine Statistik noch eine Information zu diesen angezeigt. Lediglich von einem einzigen Knoten werden Informationen angezeigt.

/T040/: Navigation
Anmerkungen: Das Verschieben des Sichtfeldes geschieht nach der Implementierung nicht per Mittelmaus-klick halten und ziehen, sondern über Strg + Rechts-klick gedrückt halten und ziehen mit der Maus.

/T050/: Constraint zu Knoten eines geladenen Graphen hinzufügen
Anmerkungen: Durch Verschieben des Kriteriums der manuell hinzufügbaren Constraints zu Gruppen in die Wunschkriterien (siehe Pflichtenheft: 4.2 Wunschfunktionen, /FA300/ Constraints hinzufügen), wurde dies in der Implementierungsphase nicht hinzugefügt. Es ist daher hier lediglich möglich, Gruppen zu erstellen (bis einschließlich Punkt 4 des Tests).

/T060/: Filtern von Kanten
Anmerkungen: Die Menüführung zum Erreichen der Filter von Kanten ist „Other → Edit Filter“, danach klick auf Edges und dann hinzufügen eines Haken zum filtern dieser Kante, anstatt „Editieren → Filter anpassen“mit anschließendem klick auf Joana und entfernen von Haken.

/T070/: Export von einem geladenen JOANA-Graphen als SVG
Anmerkungen: Die Menüführung zum exportieren des geladenen JOANA-Graphen gechieht über „File → Export“, anstatt „Datei → Export → SVG“. Da nur ein Export-Format zur Auswahl steht, wurde auf den letzten Schritt der geplanten Menüführung verzichtet.

3.2 JUnit Tests

3.2.1 Sugiyama Plugin

3.2.1.1 CycleRemover

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.

3.2.1.2 LayerAssigner

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.

3.2.1.3 CrossMinimizer

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.

3.2.1.4 SugiyamaLayoutAlgorithm

1. **Test:** testSmallGraph()
   Aufgabe: Erstellt einen kleinen Graphen mit 4 Knoten und 5 Kanten, auf diesem wird dann der Sugiyama Algorithmus ausgeführt und überprüft ob dieser ohne Exceptions durchläuft.
2. **Test:** testIsolatedGraph()

Aufgabe: Erstellt einen kleinen Graphen mit 4 Knoten und 2 Kanten, sodass zwei voneinander isolierte Teilgraphen entstehen. Auf diesem wird dann der Sugiyama Algorithmus ausgeführt und überprüft ob dieser ohne Exceptions durchläuft.

3. **Test:** testRandomGraph()
   Aufgabe: Erstellt 3 zufällige Graphen mit 100 Knoten und über 400 Kanten, der auch Selbstschleifen enthalten kann. Auf diesem wird dann der Sugiyama Algorithmus ausgeführt und überprüft ob dieser ohne Exceptions durchläuft. Dieser Test wird auch zum Performance überprüfen verwendet.

3.2.2 SVG-Export

1. **Test:** fileEndingTest()
   Aufgabe: Testet ob der *SvgExporter* die untersützten Datei Formate korrekt zurückgibt.
2. **Test:** exportSucessfullTest()
   Aufgabe: Testet ob beim Exportieren eine Fehlermeldung geworfen wird.

3.3 Hallway Usability Testing

Hier wird das Programm durch fachfremde Probanden, die das Programm nicht kennen, getestet.
Es wird ihnen erst die Funktionsweise und des Programmes erläutert, anschließend dürfen sie frei testen.
Alle hierbei auftretenden Auffälligkeiten, seien es mögliche Fehler oder Verbesserungsmöglichkeiten in der Bedienbarkeit, unerwünschte Nebeneffekte oder gar schwerwiegende Fehler, die das Programm abstürzen lassen, werden notiert.

