Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik

Eine Implementierung von Dungs abstrakten Argumentations – Frameworks

Bachelorarbeit

Leipzig im September 2010

vorgelegt von Jochen Tiepmar
Studiengang Theoretische Informatik

Betreuender Hochschullehrer: Prof. Dr. Gerhard Brewka
Institut für Informatik, Abteilung Intelligente Systeme

Inhaltsverzeichnis

1 Einführung

Abstrakte Argumentations – Frameworks (Abstract Argument Systems / Abstract Argumentation Frameworks) sind Modelle, mit denen sich Argumentmengen und deren Beziehung untereinander darstellen lassen. Diese Frameworks können dazu genutzt werden, eine Menge von Aussagen formal so darzustellen, dass es möglich ist, zu berechnen, welche dieser Aussagen im Kontext der anderen (der Aussage eventuell widersprechenden) Aussagen als gerechtfertigt oder ungerechtfertigt gelten. Dabei ist es wichtig, dass die Aussagen nicht generell zu richtig oder falsch ausgewertet werden, sondern immer im Kontext zu den anderen Aussagen stehen.
Die Aussage „Herr Drechselmeyer ist unschuldig“ würde in dem Kontext einer gerechtfertigten Aussage „Ein Augenzeuge hat Herrn Drechselmeyer gesehen“ als ungerechtfertigt gelten. Würde diese Aussage wegen einer Aussage „Der Augenzeuge ist stark kurzsichtig und hatte seine Brille vergessen“ als ungerechtfertigt gelten, würde Herr Drechselmeyer wieder als unschuldig gelten, unabhängig davon, was jetzt stimmt und was nicht.

Bei dieser Bachelorarbeit handelt es sich um eine Implementierung dieser Systeme und einer passenden Bedienung mittels einer graphischen Nutzeroberfläche in der Programmiersprache JAVA.
Im weiteren Verlauf werde ich zuerst die theoretischen Grundlagen erklären, anschließend die Bedienung der im Rahmen dieser Arbeit entstandenen graphischen Oberfläche erläutern und den strukturellen Aufbau der Implementierung beschreiben.

Die zur Illustration eingefügten Bilder wurden direkt von dem entwickelten Programm mithilfe des Freeware – Programms ScreenshotCaptor [1] angefertigt.
Die Diagramme, welche bei der Beschreibung der Programmstruktur Verwendung finden, wurden mit der frei erhältlichen Community – Version des Programms JUDE[2] oder dem unter Open Source – Lizenz stehenden Programm Dia[3] manuell angefertigt.

Eine ausführbare .jar – Datei sowie der Quellcode und die JavaDoc – Dokumentation liegen dieser Arbeit als CD – Rom bei und sind per Download über http://sourceforge.net/projects/argumentationfr/files/ erhältlich.

1 http://screenshot-captor.softonic.de/
2 http://jude.change-vision.com/jude-web/product/community.html
3 http://dia-installer.de/index_de.html

2 Theoretische Grundlagen

Zu Beginn möchte ich die theoretischen Grundlagen, die dieser Arbeit zugrunde liegen erläutern.
Diese Grundlagen sind aus [1] und [2] von mir frei übersetzt worden; somit stellt dieses Kapitel im Ganzen ein sinngemäßes Zitat dar. Die Algorithmen, welche für die Implementierung genutzt wurden, sind in [2] beschrieben.

2.1 Argumente und Attacken

Abstrakte Argumentations – Frameworks basieren auf einem Artikel von Dung[3] und stellen eine Menge von (abstrakten) Argumenten A und deren Beziehungen (Attacken) R untereinander dar. Ein Argument a kann hier als ein Träger einer Aussage angesehen werden.
Eine Attacke (a,b) findet immer von einem Argument a zu einem anderen Argument b statt. Diese gerichtete Relation zwischen 2 Elementen impliziert eine Darstellung mittels gerichteter Graphen. Die Knoten bilden dabei die Argumente, die Attacken werden von den Kanten in Richtung des attackierten Argumentes repräsentiert.

Fig. 2.1 zeigt ein einfaches Framework, mit 2 Argumenten a und b und einer Attacke (a,b) .

Fig. 2.1: a attackiert b

Das Framework in Fig 2.1 kann man nun so interpretieren, dass b unter der Annahme, dass a gerechtfertigt (justified) ist, als ungerechtfertigt (unjustified) ausgewertet wird.

Wird ein Argument attackiert, so werden seine Attacken unterdrückt und sie haben keine Auswirkungen auf die von ihm attackierten Argumente. Es gibt also die Möglichkeit ein attackiertes Argument wieder zu beleben. Dieses Prinzip wird als **reinstated – principle** [1, S. 30] bezeichnet und für sinnvolle Frameworks als gegeben angesehen.
Wird a von einem neuen Argument c attackiert, wie in Fig 2.2 dargestellt, wird b wieder als gerechtfertigt interpretiert, da das attackierende Argument a als ungerechtfertigt ausgewertet wird und dessen Attacken keine Auswirkungen haben.

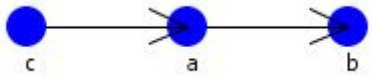


Fig. 2.2: c attackiert Fig 2.1

Für das Vorgehen beim Auswerten von Frameworks gibt es 2 grundsätzliche Ansätze: der Extension – Ansatz und der Labeling – Ansatz.

2.2 Extensions

Bei dem Extension – Ansatz bildet die Menge aller als gerechtfertigt ausgewerteten Argumente je eine Extension. Fig 2.1 hätte {a} und Fig 2.2 {c,b} als mögliche Extensions.
Dieser Ansatz eignet sich gut um einen relative einfachen Einstieg in das Thema zu erhalten und gewisse Dinge, wie zum Beispiel die Preferred Semantik lassen sich mithilfe dieses Ansatzes leichter erklären. Für die Implementierung erschien der Labeling – Ansatz jedoch aus Gründen, die im folgenden Abschnitt erläutert werden, besser geeignet.

2.3 Labelings

Beim Labeling – Ansatz wird jedem Argument des Frameworks ein Label zugewiesen, welches wiedergibt, ob dieses als gerechtfertigt gilt. Diese Label – Menge wird im Weiteren als Labeling bezeichnet.

Die Label werden dabei nach folgenden Kriterien vergeben [2, S. 107] :

> Ein Argument ist **legal IN**, wenn es von keinem Argument attackiert wird, welches IN oder UNDEC ist.
> Ein Argument ist **legal OUT**, wenn es von mindestens einem Argument attackiert wird, welches IN ist.
> Ein Argument ist **legal UNDEC**, wenn es von keinem Argument attackiert wird, welches IN ist und nicht jedes attackierende Argument OUT ist.

IN entspricht hier den Argumenten, die gerechtfertigt sind, OUT den Argumenten, die ungerechtfertigt sind und alle Argumente, über die keine Aussage getroffen werden kann erhalten das Label UNDEC.

Die Bezeichnung legal lässt bereits erahnen, dass es auch jeweils illegale Fälle geben kann [2, S. 107].

> Ein Argument ist **illegal IN|OUT|UNDEC,** wenn es ein IN|OUT|UNDEC – Label erhalten hat, aber damit gegen die Regeln für das jeweilige Label verstößt.

Daraus lassen sich Eigenschaften für das Labeling ableiten, die im Folgenden auch für die gefundenen Labelings als gegeben gefordert sein sollen.

> Ein Labeling ist **zulässig** (admissable [2, S. 107]), wenn es keine Argumente enthält, die illegal IN sind, und keine Argumente enthält, die illegal OUT sind.

Dies entspricht der Eigenschaft, dass kein Argument, welches IN ist, von einem anderen Argument, welches ebenfalls IN ist, angegriffen wird und kein Argument OUT ist, welches nicht attackiert wird.

> Ein Labeling ist **komplett** (complete [2, S. 107]), wenn es zulässig ist und keine Argumente enthält, die illegal UNDEC sind.

Diese Forderung wirkt sich in Verbindung mit der Eigenschaft zulässig so aus, dass jedes Argument in irgendeiner Form ein Label erhalten muss, und so wenige Argumente wie möglich UNDEC sein sollen.

Bei der implementierten GUI wird eine solche Zuordnung von Labels u.a. durch folgende farbliche Zuordnungen dargestellt:

Fig. 2.3: a=IN b=OUT c=UNDEC

Ein Labeling lässt sich nun als ein Tripel von Argumentmengen
{ (Argumente IN), (Argumente OUT), (Argumente UNDEC) } darstellen, welches sozusagen eine Interpretation des Frameworks wiedergibt.

Für Fig 2.1 ergibt sich beispielsweise das Labeling

Fig. 2.4: { (a),(b)( ) }

und für Fig 2.2

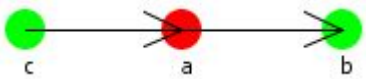


Fig. 2.5: { (b,c),(a),( ) }

UNDEC lässt sich nicht am Beispiel erläutern, da UNDEC – Argumente abhängig von der gewählten Semantik auch ein IN – oder OUT – Label erhalten können. Ein einfaches Framework mit einem Argument, welches unabhängig von der Semantik

UNDEC ist, besteht aus einem Argument, welches sich selbst attackiert und sonst keinerlei attackierende Argumente hat. Unter Verwendung der Grounded Semantik werden weiter alle in sich geschlossenen Kreise zu UNDEC ausgewertet, was bei der Preferred Semantik nicht zwingend der Fall ist.

Jedes Labeling lässt sich in eine Extension umwandeln, indem man alle Argumente, welche IN sind zu der (vorher leeren) Extension hinzufügt, was umgekehrt nicht zwingend möglich sein muss.
Beispielhaft lässt sich das gut an Fig 2.2 erkennen:

> Während aus dem Labeling *{ (b,c),(a),( ) }* einfach die Extension *{ b,c }* abgelesen werden kann, lässt sich um umgedrehten Fall nicht erkennen, welches Labeling sich aus der Extension *{ b,c }* ergibt, da das Argument a sowohl OUT als auch UNDEC sein könnte.

Der Labeling – Ansatz hat somit einen höheren Informationsgehalt als der Extension – Ansatz und jede Extension lässt sich auf Wunsch einfach aus jedem Labeling ablesen

Deshalb erscheint mir der Labeling – Ansatz für eine Implementierung sinnvoller.

2.4 Semantiken

Semantiken geben an, nach welchen Regeln die Interpretation der Argumente stattfindet, das heißt, wie beim Finden von Labelings vorgegangen werden soll. Prinzipiell gelten dabei natürlich nach wie vor die oben erwähnten Regeln, allerdings können sich je nach Art des Vorgehens (je nach Wahl der anzuwendenden Semantik) bei bestimmten Frameworks unterschiedliche Labelings als gültige Auswertungen ergeben. Im Folgenden werde ich die Semantiken Grounded, Preferred und Stable erläutern.

2.4.1 Grounded Semantik

Die Grounded Semantik [1, S. 36] lässt sich am besten konstruktiv erklären.

Gestartet wird mit einem leeren Labeling *{ ( ),( ),( ) }*.
Alle Initial – Argumente erhalten das Label IN, wenn sie kein Label besitzen. Jedes Argument, welches nicht attackiert wird, gilt hierbei als Initial – Argument.
Anschließend erhalten alle Argumente, die von einem dieser Initial – Argumente attackiert werden und noch kein Label besitzen, ein OUT – Label, wodurch unter Umständen neue Initial – Argumente entstehen[4]. Diese beiden Schritte werden solange wiederholt, bis kein neues Initial – Argument entsteht.
Letztendlich erhalten alle Argumente, die kein Label besitzen, UNDEC als Label.

4 Wegen reinstated principle

Beispielhaft möchte ich dieses Vorgehen an dem in Fig 2.6 dargestellten Framework erläutern

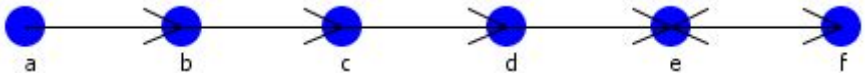


Fig. 2.6: Beispiel - Framework für die Grounded Semantik

Das Grounded Labeling in diesem Framework lässt sich (ausschließlich) mit folgenden Schritten erreichen:

0. *{ ( ), ( ), ( ) }*	a ist einziges Initial – Argument
1. *{ (a),( ),( ) }*	a IN, b wird attackiert.
2. *{ (a),(b),( ) }*	b OUT, c wird Initial – Argument.
3. *{ (a,c),(b),( ) }*	c IN, d wird attackiert.
4. *{ (a,c),(b,d),( ) }*	d OUT, e wird nicht Initial – Argument, weil es von f attackiert wird.
5. *{ (a,c),(b,d),(e,f) }*	keine neuen Initial – Argumente, e und f bleiben übrig und werden UNDEC.

Damit ergibt sich das Grounded Labeling :

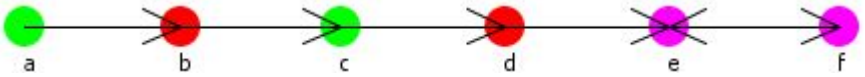


Fig. 2.7: { (a,c),(b,d),(e,f) }

Für jedes Framework existiert genau 1 (eindeutiges) Grounded Labeling. Bei leeren Frameworks gilt dabei das leere Labeling als Grounded.

Argumente, die im Grounded Labeling IN bzw. OUT sind, sind in jedem Labeling des Frameworks IN bzw. OUT. Dies liegt darin begründet, dass der Algorithmus mit den Argumenten beginnt, die zwangsläufig IN sind und von da aus die Attacken der anderen Argumente zu (zwangsläufigen) Ergebnissen auswertet.
Über die Argumente, die im Grounded Labeling das Label UNDEC erhalten, lassen sich keine weiteren Aussagen treffen. Diese könnten in anderen Semantiken je nach Kontext zu jedem Label ausgewertet werden, wodurch sich bei anderen Semantiken die Möglichkeit ergibt, mehrere Labelings zu einem Framework zu erhalten.
Diese Eigenschaft führt zu der Idee, Grounded Labelings ohne dessen UNDEC – Labels als Grundlage für Stable und Preferred Semantiken zu nutzen[5], was ich als Option mit implementiert habe.

Eine Besonderheit ergibt sich hier bei Frameworks, die in sich Kreise bilden, wie die beiden Beispiele in Fig. 2.8.

5 Siehe Kapitel 3.6 'Grounded für Preferred und Stable nutzen'. Es handelt sich dabei um keine fundierte Aussage, sondern um eine Idee, die sich während der Implementierung ergab und interessant schien. Bemerkenswerten Einfluss auf die Performanz habe ich aber nicht festgestellt.

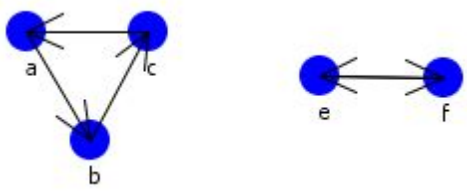


Fig. 2.8: Kreisförmiger Aufbau eines Frameworks

Da sich hier kein Initial – Argument finden lässt, bricht der Algorithmus nach dem 1. Durchlauf ab und alle Argumente erhalten das Label UNDEC, da kein Argument die Voraussetzungen dafür erfüllt, ein anderes Label zu erhalten.

Es ergeben sich also die Labelings

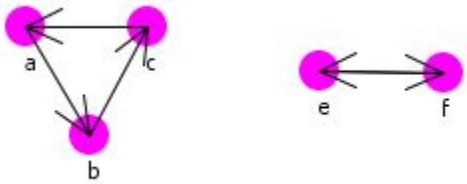


Fig. 2.9: { ( ),( ),(a,b,c) } und { ( ),( ),(e,f) }

Folgender Algorithmus [2, S.112] liefert das (eindeutige) Grounded Labeling L_G zu jedem Framework mit einer Argumentmenge A.

1. $L_0 = \{ ( ), ( ), ( ) \}$
2. **repeat**
3. $IN(L_{i+1}) = IN(L_i) \vee (x : x \notin Li \wedge \forall y : \text{if } yRx \text{ then } y \in OUT(L_i))$
4. $OUT(L_{i+1}) = OUT(L_i) \vee (x : x \notin L_i \wedge \exists y \in IN (L_i) : yRx)$
5. **until** $L_{i+1} = L_i$
6. $L_G = \{ IN (L_i), OUT (L_i), A / (IN (L_i) \vee OUT (L_i)) \}$

Dieser Algorithmus liefert die Lösung in vertretbarem Aufwand, was gegenüber dem sehr großen Aufwand bei der Berechnung der Preferred Semantik ein großer Vorteil ist.

2.4.2 Preferred Semantik

Die Preferred Semantik [1, S. 38] lässt sich sehr gut mithilfe des Extension – Ansatzes erklären.
Eine Extension ist die Menge der gerechtfertigten Argumente.
Eine Preferred Extension ist eine Extension maximaler Größe, die jeden Angreifer der Extension selbst attackiert.

Ein Preferred Labeling [2, S.116] L1 ist ein zulässiges und komplettes Labeling, für welches gilt:

(I) $\neg\exists$ Labeling L2 : L2 ist zulässig $\wedge$ IN (L1) $\subset$ IN (L2)
Es existiert kein zulässiges Labeling L2, sodass IN(L1) eine Teilmenge von IN(L2) ist.

Beispielhaft möchte ich das an Fig. 2.10 erklären.

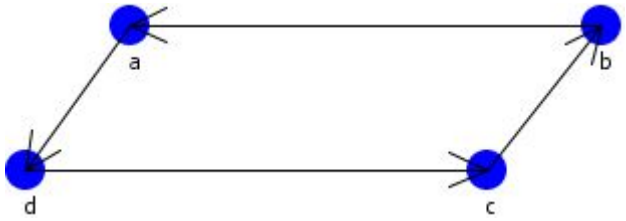


Fig. 2.10: Beispiel - Framework für die Preferred Semantik

Es existieren in diesem Beispiel 2 Preferred Labelings:

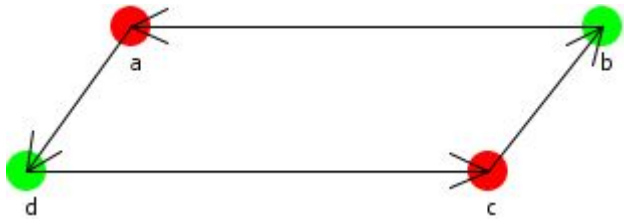


Fig. 2.11: { (b,d),(a,c),( ) }

und

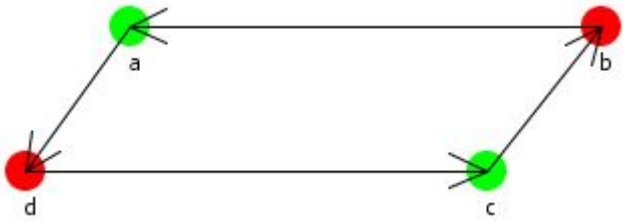


Fig. 2.12: { (a,c),(b,d),( ) }

Zum Vergleich möchte ich anmerken, dass das Grounded Labeling hier *{ ( ),( ), (a,b,c,d) }* wäre.

Jedes andere Labeling würde gegen eine der Voraussetzungen für die Preferred Semantik verstoßen.
So würde jedes Labeling L1 mit nur 1 Argument mit dem Label IN ein Labeling L2 erlauben, welches die IN – Argumente von L1 enthält und damit gegen Regel (I) verstoßen. Jedes Labeling mit 2 benachbarten Argumenten, die IN wären, wäre nicht zulässig, da eines von beiden Argumenten von einem Argument, welches IN ist attackiert wird und somit illegal IN ist. Jedes Labeling mit einem Argument, welches UNDEC ist, wäre kein komplettes Labeling, da dieses Argument im Beispiel illegal UNDEC wäre.

Der Algorithmus zur Berechnung von Preferred Labelings fällt um einiges umfangreicher aus als der Algorithmus für die Grounded Labelings und erfordert noch

ein wenig Vorarbeit.

Regel (I) lässt es sinnvoll erscheinen, zuerst von einem Labeling auszugehen, welches allen Argumenten das Label IN zuteilt(All – IN Labeling), und anschließend dieses Labeling anzupassen bis es sowohl zulässig als auch komplett ist.

'Zulässiger' wird ein Labeling, indem man einem Argument, das illegal IN oder illegal OUT ist, sein legales Label zuteilt. Dies wird durch den sogenannten Transistion Step realisiert.

> Sei L ein Labeling für ein Framework, sei a ein Argument, welches illegal IN in L ist.
> Ein **Transition Step** [2, S. 116] ist nun folgendes:
>
> 1. Das Label von a ändert sich von IN zu OUT.
> 2. Jedem von a angegriffenen Argument, welches illegal OUT ist, wird das Label UNDEC zugewiesen.

Schritt 1 gibt dem Argument a also sein legales Label, Schritt 2 ändert das Label der Argumente, die durch Schritt 1 illegal OUT geworden sind, zu UNDEC.

> Eine **Transition Sequence** [2, S. 116] ist eine Menge von Transition Steps.

Durch diesen Transition Step kann es nun passieren, dass ein Argument illegal UNDEC wird, und damit das Labeling nicht mehr komplett ist. Dies kann vorkommen, wenn der Algorithmus die Argumente in einer ungünstige Reihenfolge bearbeitet. Um das zu verhindern werden Argumente bevorzugt behandelt, welche super – illegal IN sind.

> Ein Argument a in einem Labeling L, welches illegal IN ist, ist zusätzlich **super – illegal IN** [2, S. 117], wenn es von einem Argument attackiert wird, welches legal IN oder UNDEC in L ist.

Preferred Labelings lassen sich also finden, indem man solange Transition Steps mit einem Argument, welches illegal IN ist, auf ein All – IN Labeling ausführt, bis dieses Labeling zulässig ist.
Um dabei ein komplettes Labeling beizubehalten, werden Argumente, welche super – illegal IN sind, für den Transition Step bevorzugt.

Sobald man auf diesem Wege ein zulässiges Labeling erreicht hat, würde jeder weitere Transition Step dazu führen, dass Regel (I) verletzt wird, da jeder Transition Step die Anzahl der Argumente, die IN sind, reduziert bzw. dazu führen, dass das Labeling nicht zulässig ist.

Die Abbruchbedingung, dass das Labeling zulässig sein soll, lässt sich hier darauf reduzieren, dass kein Argument mehr illegal IN ist, da Argumente, die illegal OUT werden, im gleichen Transistion Step ein legales Label erhalten, in dem sie illegal OUT wurden.

Folgender rekursiv arbeitende Algorithmus [2, S. 118] setzt dieses Vorgehen um:

```
1.  kandidaten:= { }
2.  findLabelings ( all - IN )
3.  print kandidaten;
4.  end;
5.
6.  procedure findLabelings ( Labeling L )
7.  #wenn L eine schlechtere Lösung als ein schon vorhandenes Labeling darstellt
8.  #backtrack
9.  if ( ∃ L' ∈ kandidaten : IN ( L ) ⊂ IN ( L' ) )
10. then return;
11.
12. #Wenn keine weiteren Transistion Steps notwendig sind
13. if ( ¬∃ x ∈ L : x illegalIN )
14. then
15.         for each L' ∈ kandidaten
16.         do
17.                 #Wenn IN(L') eine echte Teilmenge von IN(L) ist, lösche L'
18.                 if ( IN ( L' ) ⊂ IN ( L ) )
19.                         then kandidaten = kandidaten - { L' };
20.                 endIf
21.         endFor
22.         # L ist eine Lösung
23.         kandidaten = kandidaten ∨ { L };
24.         #Fertig, also versuche die nächste Möglichkeit
25.         return;
26.
27. else
28.         if ( ∃ x ∈ L : x superIllegalIn )
29.         then
30.                 y := x ∈ L : x superIllegalIn;
31.                 findLabelings ( transitionStep ( L , y ));
32.         else
33.                 for each x ∈ L : x illegalIn
34.                 do
35.                         findLabelings ( transitionStep ( L , x ));
36.                 endFor
37.         endIf
38. endIf
39. endProc
```

Der Algorithmus beginnt mit einem All – IN Labeling und prüft, ob es in den bereits gefundenen Kandidaten ein Labeling gibt, welches die IN – Argumente des übergebene Labeling als Teilmenge besitzt.
Ist das der Fall, wird ein Rekursionsschritt zurückgegangen.
Andernfalls wird das Labeling auf Zulässigkeit geprüft.
Ist das Labeling zulässig, wird mithilfe der bisher angesammelten Kandidaten überprüft, ob Regel (I) bei einem Labeling der Kandidaten verletzt wird. Wenn dies der Fall ist, wird das bei den Kandidaten gefundene Labeling, welches Regel (I) verletzt, wieder verworfen.

Dann wird das übergebene Labeling den Kandidaten hinzugefügt.
Wenn das Labeling nicht zulässig war, wird ein Transistion Step mit einem Argument ausgeführt, welches super – illegal IN ist und der Algorithmus anschließend rekursiv aufgerufen. Wird kein solches Argument gefunden, wird mit jedem Argument, welches illegal IN ist, jeweils der Transition Step durchgeführt und die Funktion wieder rekursiv aufgerufen.

Dieser Algorithmus findet alle Preferred Labelings, die in einem Framework existieren. Der zeitliche Aufwand kann dabei, je nach Framework, sehr hoch werden, sodass sich schon bei Frameworks mit relativ wenigen Argumenten das Finden der Preferred Labelings aufgrund einer hohen Anzahl nötiger Rekursionsschritte als zu aufwändig erweisen kann.

2.4.3 Stable Semantik

Die Stable Semantik [1, S. 37] ist ähnlich der Preferred Semantik. Der zugrunde liegende Gedanke ist eine Extension, die nicht nur jedes attackierende Argument angreift (wie bei der Preferred Semantik), sondern jedes Argument, welches nicht zu dieser Extension gehört, attackiert.
Diese Semantik stellt also eine aggressivere Forderung als die Preferred Semantik dar. Daraus ergibt sich, dass jedes Stable Labeling ein spezielles Preferred Labeling ist. Umgekehrt ist aber nicht jedes Preferred Labeling ein Stable Labeling.

Dass jedes Argument entweder zu der Extension gehört (es muss das Label IN erhalten) oder attackiert wird (es muss das Label OUT erhalten), bedeutet, dass es keine Argumente mit dem Label UNDEC in einem Stable Labeling geben kann. Genauer gesagt ist jedes Preferred Labeling, welches kein UNDEC – Argument enthält, ein Stable Labeling. Da es Frameworks gibt, die kein solches Labeling enthalten, ist die Existenz eines Stable Labelings nicht zwingend für jedes Framework gegeben. Ein Argument, welches sich selbst attackiert oder das Framework in Fig. 2.13 enthalten zum Beispiel keine Stable Labelings.

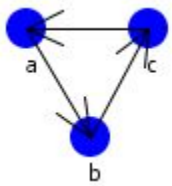


Fig. 2.13: Framework ohne Stable Labeling

Allerdings existiert in Fig. 2.13 ein Preferred Labeling:

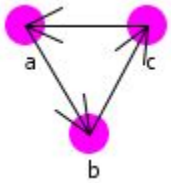


Fig. 2.14: { ( ),( ),(a,b,c) }

Das folgende Beispiel ist mit Fig. 2.10 identisch. Die beiden Preferred Labelings enthalten keine Argumente, die UNDEC sind und sind somit auch Stable Labelings

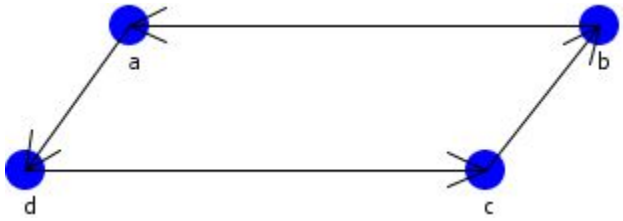


Fig. 2.15: Beispiel - Framework für die Stable Semantik

Es existieren in diesem Beispiel 2 Stable Labelings:

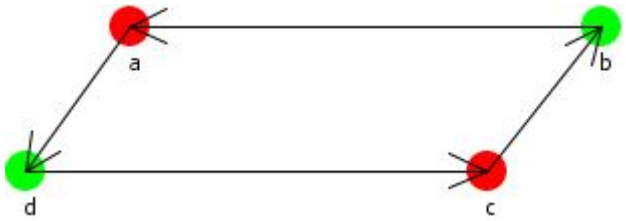


Fig. 2.16: { (b,d),(a,c),( ) }

und

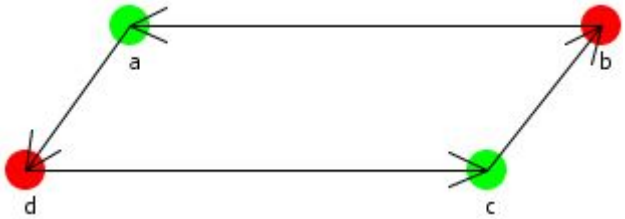


Fig. 2.17: { (a,c),(b,d),( ) }

Da die Stable Labelings ähnlich der Preferred Labelings gefunden werden, erscheint es sinnvoll, den Algorithmus der Preferred Labelings abzuwandeln. Genau genommen reicht es, Zeile 9 des Algorithmus für Preferred Labelings mit
if (UNDEC (L) != { }) zu ersetzen [2, S. 124] , sodass die Abbruchbedingung darin besteht, dass ein Argument das Label UNDEC erhalten hat. Laut [2, S. 124] ist es möglich, die Zeilen 15 bis 21 komplett zu löschen, sodass der Test auf ein bereits gefundenes Labeling, welches die aktuellen IN – Argumente enthält, wegfallen kann. Allerdings kam es dann dazu, dass identische Labelings mehrfach als richtige Labelings gefunden wurden, weshalb ich diesen Teilmengentest bei der Implementierung nicht entfernt habe.