Proband 1: Datum 16.08.2016, commit hash: 6b06dd0
Auffälligkeiten: Es traten keine unerwarteten Fehler im Programm auf.
Es wurde vor allem mit collapse gearbeitet, den Menüpunkten zm Filtern, neu importieren, exportieren wurde kaum Aufmerksamkeit geschenkt.
Der Proband war mit der Menüführung etwas überfordert, da es dort viele für ihn unbekannte Funktionen gab, welche er somit nur selten testete.
Proband 2: Datum 16.08.2016, commit hash: 6b06dd0
Auffälligkeiten: Auch hier traten keine unerwarteten Fehler im Programm auf.
Verwirrend war zum einen, dass man durch Doppelklick auf einen Methodengraphen in der Strukturansicht diesen öffnen kann, jedoch durch einen Doppelklick auf einen Knoten im Callgraphen diesen dazugehörigen Methodengraphen nicht öffnen konnte, nur mit Rechtsklick->open".
Zum anderen war das Graph verschieben mithilfe der Tastenkombination Strg + Recktsklick ungewöhnlich.
Ebenso unschön wurde das Filtern von Kanten über das checken der checkboxes gefunden, sowie die Unwissenheit über momentan vorhandene Knoten und Kanten im Graphen. Man kann auch Knoten und Kanten filtern, die gar nicht im Graph vorhanden sind.

3.4 Manuelle Tests

3.4.1 Randfalltests

Es wurden Randfälle des Programmverlaufes getestet, die entweder zu Fehlern führen könnten oder schon zu welchen geführt haben.
Ausgeführt wurden Randfalltests zum Beispiel im Import, dem Layoutalgorithmus.
Dem Importer wurden .graphml Dateien übergeben, welche nicht einen JOANA-Graphen darstellen
Im Layoutalgorithmus wurden Graphen mit isolierten Knoten, Knoten auf dem selben Layer, Kanten die zumindest ein Layer überspringen und somit intern einen Pfad darstellen und Knoten mit Selbstzykel getestet. Diese Fälle müssen einzeln betrachtet werden, da davon ausgegangen wurde, dass das die Knotenstruktur auf dan Layer hierarchisch ist, also die Kanten nur von oben nach unten gehend.

3.5 Andere durchgeführte Tests

3.5.1 Überdeckungstests

Es wurden Überdeckungstests sowohl über JUnit-Tests der einzelnen Projekte, als auch über alle Projekte während eines Programmlaufes mithilfe von EclEmma durchgeführt.
Das Programm enthält folgende Projekte, die nach Möglichkeit einzeln getestet wurden und für die es auch, falls möglich, extra JUnit Tests gibt : app, Graph von Ansicht, graphml, joana, shared, sugiyama und svg.
Im Projekt ”app” befindet sich alles zu GUI gehörende, dieses Porjekt kann aufgrund der enom hohen Zahl an möglichen Interaktionen (Klick, Klick + ziehen, Rechtsklick, Graph verschieben, Knoten selektieren .....) nicht vollständig abgedeckt werden.
”Graph von Ansicht” enthält sowohl Testfälle, die einzelne Projekte und Klassen testen, als auch Testfälle, die zum Testen nacheinander ablaufender Prozesse genutzt werden
In ”graphml” befindet sich der GraphML-Importer.
”joana” enthält JOANA-spezifische Funktionalitäten und Strukturen (z.B. Call- und Methodengraph, FieldAccess...)
”shared” beinhaltet das Graphmodel und Graphbuilder-Interfaces.
Im Porjekt ”sugiyama” befindet sich alles zum Layoutalgorithmus Sugiyama-Layout gehörende und notwendige(z.B. die fünf Sugiyama-Schritte, SugiyamaGraph, SugiyamaLayoutAlgorithm...)
”svg” enthält den SVG-Exporter.
Da ”Graph von Ansicht” nur Testfälle beinhaltet, wird auf das Anzeigen einer Überdeckung über dieses Projekt verzichtet.