3 Die Bedienung der GUI

Nachdem die theoretischen Grundlagen erläutert wurden, kommen wir nun zur Bedienung der im Rahmen dieser Arbeit entwickelten GUI (Graphical User Interface – Benutzeroberfläche) zum Arbeiten mit den beschriebenen Frameworks.
Nach dem Start des Programms sollte folgende GUI (ohne die roten Pfeile) erscheinen:

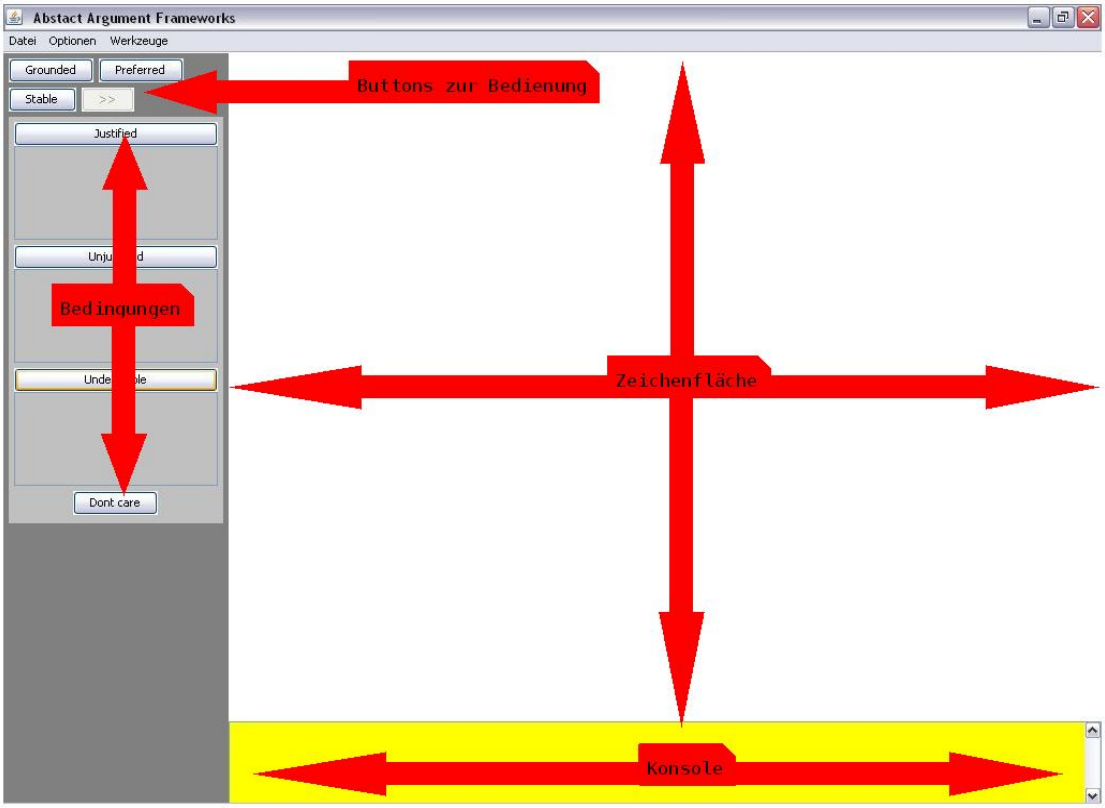


Fig. 3.1: Übersicht über die GUI

Die GUI unterteilt sich grob in 4 Bereiche:

1. Die Zeichenfläche
   Auf dieser Fläche werden die Argumente erzeugt und ein Labeling farblich passend ausgegeben.
2. Die Konsole
   Auf der Konsole werden alle passenden Labelings aufgelistet, die gefunden wurden und eventuelle weitere Informationen ausgegeben.
3. Die Bedienbuttons
   Diese Buttons dienen zur Steuerung des Programms
4. Ein Bereich, in dem Bedingungen an das zu findende Label gestellt werden können.

3.1 Erstellen eines Frameworks.

Wenn das Programm gestartet wurde startet es mit einem leeren Framework.
Als erstes sollte man also dem Framework einige Argumente hinzufügen. Dies geschieht, indem man auf der Zeichenfläche mit linker Maustaste auf einen leeren Bereich klickt. Fig. 3.2 zeigt was dann geschieht.

Fig. 3.2: Ein Argument

Es wird ein Kreis gezeichnet, der ein Argument repräsentiert. Die Farbe blau bedeutet, dass das Argument kein Label besitzt und ändert sich sobald irgendein Labeling ausgeben wird.
Unter dem Kreis steht die Zahl 0. Dies ist der Name des Argumentes. Die Argumente werden mit einer aufsteigenden ID erzeugt, das heißt das nächste erzeugte Argument bekommt '1' als ID zugewiesen.
Wird für ein Argument kein Name angegeben, wird diese ID als Name verwendet.
Der einfachste Weg einem Argument einen Namen zu geben eröffnet sich, indem man das Argument doppelt anklickt. Dadurch kommt man in das Menü dieses Argumentes, wie in Fig. 3.3.

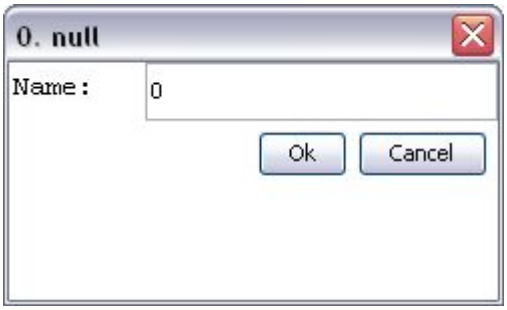


Fig. 3.3: Pointdialog

Hier kann jeder beliebige Name eingegeben und durch Druck auf 'Ok' dem Argument zugewiesen werden.

Am Beispiel nenne ich mein Argument 'Die Sonne scheint' und erhalte Fig. 3.4.

Fig. 3.4: Ein Argument mit Namen

Das Argument hat jetzt einen Namen bekommen. Beim Klicken auf das Argument ist Ihnen vielleicht ein gelber Kreis aufgefallen, der sich um das Argument gezeichnet hat, wie in Fig. 3.5.

Fig. 3.5: Ein markiertes Argument

Dieser Kreis erscheint, wenn ein Argument durch einen Linksklick markiert wurde. Wenn die Leertaste auf der Tastatur gedrückt gehalten wird, lassen sich auch mehrere Argumente auf diese Weise markieren.
Diese markierten Argumente lassen sich bei gedrückter linker Maustaste auf der Zeichenfläche verschieben, sodass man sie übersichtlicher verteilen und anordnen kann, wenn man das möchte.
Mit Druck auf die Taste Entf/Del der Tastatur werden die markierten Argumente (und alle ihrer Attacken) wieder aus dem Framework entfernt. Eine weitere Möglichkeit zum Entfernen von Argumenten besteht darin, diese über den rechten oder unteren Rand der GUI zu ziehen. Die Möglichkeit kann aber für Verwirrung sorgen, weshalb sie in den Optionen deaktiviert werden kann.

Da ein Framework mit nur einem Argument wenig Sinn macht, erzeuge ich ein neues Argument, nenne es 'Es regnet' und erhalte somit Fig. 3.6.

Fig. 3.6: Zwei Argumente

Um eine Attacke von 'Es regnet' zu 'Das Wetter ist schön' auszuführen, markiert man mit einem Linksklick 'Es regnet', klickt anschließend mit der rechten Maustaste aus 'Das Wetter ist schön' und erhält Fig. 3.7

Fig. 3.7: Zwei Argumente und eine Attacke

Der Pfeil symbolisiert eine Attacke zwischen den beiden Argumenten.
Bei einer zusätzlichen Attacke von 'Das Wetter ist schön' auf 'Es regnet' wird ein Doppelpfeil zwischen diesen gezeichnet und man erhält Fig. 3.8

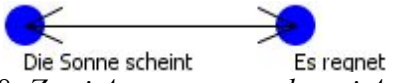


Fig. 3.8: Zwei Argumente und zwei Attacken

Interpretieren lässt sich dieses Framework (ohne Beachtung einer besonderen Semantik)auf zwei Arten:

Wenn das Wetter schön sein soll, ist Regen ungerechtfertigt.
Wenn es regnet, ist es ungerechtfertigt, dass das Wetter schön ist.

Damit haben wir ein Framework mit 2 Argumenten und den Attacken zwischen diesen erstellt und können Labelings suchen.

3.2 Die Konsole

Die Konsole dient zum Auflisten aller gefundenen Labelings. Die Labelings werden dabei in folgender Weise ausgegeben:

{{Argumente IN}, {Argumente OUT}, {Argumente UNDEC}}

Die Suche nach dem Grounded Labeling für Fig. 3.8 liefert also die folgende Ausgabe auf der Konsole:

```
({}{}{Es regnet , Das Wetter ist schön })
```

Fig. 3.9: Konsolenausgabe für ein Labeling

Es gibt kein Argument, welches IN oder OUT ist, alle Argumente sind UNDEC (Da es kein Initial– Argument gibt).

3.3 Labelings suchen

Das Berechnen von Labelings bestimmter Semantiken funktioniert über die Buttons im linken oberen Bereich der GUI.
Im Folgenden stelle ich die Buttons vor, mit denen sich die passenden Labelings berechnen lassen.

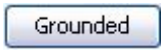


Fig. 3.10: Grounded - Button

Dieser Button liefert das Grounded Labeling zum erstellten Framework. Das Labeling wird anschließend auf der Konsole ausgegeben und auf der Zeichenfläche dargestellt.

Fig. 3.11: Preferred - Button

Dieser Button liefert das Set von Preferred Labelings zu dem erstellten Framework. Alle Labelings werden auf der Konsole aufgelistet und eines davon auf der Zeichenfläche dargestellt.

Fig. 3.12: Stable - Button

Dieser Button liefert das Set von Stable Labelings zu dem erstellten Framework. Alle Labelings werden auf der Konsole aufgelistet und eines davon auf der Zeichenfläche dargestellt.

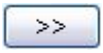


Fig. 3.13: Next - Button

Dieser Button dient zum Wählen des zu zeichnenden Labelings auf der Zeichenfläche. Er wird erst aktiviert, wenn mehrere Labelings zu einer Semantik gefunden wurden, und ermöglicht das Durchschalten des zu zeichnenden Labelings. Das aktuell gezeichnete Labeling wird zusätzlich auf der Zeichenfläche oben links ausgegeben.

3.4 Anfragen an das Labeling stellen

Es ist möglich an ein Framework Anfragen zu stellen, die Auskunft darüber geben, ob ein Labeling existiert, in dem ein oder mehrere Argumente ein bestimmtes Label erhalten. Solche Anfragen sind zum Beispiel (für ein jeweils passendes Framework):

1. Existiert ein Labeling, in dem das Argument '*a*' IN ist ?
2. Gib mir alle Labelings, in denen '*a*' IN ist.
3. Existiert ein Labeling, in dem '*a*' IN ist und '*b*' OUT ist?
4. Gibt es ein Argument, welches immer IN ist ?
5. Welche Argumente erhalten in allen Labelings dieselben Label?

Dabei werden manche Anfragen durch andere Anfragen mit beantwortet. Zum Beispiel beantwortet Anfrage 2 auch Anfrage 1 . Anfrage 4 wird von Anfrage 5 mit beantwortet.

Anfragen 1 – 3 kann man für eine Implementierung in folgender Weise umformulieren:

Welche Labelings passen zu einem Labeling, welches vorher festgelegt wurde ?

Anfragen 4 und 5 lassen sich lösen, indem ein **universelles Labeling** erzeugt wird, welches alle Argumente enthält, die in jedem gefundenen Labeling dasselbe Label erhalten.
Es gibt also 2 grundsätzliche Aufgaben, aus denen sich die die einzelnen Anfragen ablesen lassen:

(I). Liefere mir alle zu einem Dummy – Labeling passenden Labelings.
(II). Liefere mir (wenn vorhanden) ein universelles Labeling, welches alle Argumente, deren Label sich in den unterschiedlichen Labelings nicht ändert, enthält.

(II) wird automatisch auf der Konsole mit ausgegeben, wenn die Berechnung dieses universellen Labelings nicht in den Optionen deaktiviert wurde und ein solches Labeling existiert.
Anfrage 1 lässt sich über das Bedingungsmenü links der Zeichenfläche lösen.

Fig. 3.14: Bedingungsmenü

Dieses Menü besteht aus 3 Buttons, mit denen sich ein Labeling erzeugen lässt, welches anschließend mit den gefundenen Labelings verglichen wird. Um ein Label für ein Argument festzulegen, markiert man die gewünschten Argumente auf der Zeichenfläche und klickt auf den passenden Button. Unter den Buttons werden die Argumente aufgelistet, die dem Label zugeordnet wurden.
Unter diesem Menü findet sich ein Button, mit dem die aktuell markierten Argumente aus dem Labeling, welches die Bedingungen stellt, wieder entfernt werden können.
Betrachten wir folgendes Beispiel, welches eine Erweiterung von Fig. 3.8 darstellt:

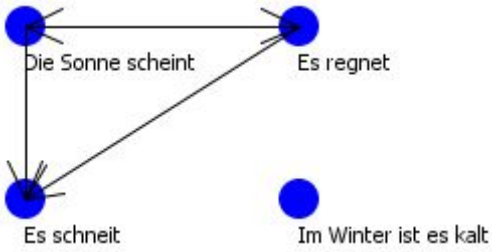


Fig. 1: Ein weiteres Framework

Es existieren 2 Preferred Labelings für dieses Beispiel:

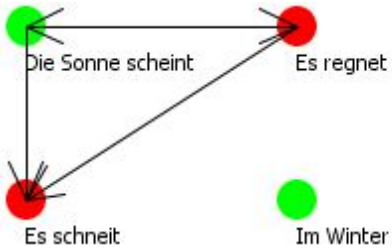


Fig. 3.15: { (Im Winter ist es kalt, Die Sonne scheint) (Es regnet, Es schneit) ( ) }

und

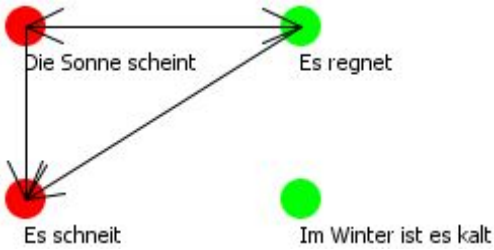


Fig. 3.16: { (Im Winter ist es kalt, Es regnet) (Die Sonne scheint, Es schneit) ( ) }

Das Argument '*Es schneit*' wird in beiden Labelings als ungerechtfertigt ausgewertet, da es in beiden Labelings von einem gerechtfertigten Argument attackiert wird.
Das Argument '*Im Winter ist es kalt*' gilt in beiden Labelings als gerechtfertigt, was einfach daran liegt, dass kein Argument es attackiert.

Es gibt also 2 Argumente, die in jedem Preferred Labeling das selbe Label erhalten: eines, welches immer IN ist und eines, welches immer OUT ist. Diese beiden Argumente sind im universellen Labeling dokumentiert und es findet sich, sofern dies in den Optionen nicht deaktiviert wurde, folgende Ausgabe auf der Konsole:

```
Allgemeingültiges Labeling:
({Im Winter ist es kalt }{Es schneit }{ })
```

Fig. 3.17: Konsolenausgabe für das universelle Labeling

Fig. 3.17 findet sich über der Auflistung der gefundenen Labelings und gibt alle Argumente wieder, die in jedem gefundenen Labeling dasselbe Label erhalten haben. Damit werden alle Anfragen der Form (II) bzw. 4 – 5 beantwortet.

Jetzt suchen wir nach einem Labeling, welches das Argument '*Die Sonne scheint*' als gerechtfertigt auswertet.
Nach dem Markieren des Argumentes '*Die Sonne scheint*' und anschließendem Klick auf den Button 'Justified' findet sich das Argument in dem Ausgabefeld unter dem Button 'Justified'.

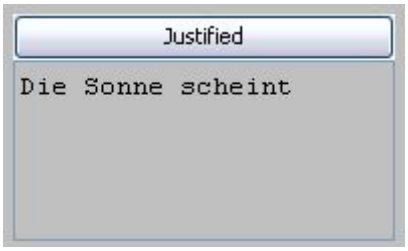


Fig. 3.18: Bedingungsmenü im Detail

In diesem Ausgabefeld werden alle Argumente, die als gerechtfertigt gelten sollen, aufgelistet.
Nach einem Klick auf den 'Preferred' – Button findet das Programm das passende (in diesem Fall eindeutige) Labeling:

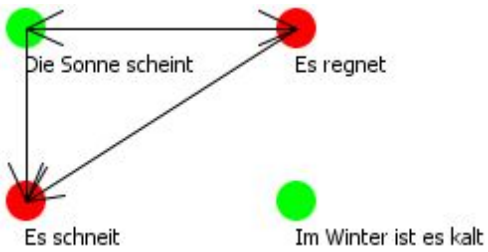


Fig. 3.19: { (Im Winter ist es kalt, Die Sonne scheint) (Es regnet, Es schneit) ( ) }

Auf der Konsole wird nun kein allgemeingültiges Labeling mehr ausgegeben, was daran liegt, dass generell nur ein Labeling gefunden wurde.

Betrachten wir folgende Kombination von Bedingungen:

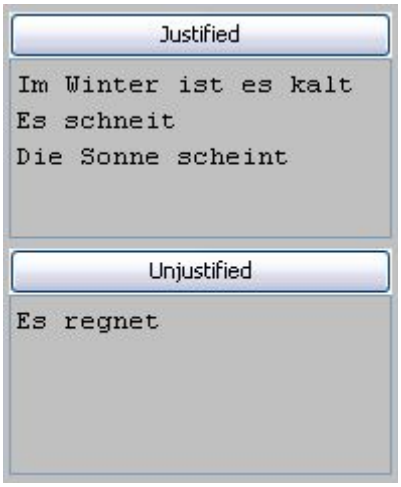


Fig. 3.20: Nicht lösbare Bedinung

Es soll also ein Labeling gefunden werden, welches Die Argumente '*Es schneit*', ' *Die Sonne scheint*' und '*Im Winter ist es kalt*' als gerechtfertigt ansieht und das Argument '*Es regnet*' als ungerechtfertigt bewertet.

Ein Klick auf Preferred liefert nun kein Labeling zurück, da kein passendes Preferred

Labeling existiert, welches diese Kombination von Bedingungen erfüllt. Der Fehler liegt bei dieser Kombination bei der Bedingung für das Argument '*Es schneit*'. Dieses Argument erhält in jedem Labeling das Label OUT und kann somit nicht als gerechtfertigt bewertet werden. Es wurde also eine nicht lösbare Anfrage gestellt.

Es gibt 2 Wege diese Kombination von Bedingungen so anzupassen, dass wieder ein passendes Labeling gefunden werden kann:
Entweder man markiert nun das Argument '*Es schneit*' und klickt auf 'Unjustified', um die Bedingung anzupassen, oder man markiert es und klickt auf den Button 'Dont care' unter dem Bedingungsmenü. Dieser Button entfernt das Argument aus der Menge der Argumente, die die Bedingungen an das zu findende Labeling stellen.
Ich klicke auf 'Dont care' und erhalte die folgende Kombination von Bedingungen:

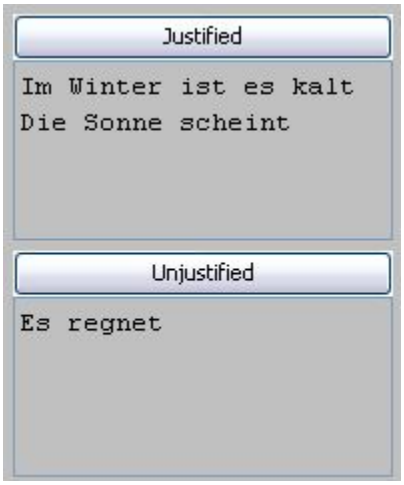


Fig. 3.21: Korrigierte Bedingung

Ein Klick auf 'Preferred' liefert nun wieder das gewünschte Labeling

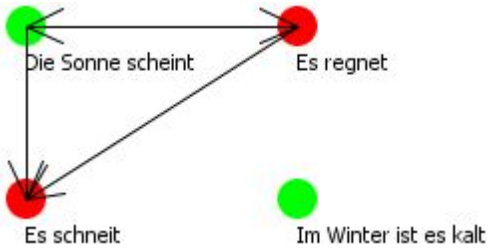


Fig. 3.22: { (Im Winter ist es kalt, Die Sonne scheint) (Es schneit, Es regnet) ( ) }

Damit lassen sich alle Anfragen der Form (I) bzw. 1 – 3 lösen

Dieses Prinzip funktioniert für jedes Argument, jedes Label und jede Semantik analog und es lassen sich beliebige Kombinationen von Bedingungen formulieren.

3.5 Das Datei Menü

Über dieses Menü können Frameworks gespeichert, verworfen und wieder geladen werden. Ich denke, dieses Menü ist selbsterklärend und weitere Erläuterungen sind nicht nötig.

3.6 Das Optionen Menü

Über dieses Menü lassen sich Optionen für das Programm festlegen. Die einzelnen Optionen haben dabei folgende Auswirkungen:

Anti – Aliasing aktivieren

Bei Aktivierung dieser Option wird Kantenglättung für das Panel aktiviert. Dadurch werden sogenannte Treppeneffekte beim Zeichnen reduziert und das Bild sieht glatter aus.
Allerdings kann das Zeichnen mit Kantenglättung unter Umständen merkliche Performance – Kosten mit sich bringen, die bei mir aber erst bei großen Frameworks (1000+ Argumente, 500+ Attacken) und dann auch nur beim Arbeiten mit dem Panel selbst auftraten.
Fig. 3.23 zeigt den Unterschied bei der graphischen Darstellung mit und ohne Anti – Aliasing. Bei Programmstart ist diese Option aktiviert.

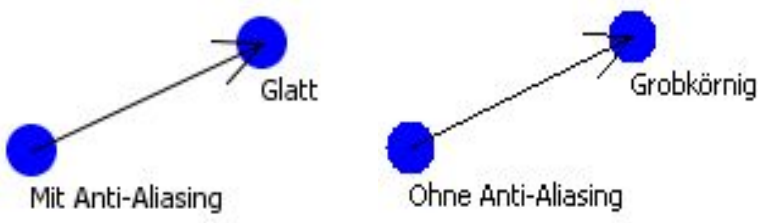


Fig. 3.23: Anti - Aliasing Vergleich

Argumentnamen anzeigen

Über diese Option lässt sich festlegen, ob die Namen der Argumente auf dem Panel gezeichnet werden sollen oder nicht. Auf die Performance hat diese Option einen eher unauffälligen Einfluss, aber beim Arbeiten mit und Visualisieren der Frameworks können die Namen unnötig oder sogar unübersichtlich werden. Bei Programmstart ist diese Option aktiviert.

Allgemeingültiges Labeling berechnen

Hier lässt sich festlegen, ob nach dem Suchen von Labelings zusätzlich nach

dem universellen Labeling gesucht werden soll. Da dabei ein Kreuzvergleich über alle gefundenen Labelings durchgeführt wird, was bei n gefundenen Labelings n*n Vergleiche bedeutet, kann diese Funktion Einfluss auf die Performance haben. Wenn sich die Anzahl der gefundenen Labelings aber halbwegs in Grenzen hält, kann diese Option ruhig aktiviert bleiben. Ich habe bisher keine nennenswerten Unterschiede registrieren können

Konsole bei Änderungen löschen

Diese Option erfüllt einen rein kosmetischen Zweck. Wird ein Labeling auf der Konsole ausgegeben und anschließend das Framework auf der Zeichenfläche geändert, stimmt die Anzeige der Konsole nicht mehr mit dem aktuell gezeichneten Labeling überein. Bei Aktivierung dieser Option wird die Konsole beim Löschen / Hinzufügen von Argumenten sowie beim Ändern des Namens eines Argumentes geleert. Diese Option ist bei Programmstart deaktiviert.

Argumente per Ziehen löschen

Das Entfernen von Argumenten, indem man sie über den Rand der GUI zieht, kann hier aktiviert oder deaktiviert werden. Bei Programmstart ist diese Option aktiviert.

Grounded für Preferred und Stable nutzen

Hier kann man wählen, ob für die Berechnung des Preferred Labelings ein Grounded Labeling als Grundsatz gewählt wird. Dabei wird vor der Berechnung des Labelings ein Grounded Labeling erzeugt, dessen UNDEC – Label wieder verworfen werden. Unter Umständen reduziert sich dabei die Anzahl der nötigen Rekursionsschritte, da viele Argumente bereits ihr zwangsläufiges Label erhalten. Standardmäßig ist diese Option deaktiviert und die Algorithmen werden mit einem All – IN Labeling gestartet.

3.7 Der Framework – Generator

Da es sich mit steigender Größe des Frameworks sehr mühsam gestaltet, jedes Argument und jede Attacke von Hand zu erzeugen erschien es schnell wünschenswert, einen Generator zu haben, der diese Aufgabe übernehmen kann. Dieser Generator ist über das Werkzeuge – Menü erreichbar und ermöglicht das Generieren von zufälligen Frameworks anhand von einigen Vorgaben.

Beim Auswählen des Framework – Generators erscheint folgendes Menü:

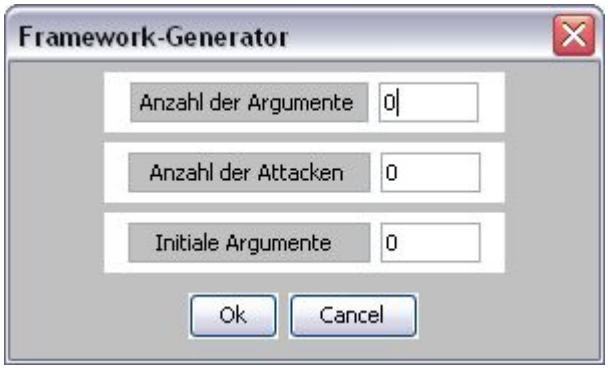


Fig. 3.24: Framework - Generator

Hier lassen sich jetzt die Eckdaten des gewünschten Frameworks eingeben.
Bei den Eckdaten handelt es sich um die Anzahl der Argumente, die Anzahl der Attacken zwischen diesen und die Anzahl der Initial – Argumente.
Die Initial – Argumente beziehen sich dabei auf die Anzahl der erzeugten Argumente und werden nicht extra generiert. Sie stellen eine Teilmenge der gesamten Argumente dar. Wird also ein Framework mit 10 Argumenten erzeugt, können (ohne Attacken) maximal 10 dieser Argumente als Initial – Argumente generiert werden.
Weiter handelt es sich bei der Anzahl der Initial – Argumente nicht um eine genaue Vorgabe, wie viele Argumente Initial – Argumente sein sollen, sondern um eine Angabe über die Mindestanzahl dieser. Ein Framework mit 10 Argumenten, 1 Attacke und 2 Initial – Argumenten wird also in der Umsetzung 9 Initial – Argumente erhalten, da die Anzahl der Attacken einfach keine Beschränkung auf 2 Initial – Argumente zulässt.
Im Unterschied dazu geben die Anzahl der Attacken und die Anzahl der Argumente eine genaue Vorgabe an das Framework.
Diese Daten werden anschließend auf Sinnhaftigkeit überprüft (Siehe Kapitel 4.1.2.5) und nach Betätigen des Ok – Buttons wird ein passendes zufälliges Framework generiert und gezeichnet. Dabei werden alle Punkte im Sichtbereich der Zeichenfläche platziert. Wurden keine sinnvollen Eingaben getätigt, wird eine Fehlermeldung angezeigt und das Fenster bleibt solange aktiv, bis die fehlerhaften Eingaben geändert werden oder die Aktion mit dem Cancel – Button abgebrochen wird.