3.5.1.1 Testfälle aus dem Pflichtenheft

Alle Testfallszenarien aus dem Pflichtenheft werden direkt nacheinander ausgeführt.
Die Überdeckung über alle Projekte ist sehr stark abhängig von der importierten Datei und dem benutzten Graphen dieser Datei, da diese Graphen zum Teil unterschiedliche Eigenschaften haben(selfloop, kanten auf dem selben layer...).

Datei: Fibonacci.pdg.graphml
Graph: Fibonacci.mult()
Überdeckung app: 72,1%
Überdeckung graphml: 86,4%
Überdeckung joana: 80,8%
Überdeckung shared: 54,2%
Es werden Parameter, die vor dem Layouten des Methodengraphen verändert werden können, durch diese Tests nicht abgedeckt.
Überdeckung sugiyama: 75,2%
Auch hier können keine änderbaren Parameter verarbeitet werden, weil keine gesetzt wurden.
Überdeckung svg: 90%

3.5.1.2 JUnit Tests

Überdeckung app:
Überdeckung graphml:
Überdeckung joana:
Überdeckung shared:
Überdeckung sugiyama:
Überdeckung svg: 90%

3.5.1.3 Test über Programmdurchlauf

Es werden alle Method- und Callgraphen mit einer Knotenzahl von unter 1000 aus allen Dateien außer BigCG nacheinander importiert und gelayouted.

Überdeckung app: 0%
Keine GUI Interaktionen.
Überdeckung graphml: 80.9%
Überdeckung joana: 80.1%
Überdeckung shared: 47.2%
Parameter werden nicht angepasst, FastGraphAccessor ist komplett unbenutzt.
Überdeckung sugiyama: 78.6%
SugiyamaGraph bietet manche unbenutzte Funktionalität, die nie benutzt wird.
CrossMinimizer hat nur eine Überdeckung von 57.4%, da dieser veränderte Parameter berücksichtigt, werden diese Abschnitte aufgrund von nicht veränderten Parametern nicht ausgeführt.
Überdeckung svg: 0.4%

Kein Export wird ausgeführt.

3.5.2 Performance Tests

Getestet wurden hier Methodengraphen aus erstellten Graphen des JOANA-Graph-Analyzer. Gemessen wird die Zeit für das Berechnen des Layout des Methodengraphen, also von der Übergabe des Methodengraphen in interner Repräsentation nach dem Import an den Layoutalgorithmus bis dieser erst einen SuguyamaGraph erstellt und anschließend die fünf Sugiyama-Steps ausgeführt hat.
Die benötigte Zeit zum Layouten ist nicht nur abhängig von der Anzahl der Knoten und Kanten eines Graphen, sondern auch von dessen Aufbau (Anzahl an logischen Layern, einzelne dichte Stellen im Graphen, ...).
Ebenso hängt die benötigte Zeit stark von der Anzahl der Läufe des Kreuzungsminimierers bei der Kreuzungsminimierung zwischen zwei Layern ab. Die Anzahl an Läufen kann über den Menüpunkt „Layout → Properties“angepasst werden.
Im folgenden werden Graphen mit einer Anzahl von 10 Läufen gelayouted.

Datei: BigControl.pdg.graphml
Methodengraph: BigControl.main()
Information zum Graph: 779 Knoten, 4917 Kanten, 1 FieldAccess
Programmeinstellungen: CrossMinimizer runs: 10, use threshold: false
Zeit: 6.6s, 162.4s, 30.5s, 8.9s, 6.6s, 7.6s, 369.1s, 6.9s

Datei: PhiMadness.pdg.graphml
Methodengraph: PhiMadness.phiMadness()
Information zum Graph: 393 Knoten, 2937 Kanten, 25 FieldAccesses
Programmeinstellungen: CrossMinimizer runs: 10, use threshold: false
Zeit: 44.5s, 79.6s, 46.6s, 58.3s, 51.7s, 38.3s, 43.4s, 45.1s