4 Aufbau und Struktur des Programms

Dieses Kapitel setzt ein grundlegendes Verständnis der Programmiersprache JAVA, beziehungsweise Objektorientierter Programmierung allgemein, voraus. Empfehlenswerte und per Internet frei einsehbare Nachschlagewerke sind u.a. die API von JAVA[4][6] und das Lehrbuch „JAVA ist auch eine Insel“ [5][7]

Einer der Vorteile objektorientierter Programmiersprachen liegt in der Möglichkeit, einzelne Teile eines Programms unabhängig voneinander zu erstellen und anschließend miteinander interagieren und kommunizieren zu lassen. Dadurch ist es möglich, für einzelne Aspekte eines Programms voneinander (weitestgehend) unabhängige Pakete zu entwerfen, die dann mit geringem Anpassungsbedarf austauschbar sind.

Es erschien mir im Rahmen dieser Arbeit wünschenswert, die Theorie und Algorithmen hinter diesen Frameworks so zu implementieren, dass dabei ein in sich abgeschlossenes Paket entsteht, welches auch von anderen Implementierungen genutzt werden oder per beliebiger GUI (Graphical User Interface, graphische Nutzeroberfläche) und/oder Konsole gesteuert werden kann. Die Programmlogik müsste also möglichst von der GUI getrennt betrachtet werden können und trotzdem mit dieser zusammen funktionieren.
Solch eine Aufteilung lässt sich mithilfe einer MVC – Architektur relativ einfach umsetzen.

4.1 Das MVC – Modell

Das MVC – Modell[5, Kap. 16.15][8] ist ein Entwurfs – Muster, welches genau die oben beschriebene Möglichkeit liefert. Dabei wird das Programm in 3 Komponenten aufgeteilt: **Model**, **View** und **Controller**
Die einzelnen Komponenten übernehmen dabei folgende Aufgaben:

- Der **View** ist für die graphische Darstellung des Programms zuständig.
- Das **Model** übernimmt die Programmlogik.
- Der **Controller** kommuniziert zwischen Model und View.

Der View stellt also die GUI bereit und steuert das Aussehen des Programms, das Model enthält den Teil des Programms, der für Berechnungen und ähnliches zuständig ist und

6 http://download.oracle.com/javase/1.5.0/docs/api/
7 http://openbook.galileocomputing.de/javainsel8/
8 http://openbook.galileocomputing.de/javainsel8/javainsel_16_015.htm

der Controller reagiert auf Benutzereingaben, teilt dem Model mit, was es berechnen soll und stößt den View gegebenenfalls zu einem Update der GUI an.

4.1.1 Das Model (model_AAF)

Dieses Paket wurde so entworfen, dass es komplett alleine bestehen oder an eine beliebige GUI gebunden werden kann. Es besteht aus den 3 Klassen Argument, Labeling und Framework. Fig 4.1 zeigt ein Diagramm dieses Paketes, welches aus Gründen der Übersicht um Funktionen und Attribute beraubt wurde.

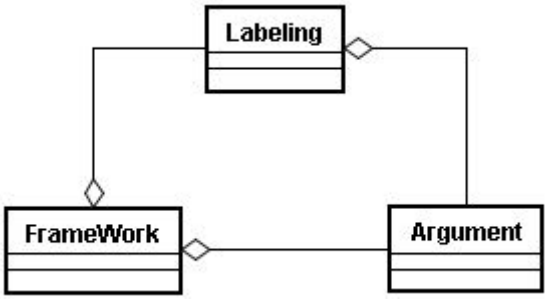


Fig. 4.1: Überblick model_AAF

Die einzelnen Klassen werden im jeweiligen Kapitel genauer dargestellt, wobei ich in diesen Darstellungen auf die get ( ) – und set ( ) Methoden verzichten werde, sofern diese trivial ausfallen. Diese Methoden dienen in der Regel dem Auslesen und Setzen von Variablen und neigen dazu, Diagramme unnötig aufquellen zu lassen. Für eine Variable 'name' gibt es (beispielhaft) eine Methode getName ( ) und eine Methode setName (name), mit denen sich die Variable setzen und lesen lässt. Für 5 Variablen gäbe es also schon 10 (triviale) Methoden, bei mehr Variablen würde sich das Diagramm langsam aber sicher unlesbar gestalten.
Die drei Klassen Argument, Labeling und Framework sind über Aggregations – Beziehung miteinander verbunden. Als Aggregation bezeichnet man in der UML eine Beziehung, die aussagt, dass ein Objekt Teil eines anderen Objektes ist, welches aber (im Gegensatz zur Komposition) nicht von den Teilobjekten existenziell abhängt. Ein Framework ist eine Menge von Labelings und Argumenten, kann aber auch als leeres Framework existieren, als Framework mit leerem Labeling oder als Menge von Argumenten ohne Labeling. Das Labeling ist eine Menge von Argumenten, die ebenfalls leer sein kann.

4.1.1.1 Argument.java

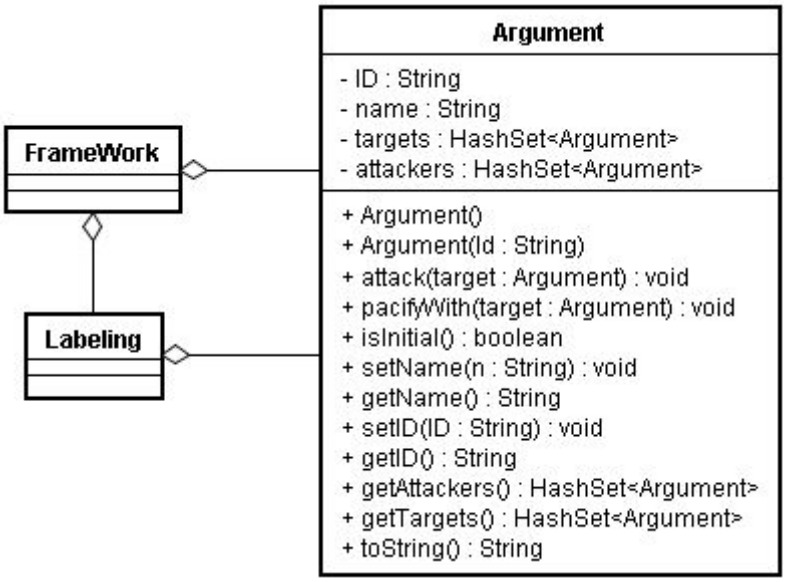


Fig. 4.2: Überblick Argument.java

Die Klasse Argument stellt ein Argument des Frameworks dar. Ich habe den Argumenten eine *ID* gegeben, um es zu ermöglichen mehrere Argumente mit gleichen Namen zu erzeugen und diese trotzdem eindeutig ansprechen zu können.
Der String *name* gibt den Namen des Argumentes wieder. Dadurch kann dem Argument eine individuelle Aussage gegeben werden.
Jedes Argument bekommt jeweils 1 Hashset für die Argumente, welche von diesem Argument attackiert werden (*targets*) und eines für die Argumente, die dieses Argument selbst attackieren (*attackers*). Hashsets [4, S. Hashset] sind Sammlungen von duplikatfreien, ungeordneteten Elementen, die in einer schnellen Hash – basierten Datenstruktur verwaltet werden. Da die Reihenfolge der Argumente keine Rolle spielt und Duplikate unerwünscht sind, erschien mir dieser Collection – Typ am passendsten. Alle Variablen dabei werden im Konstruktor initialisiert um NullPointerExceptions vorzubeugen.

Abgesehen von den selbsterklärenden get( ) – und set( ) – Methoden für jede Variable und der **toString( )** – Methode, welche eine Beschreibung des Argumentes als String zurück gibt, gibt es noch die beiden Methoden **attack (Argument a)** und **pacifyWith (Argument a)**.
Die Methode **attack (Argument a)** fügt das Argument a in das *targets* – Hashset des ausführenden Argumentes ein und das ausführende Argument in das *attackers* – Hashset von Argument a. Die dadurch umgesetzte Attacke zwischen beiden Argumenten kann anschließend wieder mit **pacifyWitch (Argument a)** rückgängig gemacht werden.

Die Methode **isInitial ( )** prüft, ob das Hashset *attackers* des ausführenden Argumentes leer ist und damit, ob das Argument ein Initial – Argument ist.

4.1.1.2 Labeling.java

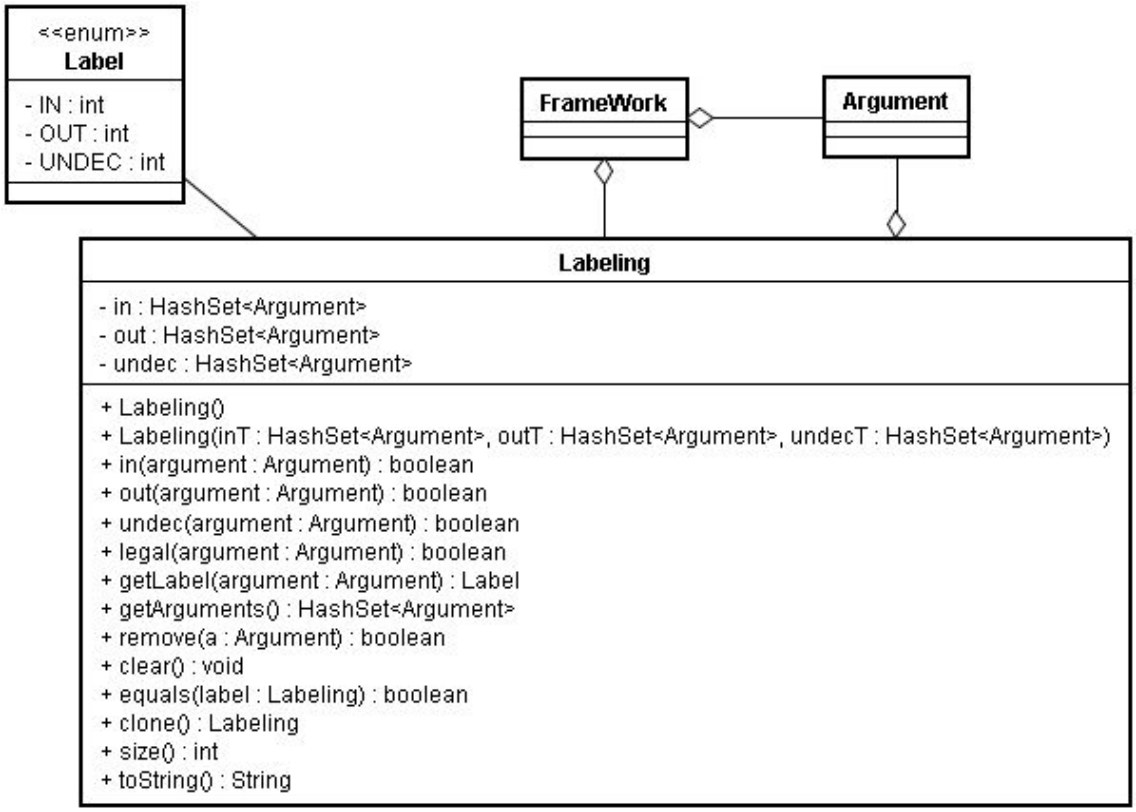


Fig. 4.3: Überblick Labeling.java

Ein Labeling besteht grundsätzlich aus 3 Hashsets:

in: enthält alle Argumente, die IN sind.
out: enthält alle Argumente, die OUT sind.
undec: enthält alle Argumente, die UNDEC sind.

Um die Label auf die erwähnten zu beschränken und Fehlerquellen vorzubeugen wurden die Label mittels einer Aufzählung (enum) auf UNDEC, IN und OUT festgelegt. Diese lassen sich zum Beispiel mit Labeling.IN ansprechen und verhindern Inkonsistenzen bezüglich der Bezeichnung der Label innerhalb des Programms.

Es gibt einen Default – Konstruktor, der ein neues Labeling erzeugt und dabei die Hashsets initialisiert und einen Konstruktor, der 3 Hashsets als Parameter benötigt. Diese 3 Hashsets stellen quasi ein Labeling dar, welches anschließend nach Initialisierung der Hashsets auf das erzeugte Labeling umgesetzt wird. Dadurch lassen sich zum Beispiel All – IN Labelings schnell und einfach mittels
new Labeling ({ alle Argumente }, { }, { }) erzeugen.

Die Methoden **in (Argument a)**, **out (Argument a)** und **undec (Argument a)** arbeiten alle auf analoge Weise. Das übergeben Argument wird aus allen anderen Hashsets entfernt und dem jeweils passenden Hashset hinzugefügt. Dabei wird *false*

zurückgegeben, falls das Argument bereits im passenden Hashset gelagert war.
Mittels **getArguments ( )** wird ein Hashset erzeugt und zurückgegeben, welches alle Argumente der Hashsets *in*, *out* und *undec* – und somit alle Argumente des Labelings – enthält.
Mithilfe von **remove (Argument a)** wird das übergebene Argument aus dem passenden Hashset des Labeling entfernt, **clear ( )** leert alle Hashsets des ausführenden Labelings und damit das gesamte Labeling.
Das Label eines Argumentes wird mittels der Methode **getLabel (Argument a)** berechnet. Dabei wird, je nachdem in welchem Hashset sich das Argument in dem Labeling befindet, das jeweils passende Label zurückgegeben.
Die Legalität eines Argumentes, beziehungsweise dessen Labels, lässt sich mithilfe der Methode l**egal (Argument a)** herausfinden. Diese Methode prüft, ob das übergebene Argument den Voraussetzungen des ihm zugewiesenen Labels genügt.
Zum Vergleichen zweier Labeling dient die Methode **equals (Labeling label)**, welche prüft, ob alle Hashsets des ausführenden und des übergebenen Labelings den gleichen Inhalt haben.
Size ( ) gibt die Größe des Labelings an, indem die Methode die Größe der Hashsets *in*, *out* und *undec* addiert.
Die **toString ( )** – Methode liefert einen String zur Ausgabe des Labelings in der Form „{ { IN – Argumente } {OUT – Argumente} {UNDEC – Argumente} }“ zurück.
Mit **clone ( )** wird eine identische Kopie des ausführenden Labelings erzeugt und zurückgegeben.

4.1.1.3 Framework.java

Fig. 4.4: Überblick Framework.java

Diese Klasse stellt die Implementierung des eigentlichen Frameworks dar, sammelt die erstellten Argumente und enthält die Algorithmen für das Finden von Labelings der einzelnen Semantiken.
Die Algorithmen sind in Kapitel 2.4 dieser Arbeit genauer beschrieben, weshalb ich nicht im Detail auf diese eingehen werde.
Das Framework enthält folgende 4 Hashsets:

arguments :	Dieses Hashset enthält generell alle Argumente des Frameworks.
candidates :	Hier werden die Kandidaten für Preferred und Stable Semantiken gesammelt.
junk :	*junk* entspricht von der Idee her einem Papierkorb, in dem Argumente, die gelöscht werden sollen, zwischengespeichert

werden. Damit lässt sich das Problem umgehen, dass bei Hashsets während des Iterierens über die Elemente keine Elemente gelöscht werden können.

unlabeled : Hier sind die Argumente gesammelt, die im laufenden Vorgang noch kein Label erhalten haben.

Die 2 timestamps vom Typ java.util.Date *start* und *end* dienen der Berechnung der Dauer der Algorithmen und *steps* zählt die nötigen Rekursionsschritte der Algorithmen. Dabei wird vor dem 1. Aufruf der jeweiligen Methode ein timeStamp für st*art* festgelegt und *steps* = 0 gesetzt. Bei jedem rekursiven Aufruf wird *steps* um 1 erhöht und nach Beendigung des Algorithmus ein timeStamp für *end* gesetzt.

Die Methoden **remove (Argument a)** und **add (Argument a)** kümmern sich um das Hinzufügen und Entfernen eines Argumentes zu dem initialisierten Framework. Beim Hinzufügen wird das Argument ohne ein Label angenommen und zusätzlich in *unlabeled* integriert. Beim Löschen eines Argumentes werden auch alle Abhängigkeiten, die von diesem ausgehen, beendet. Das heißt, das Argument wird aus den *targets* und *attackers* aller anderen Argumente entfernt.

Die Methode **getUniversalLabeling ( )** liefert ein Labeling zurück, welches genau allen Argumenten, die in jedem gefundenen Labeling das gleiche Label erhalten haben, eben dieses Label zuweist. Argumente, die in unterschiedlichen Labelings unterschiedliche Label erhalten, werden nicht mit aufgenommen. Dabei wird über *candidates* iteriert und für jedes Argument jedes Labelings in *candidates* das jeweilige Label mithilfe der Methoden **alwaysIn (Argument a)** , **alwaysOut (Argument a)** und **alwaysUndec (Argument a)** auf Konsistenz innerhalb der anderen *candidates* überprüft.
Damit lassen sich die in 3.3 gestellten Beispielfragen 4 und 5 lösen, wie „Welche Argumente sind in jedem Labeling IN ?“. Diese Methode macht natürlich nur Sinn, wenn Labelings für Semantiken berechnet wurden, die mehrere Labelings erlauben (Preferred und Stable).
Verbesserungspotential besteht hier in einem Algorithmus, der diese Aufgabe effizienter als der von mir implementierte Brute – Force Ansatz löst, da bei meiner Lösung Doppelvergleiche nicht unwahrscheinlich sind. Denkbar wäre zum Beispiel, alle bereits als nicht konsistent erkannten Argumente aus späteren Vergleichen auszuschließen.

Interessanter sind die Methoden zum Finden von Labelings, die im Folgenden aufgeführt werden. Die Algorithmen im Detail finden sich in Kapitel 2.4, weshalb ich diese nicht erneut beschreiben möchte, sondern mich darauf konzentrieren werde, zu beschreiben, wie ich die Algorithmen implementiert habe.

Grundsätzlich wurde von mir folgendes Vorgehen gewählt:
Es gibt die Methoden **findGrounded (Labeling condition)**, **findStable (Labeling condition)** und **findPreferred (Labeling condition),** welche von außerhalb mit einem Labeling, welches die Bedingungen (Siehe Kapitel 3.4) an das zu findende Labeling enthalten, aufgerufen werden. Diese Methoden kümmern sich darum, die Algorithmen vorzubereiten, aufzurufen und nachzubereiten. Die Vorbereitung besteht dabei in der Initialisierung benötigter Attribute und der Vorbereitung des Frameworks auf den Algorithmus, das heißt zum Beispiel, dass bei Preferred und Stable Semantiken ein All

– IN Labeling erzeugt wird und bei der Grounded Semantik alle Argumente ihr Label verlieren. Anschließend wird die jeweilige Methode **grounded (Labeling label)**, **stable (Labeling label)** oder **preferred (Labeling label)** aufgerufen, die dann das (die) gesuchte(n) Labeling(s) berechnet. Die Nachbereitung besteht darin, gefundene Labelings mit dem übergebenen *condition* – Labeling mit der Methode **checkCondition (Labeling cond, Labeling label)** abzugleichen und Labelings, die nicht der *condition* entsprechen, wieder zu verwerfen. Hier könnten Performancegewinne erzielt werden, indem der Abgleich mit der *condition* schon während der Algorithmen zum Finden der Labelings als Abbruchbedingung implementiert würde, aber ich beabsichtigte die Algorithmen selbst nicht zu verändern, weshalb ich auf diese Lösung zurückgriff. Letztendlich werden die gefundenen Labelings zurückgegeben.
Um zu verhindern, dass die (nicht ganz triviale) Idee mit dem übergebenen *condition* – Labeling für unnötige Verwirrung sorgt, wurden noch die Methoden **findGrounded ( )**, **findStable ( )** und **findPreferred ( )** ohne ein zu übergebendes Labeling implementiert. Diese rufen die oben erwähnten Methoden mit einem neuen (leeren) *condition* – Labeling auf.
Um die in Kapitel 3.6 beschriebene Option 'Grounded für Preferred und Stable nutzen' umzusetzen, wurden die Methoden **findStable_Grounded(Labeling condition)** und **findPreferred_Grounded(Labeling condition)** implementiert. Diese sind mit den jeweiligen Methoden ohne **_Grounded** identisch, nur mit dem Unterschied, dass das Labeling, mit dem gestartet wird, ein Grounded Labeling ist.

Die Methoden zur Berechnung der Labelings **grounded (Labeling label)**, **stable (Labeling label)** und **preferred (Labeling label)** wurden nach Vorlage implementiert und in Kapitel 2.4 beschrieben.

Die Methode **transitionStep (Labeling label, Argument argument)** führt einen Transition Step (Siehe Kapitel 2.4.2) des übergebenen Argumentes im übergebenen Labeling aus und gibt das neue Labeling zurück.
Die Methode **transitionSequence_terminated (Labeling label)** überprüft ob eine Folge von Transition Steps beendet ist. Dabei wird im implementierten Fall geprüft, ob das übergebene Labeling Argumente enthält, die illegal IN sind. Wenn das nicht der Fall ist, gilt die Transition Sequence als beendet.
Mit **worseLabeling (Labeling label)** wird geprüft, ob ein übergebenes Labeling eine schlechtere Lösung darstellt als ein bereits gefundenes Labeling. Dabei wird mithilfe von **subSet (Labeling ganz, Labeling teil)** geprüft, ob alle Argumente, welche IN sind, bereits in einem anderen Labeling das Label IN erhalten haben.
Die Methode **superIllegalIn (Labeling label)** liefert ein beliebiges Argument innerhalb des übergebenen Labelings, welches super – illegal IN ist (Siehe Kapitel 2.4.2). Wird kein solches Argument gefunden wird einfach *null* zurückgegeben.

Die beschriebenen 3 Klassen bilden das Model des Programms. Sie sind so entworfen, dass sie für sich alleine existieren können, als Paket mittels einer beliebigen GUI oder Konsole gesteuert werden können und auch als Grundsatz für weiterführende Implementierungen geeignet sind. Alle in Kapitel 2 beschriebenen Berechnungen finden in diesen Klassen statt.

Folgender Beispielquellcode zeigt, wie das Modell ohne eine GUI gesteuert werden kann, indem es ein Framework mit 3 Argumenten und einigen Attacken erzeugt und anschließend berechnete Labelings auf der Konsole ausgibt:

```
public static void main(String[]args)
	{
		//Neues Framework
		FrameWork f = new FrameWork();

		//3 Argumente erzeugen
		Argument a = new Argument("1.Argument");
		Argument b = new Argument("2.Argument");
		Argument c = new Argument("3.Argument");

		//Argumente dem Framework hinzufügen
		f.add(a); f.add(b); f.add(c);

		//Attacken zwischen Argumenten hinzufügen
		a.attack(b); a.attack(c); b.attack(c);

		//Ausgabe des Grounded Labeling
		System.out.println(f.findGrounded());

		//Eine weitere Attacke um 2 Preferred Labelings zu
		//ermöglichen
		b.attack(a);

		//Ausgabe der Preferred Labelings
		System.out.println(f.findPreferred());
	}
```

4.1.2 Der View (view_AAF)

Dieses Paket stellt die graphische Darstellung des Programms bereit und wurde so entworfen, dass es die Arbeit mit Frameworks auf übersichtliche und intuitive Weise ermöglicht.
Ich habe es dabei vorgezogen, die GUI selbstständig mithilfe der von den Paketen java.awt und javax.swing bereitgestellten Mittel zu implementieren und verzichtete auf die Verwendung bereits vorhandener GUI – Pakete für JAVA. Dadurch konnte ich sicherstellen, dass kein unnötiger Overhead das Programm belastet und ich leicht Anpassungen vornehmen kann.

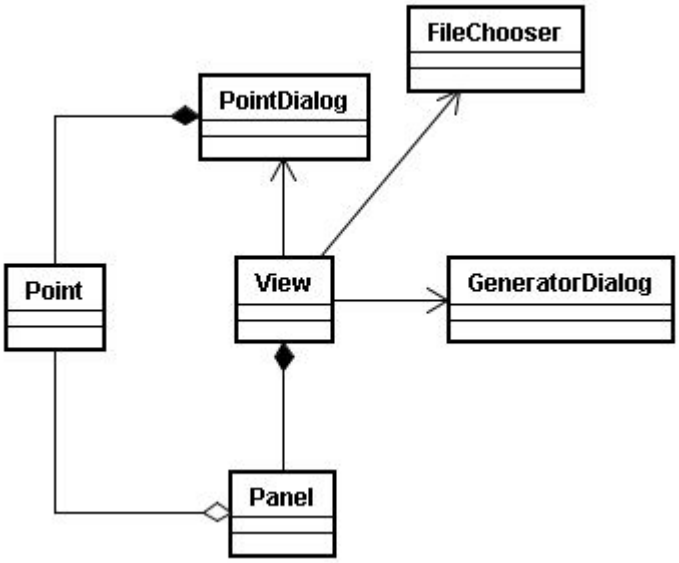


Fig. 4.5: Überblick view_AAF

Die Hauptkomponente dieser Klasse bildet der View, der das Panel, auf welchem die graphische Darstellung des Frameworks stattfindet, als Teilkomponente besitzt. Zwischen diesen herrscht eine Kompositions – Beziehung, das heißt, ähnlich der Aggregation, das Panel ist ein Teil des View. Allerdings ist der View diesmal existentiell vom Panel abhängig beziehungsweise das Panel ein fester Bestandteil des View. Die Klassen GeneratorDialog, FileChooser und PointDialog stellen vom View instantiierte Dialoge für Benutzereingaben dar. Die Klasse Point ist als Trägerklasse der Argumente zu verstehen und ein fester Bestandteil (Komposition) des PointDialog, jedoch nur ein loser Bestandteil (Aggregation) des Panels.
Bei den folgenden Diagrammen habe ich, zur einfachen Einordnung der jeweiligen Klasse, Klassen anderer Pakete mit eingefügt. Diese Grenzen zu jeweils anderen Paketen werden durch senkrechte Linien auf der jeweiligen Beziehung dargestellt.

Abgesehen von der Point – Klasse wurde jeder dieser Klassen ein Controller zur Seite gestellt. Diese sind Teil des Controller – Paketes und werden ab Kapitel 4.1.3 erläutert.

4.1.2.1 Point.java

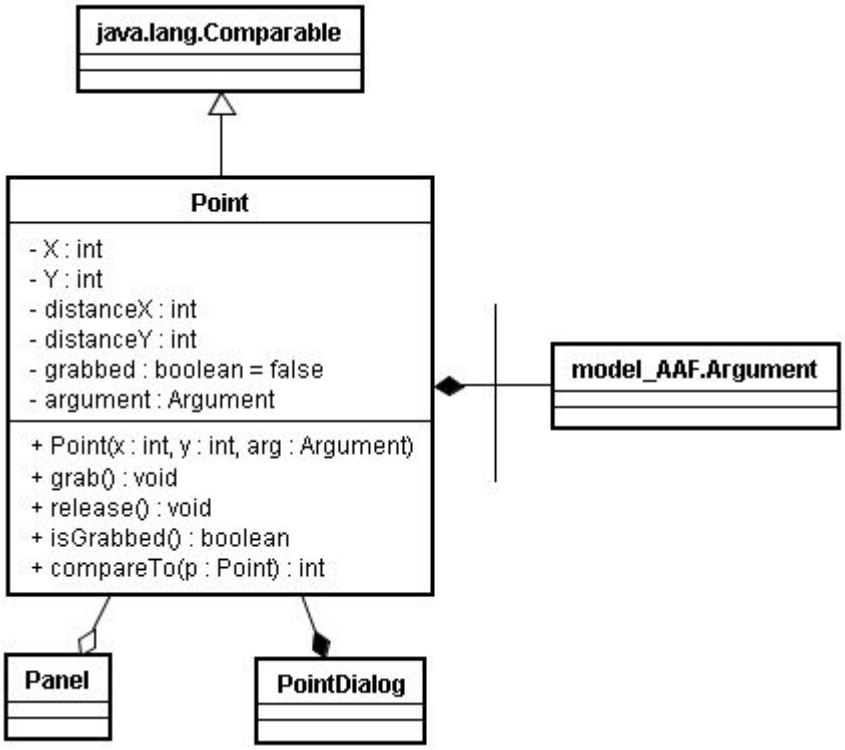


Fig. 4.6: Überblick Point.java

Diese Klasse implementiert einen Punkt, der als Träger eines Argumentes dient und besteht hauptsächlich aus einigen Attributen und jeweils passenden get ( ) - und set ( ) - Methoden.

argument	Das von dem Punkt dargestellte Argument.
x, *y*	Koordinaten zum Platzieren auf einer 2D – Ebene.
grabbed	True, wenn das Argument vom Nutzer aktiviert ist.

Die Attribute *distanceX* und *distanceY* werden intern benötigt, um das Bewegen der Punkte auf dem Panel zu ermöglichen[9].
Mithilfe von **grab ( )** und **release ( )** wird der Punkt aktiviert oder wieder deaktiviert, wobei dabei das Attribut *grabbed* jeweils den passenden Boolean – Wert erhält, der dann mit **isGrabbed ( )** ausgelesen kann.
Das *argument* stellt für den Punkt die Schnittstelle zum Paket model_AAF dar und macht den Punkt zum Träger eines Argumentes.

Es erwies sich bei der Implementierung des Panels als sinnvoll, Punkte miteinander vergleichen zu können, weshalb diese Klasse das Interface java.lang.Comparable implementiert. Dieses Interface dient dazu, Objekte mithilfe der zu implementierenden Methode **compareTo(Objekt o)** zu vergleichen. Damit kann festgelegt werden, wann das ausführende Objekt gleich (Rückgabewert 0), kleiner (Rückgabewert – 1) oder größer (Rückgabewert 1) einem übergebenen Objekt ist. Bei dieser Implementierung

9 Siehe Kapitel 4.1.3.3

gilt ein Punkt kleiner/größer/gleich einem anderen Punkt, wenn die ID des getragenen Argumentes größer/kleiner/gleich der ID des Argumentes des anderen Punktes ist.

4.1.2.2 Panel.java

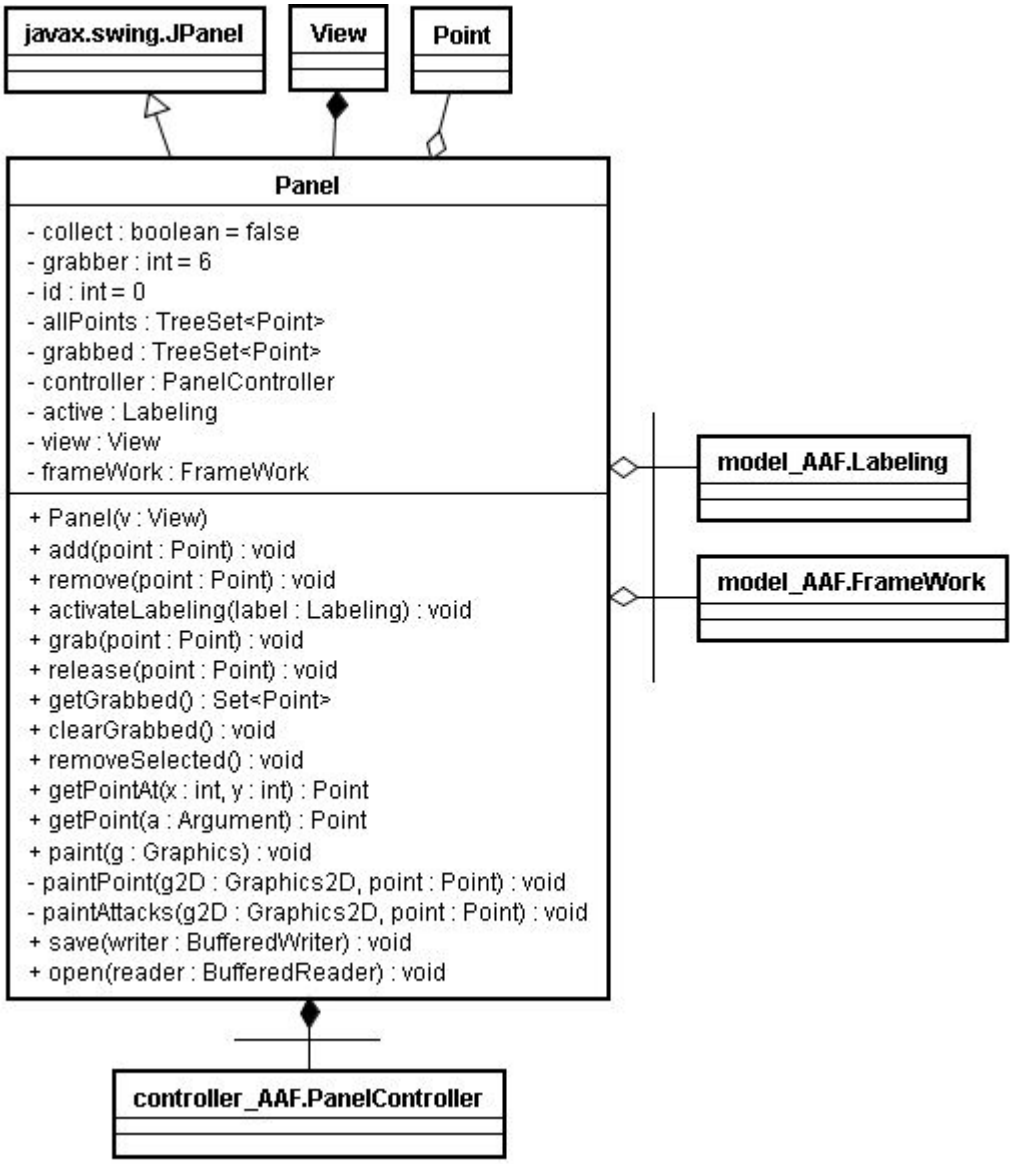


Fig. 4.7: Überblick Panel.java

Hierbei handelt es sich um die Arbeitsfläche, auf der die Argumente erzeugt, angesprochen und dargestellt werden. Es handelt sich also um die graphische Repräsentation des eigentlichen Frameworks. Daher ist es sinnvoll, dem Panel das Framework als Attribut zur Verfügung zu stellen und alle Methoden des Frameworks über das Panel zu steuern. So werden die Methoden des Framework – Paketes zum Hinzufügen und Entfernen parallel mit dem Hinzufügen und Entfernen von Punkten auf dem Panel aufgerufen. Diese Methoden heißen hier (wie auch sonst)
add (Point p) und **remove (Point p)** und geben die Änderung direkt an das Framework weiter.

Beim Hinzufügen eines Punktes wird dessen Argument eine eindeutige *ID* zugeordnet.

Das Laden und Speichern eines Frameworks wird ebenfalls von dieser Klasse übernommen. Dies erschien mir sinnvoll, da ich das Aussehen des Frameworks beibehalten wollte und damit sowohl die Koordinaten des Punktes und die Daten des Argumentes gemeinsam speichern musste, welche erst auf Ebene des Panels zusammenlaufen. Die zugehörigen Methoden sind
open (BufferedReader reader) und **save (BufferedWriter writer)**. Die Dateien können mit einem einfachen Texteditor ausgelesen werden und sind dreigeteilt aufgebaut, wobei jeweils eine Zeile '*' als Trennzeichen zwischen den Teilen verwendet wird und ein Tab als Trennzeichen innerhalb einer Zeile:

Teil 1 – Die Argumente
ID name
Teil 2 – Die Attacken
ID des Angreifers ID von Ziel 1 ID von Ziel 2 (…)
Teil 3 – Die Positionen
ID des Argumentes x – Koordinate y – Koordinate

Beim Speichern und Laden erwies sich eine geordnete Reihenfolge der Punkte als vorteilhaft. Um diese Sortierung zu ermöglichen, entschied ich mich die Elemente in Treesets zu speichern, welche man als Hashsets mit geordneter Reihenfolge verstehen kann. Treesets setzen voraus, dass die gesammelten Objekte miteinander vergleichbar sind, was wiederum erreicht wird, indem die Elemente das Interface Comparable implementieren, welches mithilfe der in 4.1.2.1. beschriebenen Methode **compareTo (Object o)** eine beliebige Art der Sortierung ermöglicht.
Diese Treesets sind *allPoints*, welches alle Punkte enthält und *grabbed*, welches alle Punkte enthält, die vom Nutzer aktuell aktiviert sind.
Zum (De –)Aktivieren von Punkten werden die Methoden **grab (Point p)** und **release (Point p)** aufgerufen, die die Änderungen an *grabbed* umsetzen und die Information an den jeweiligen Punkt weitergeben. Solange die Leertaste gedrückt ist, wird dabei *collect* zu true ausgewertet und es lassen sich mehrere Punkte aktivieren. Andernfalls – wenn *collect* false ist – wird immer nur der aktuell angeklickte Punkt aktiviert.
Mit **removeSelected ( )** werden alle aktuell aktivierten – das heißt in *grabbed* befindlichen – Punkte von dem Panel gelöscht.

Dem Panel wird ein MouseListener in Form eines PanelControllers als eigener *controller* zugewiesen. Dieser wird in Kapitel 4.1.3.3 näher erläutert.

Da nur ein Labeling gleichzeitig dargestellt werden kann, muss dem Panel das aktuell zu zeichnende Labeling bekannt gemacht werden. Dies geschieht mit dem Labeling *active*, welches über die Methode **activateLabeling (Labeling label)** festgelegt werden kann.

Es gibt 2 Methoden um einen Punkt des Panels zu finden:
Mit **getPointAt (int x, int y)** lässt sich ein Punkt anhand seiner Koordinaten finden, was wichtig ist, um die Punkte per Mausklick zu aktivieren. Dabei werden die angeklickten Koordinaten mit den horizontalen und vertikalen Grenzen aller Punkte

verglichen und ein passender Punkt sofort zurückgegeben. Wird kein Punkt gefunden, nachdem über alle Punkte iteriert wurde, wird *null* zurückgegeben.
Die zweite Möglichkeit besteht darin einen Punkt anhand seines Argumentes zu finden. Diesen Weg stellt die Methode **getPoint (Argument a)** bereit, die das übergebene Argument mit den Argumenten aller Punkte abgleicht und den passenden Punkt zurück gibt. Wird hier nichts gefunden, wird wieder *null* zurückgegeben.
Die etwas umständliche Arbeitsweise der Methode **getPoint (Argument a)** ließe sich vermeiden, indem jedem Argument der jeweilige Punkt zugewiesen wird, der dann einfach angesprochen werden kann. Um das Model des Programms unabhängig vom Rest zu machen, sah ich davon ab, die Klasse Point in der Klasse Argument bekannt zu machen. Dies würde die Klasse Argument von der Klasse Point abhängig machen.

Weiter gibt es noch mehrere Methoden, die sich um das Zeichen des Frameworks kümmern.
Die Methode **paint (Graphics g)** ist eine Standard – Methode von JAVA, in der festgelegt werden kann, was beim Zeichnen dieses Objektes passieren soll, und wird mit jedem Aufruf von **repaint ( )** ausgeführt. Dabei habe ich das erzeugte Graphics – Objekt in ein Graphics2D – Objekt umgewandelt, da dieses mehr Möglichkeiten, wie zum Beispiel Kantenglättung (Anti – Aliasing), bietet. Anschließend wird die gesamte Fläche weiß überzogen und das aktive Labeling in die linke obere Ecke geschrieben. Dann werden alle Punkte, die nicht vom Nutzer aktiviert sind, mit der Methode **paintPoint (Graphics2D g, Point p)** gezeichnet, welche mit der Farbe, die für das Label des Argumentes festgelegt wurde, einen Kreis bei den Koordinaten des Punktes zeichnet.
Als nächstes werden alle Punkte, die in *grabbed* enthalten sind, von dieser Methode bearbeitet, mit dem Unterschied, dass bei diesen vorher ein gelber etwas größerer Kreis gezeichnet wird, um eine Umrandung zu simulieren. Nachdem so alle Punkte gezeichnet wurden, wird noch einmal über alle Punkte iteriert und für jeden Punkt die jeweiligen Attacken mit der Methode **paintAttacks (Graphics2D g, Point p)** gezeichnet. Diese Methode holt sich die *targets* des Argumentes, welches an dem übergebenen Punkt gebunden wurde, und zeichnet einen Pfeil vom übergebenen Punkt aus zu jedem Punkt, dessen Argument in den *targets* vorkommt.
Es bestünde noch die Möglichkeit, die **paint ( )** – Methode so zu implementieren, dass die Attacken im gleichen Zug wie die Punkte gezeichnet werden und so mehrmaliges Iterieren über alle Punkte vermieden wird. Allerdings ergibt sich dann unter Umständen eine unvorteilhafte Darstellung, da abhängig von der Reihenfolge die Attacken, die ein Argument erfährt, gezeichnet würden, bevor das attackierte Argument gezeichnet ist. Damit würde die Pfeilspitze der Attacke von dem Argument überzeichnet. Das Argument würde einfach ausgedrückt über den Pfeil gezeichnet und diesen verdecken. Um das zu vermeiden und da der Mehraufwand sich in vertretbaren Grenzen hielt, entschied ich mich dafür, mehrmaliges Iterieren in Kauf zu nehmen.

Abgesehen von den üblichen get ( ) – und set ( ) – Methoden für die Attribute gibt es noch ein Element, welches Erwähnung finden sollte: Um eine Kommunikation mit dem View zu ermöglichen, wird dieser als Attribut mit aufgenommen. Damit lassen sich die Methoden des View mit aufrufen, um zum Beispiel die Konsole bei relevanten Änderungen wieder auf den neuesten Stand zu bringen. Eigentlich sollte das über die Controller ablaufen, aber dieser Weg stellt sozusagen eine Abkürzung dar.

4.1.2.3 PointDialog.java

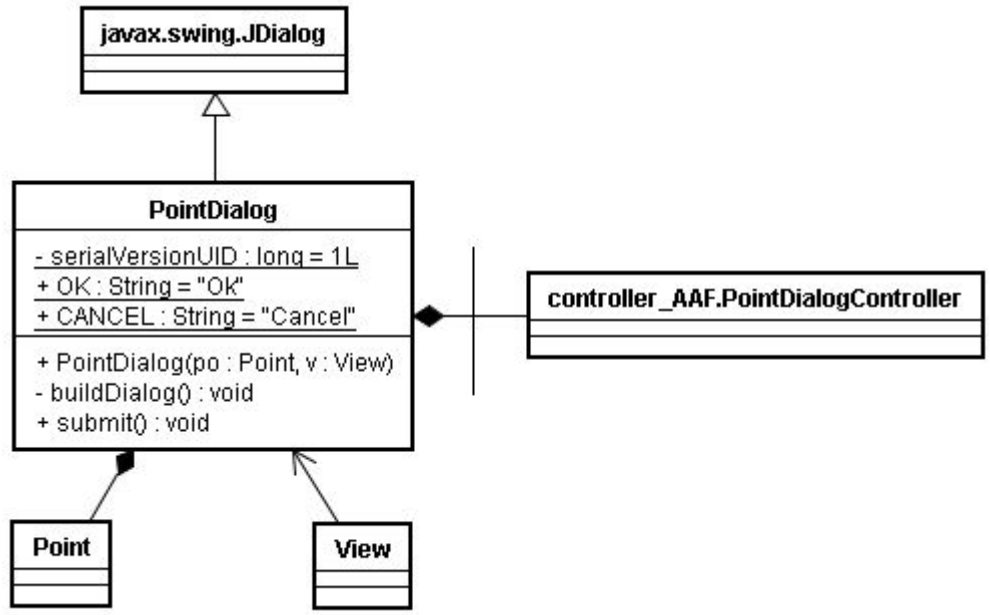


Fig. 4.8: Überblick PointDialog.java

Der vom PanelController bei Doppelklick auf ein Argument instantiierte PointDialog ist ein einfacher, von der JAVA – Klasse JDialog abgeleiteter, Dialog, der ein Textfeld bietet, in dem ein Name für ein Argument eingegeben werden kann. Dieser Klasse ist ein PointDialogController (Kap 4.1.3.5) zugewiesen, der die Eingabe prüft und umsetzt.

4.1.2.4 View.java

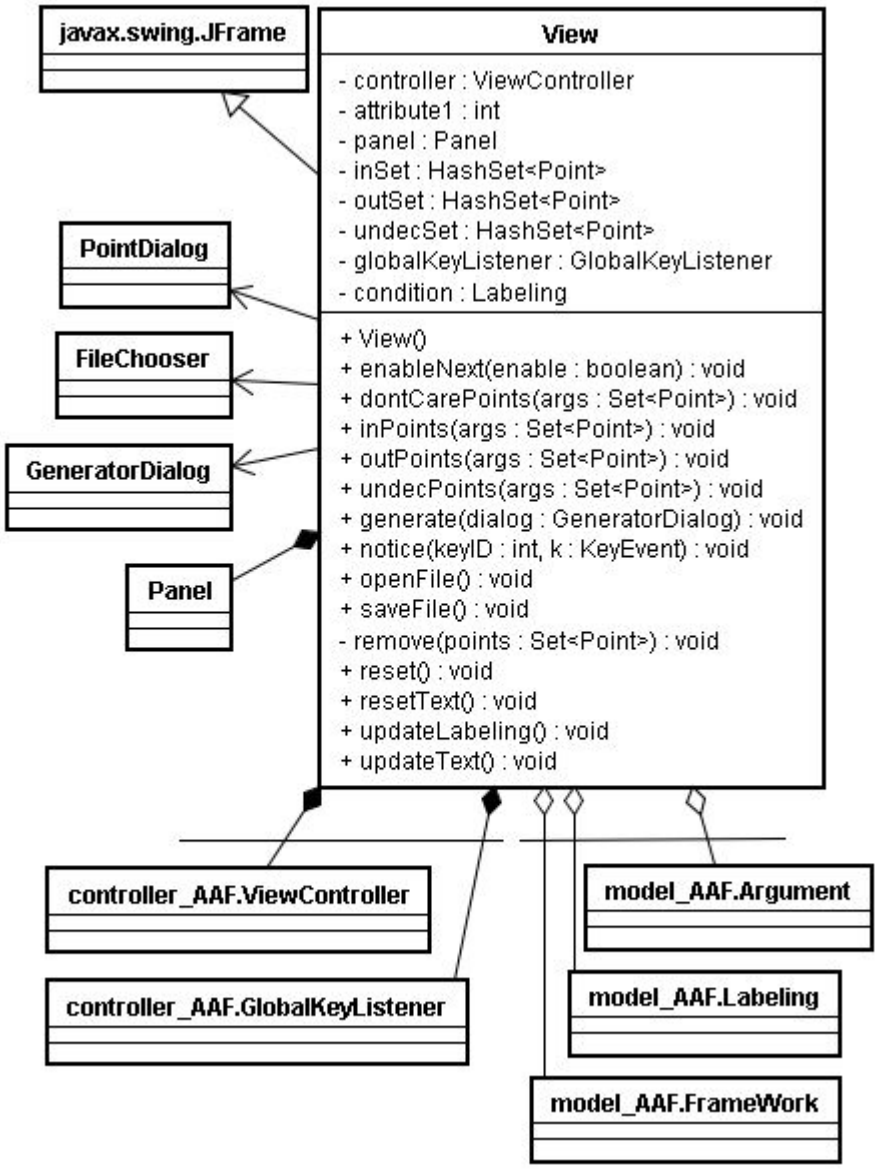


Fig. 4.9: Überblick View.java

Der View stellt die oberste Klasse in der Programmhierarchie dar, in der alle anderen Bestandteile zusammenlaufen und direkt oder indirekt instantiiert werden. Daher lässt sich das Programm auch mit Instantiierung eines neuen Views starten.
Grundsätzlich werden in dieser Klasse die Struktur der GUI festgelegt und die Mittel zur Steuerung des Programms über die GUI bereitgestellt.
JAVA stellt mit JMenuBar ein Paket bereit, welches es leicht macht, eine Menübar zu implementieren, über die sich viele Funktionen sinnvoll abrufen lassen. Diese Menubar stellt dem Nutzer die Funktionen zum Laden/Speichern/Zurücksetzen des Frameworks und die Möglichkeit, Optionen[10] zu verändern und den Framework – Generator[11] zu starten, bereit.

10 Siehe Kapitel 3.6
11 Siehe Kapitel 3.7

Diese und alle anderen Bestandteile der GUI, also die Konsole, das Menü zum Finden von Labelings und zum Erstellen des Bedingungslabelings wurde mithilfe der von JAVA bereitgestellten Möglichkeiten des Paketes javax.swing erstellt. So sind die Konsole und die Menübestandteile als JPanel implementiert, wobei die Konsole und die Ausgabefelder des Bedingungsmenüs zu einem JScrollPane erweitert wurden, um bei umfangreicher Ausgabe die Möglichkeit zu bieten den Text zu scrollen. Bei den Buttons handelt es sich um JButtons. Die Zeichenfläche ist das in 4.1.2.2 beschriebene Panel zum Darstellen des Frameworks. Um den Quellcode übersichtlich zu gestalten, habe ich den Aufbau der GUI in verschiedene kleinere Funktionen gesplittet, die hier aber nicht näher Erwähnung finden sollen.
Ebenfalls nicht im Diagramm dieses Kapitels enthalten sind einige Strings, die zur einheitlichen Verarbeitung als statische Strings implementiert wurden. Dabei handelt es sich um 20+ statische Variablen, die das Diagramm aufquellen hätten lassen, ohne dabei einen echten Informationsgehalt zu bieten.

Um mithilfe des Bedingungsmenüs ein Labeling zu erstellen, welches dann zur weiteren Verarbeitung genutzt werden kann, wurden 3 HashSets als Attribute implementiert. Diese HashSets *inSet*, *outSet* und *undecSet* sammeln und entfernen die Argumente parallel zu den Nutzereingaben über das Bedingungsmenü. Dies geschieht mit den Methoden **inPoints (Set<Point>args), undecPojnts (Set<Point> args)** und **outPoints (Set<Point> args).** Bei Betätigen der Buttons zum Finden von Labelings wird aus diesen HashSets das Bedingungslabeling erzeugt und an das Framework des Panels weitergereicht.

Um die Nutzereingaben verarbeiten zu können, werden 2 Listener als Controller benötigt. Der g*lobalKeyListener* achtet auf allgemeine Tastatureingaben, die er dann an die Methode **notice (int keyID, KeyEvent k)** zur Verarbeitung weiterreicht. Damit lassen sich beliebige Tastatureingaben verarbeiten, im Moment wird aber nur auf die Taste gehört, die zum Aktivieren mehrerer Argumente dient.
Der zweite *controller* ist ein ViewController, der an jeden Button und an jedes MenuItem der Menubar gebunden wird und bei Aktivieren eines der Elemente das jeweilige Kommando umsetzt.
Da es mühsam ist, Frameworks von Hand zu erstellen, wurde eine Methode implementiert, die es ermöglicht, zufällige Frameworks mit bestimmten Vorgaben zu erzeugen. Diese Methode nennt sich **generate (GeneratorDialog dialog).** Der übergebene (und im nächsten Kapitel beschriebene) GeneratorDialog stellt hierbei die benötigten Daten zur Verfügung. Benötigt werden die Anzahl der zu erzeugenden Argumente, die Anzahl der zu erzeugenden Attacken zwischen diesen und die (Mindest –) Anzahl der Initial – Argumente. Der benötigte GeneratorDialog wird dabei vom ViewController bei Aktivierung des MenuItems 'FrameWork – Generator' unter 'Werkzeuge' instantiiert, nimmt die Nutzereingaben entgegen und übergibt sich selbst an diese Methode. Sind die Daten sinnvoll, werden zuerst so viele Punkte mit Argumenten und zufälligen Koordinaten innerhalb des sichtbaren Bereichs des Panels erzeugt, wie durch die Anzahl der Argumente vorgegeben. Dabei wird nebenbei ein HashSet *initial* geführt, welches solange mit den gerade erzeugten Punkten gefüllt wird, bis dessen Größe die Anzahl der Initial – Argumente erreicht hat. Sind alle Punkte erzeugt, werden solange Attacken zwischen 2 immer zufällig neu gewählten Argumenten ausgeführt, bis die Anzahl der Attacken der Vorgabe entspricht. Eine Attacke zwischen 2 Punkten

(Argumenten) a und b findet dabei nur statt, wenn a nicht bereits b attackiert und b nicht in dem Hashset *initial* vorhanden ist.
Durch dieses Vorgehen wird ein Framework mit den geforderten Eigenschaften erzeugt.

4.1.2.5 GeneratorDialog.java

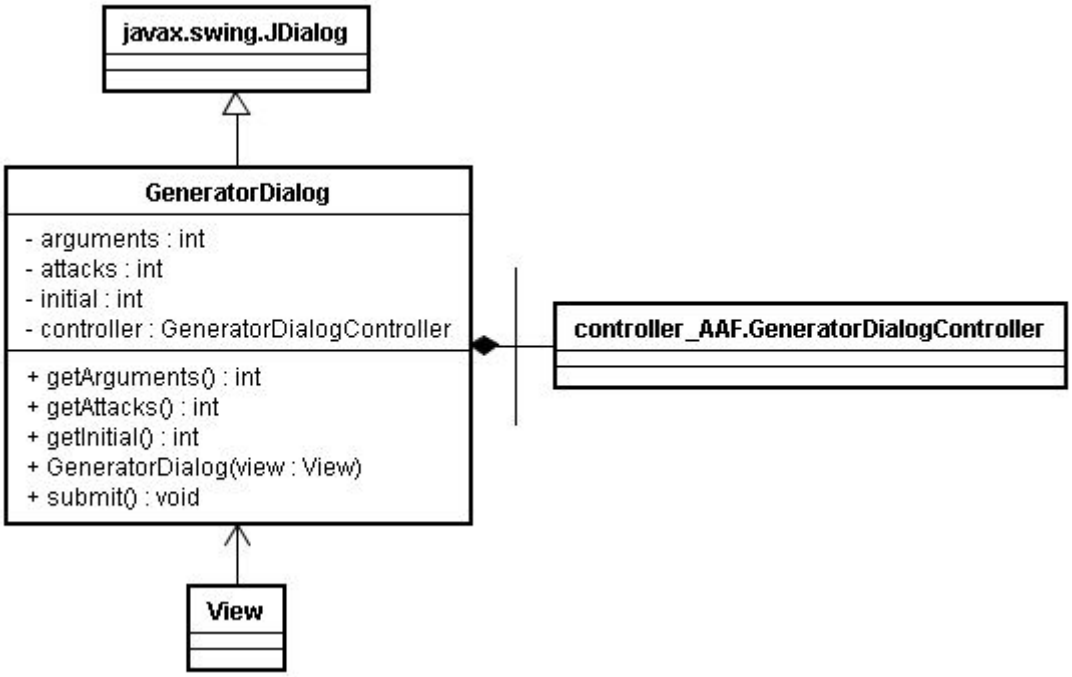


Fig. 4.10: Überblick GeneratorDialog.java

Diese wieder von JDialog abgeleitete Klasse wird vom ViewController instantiiert und soll dem Nutzer die Möglichkeit bieten, Eigenschaften des zu generierenden Frameworks festzulegen. Die Basis stellt wieder ein JPanel, auf dem Textfelder für die Eingabe der Anzahl der Argumente, der Attacken und der Initial – Argumente angebracht wurden. Den JButtons (nicht im Diagramm enthalten) zum Abbrechen und Bestätigen der Eingabe wurde ein GeneratorDialogController zugewiesen, der im Falle eines Abbruchs den Dialog wieder verwirft und im Falle der Bestätigung die Methode **Submit ( )** des GeneratorDialog aufruft.
Diese Methode prüft die Eingabe auf Umsetzbarkeit und gibt den GeneratorDialog an die Methode generate (GeneratorDialog dialog) des View weiter, der die eingegebenen Daten mithilfe get( ) – Methoden des GeneratorDialoges extrahieren kann. Im Falle einer nicht umsetzbaren Eingabe wird ein Popup mit einer Fehlermeldung angezeigt und die Bestätigung verweigert. Für die Umsetzbarkeit wird dabei geprüft, ob die Anzahl der Attacken einen bestimmten Maximalwert nicht überschreitet.
Dieser Maximalwert lässt sich relativ leicht herleiten:

> Grundsätzlich kann jedes Argument jedes Argument angreifen, auch sich selbst. Bei n Argumenten sind also n*n Attacken möglich.
> Jedes Initial – Argument kann von keinem Argument angegriffen werden, deshalb muss die Anzahl der Initial – Argumente von den Argumenten, die angegriffen werden können, abgezogen werden.
> Bei i Initial –Argumenten sind also für n>0 Argumente n * (n – i) Attacken möglich.

Ist die Anzahl der Attacken höher als dieser Wert wird die Verarbeitung verweigert und ein PopUp am oberen Rand des Fensters angezeigt, welches darauf hinweist, dass die Eingabe nicht umsetzbar ist. Dies geschieht ebenfalls, wenn anstatt einer Zahl ein Wort eingegeben wird oder die Anzahl der Initial – Argumente größer der Anzahl der Argumente ist.

4.1.2.6 FileChooser.java

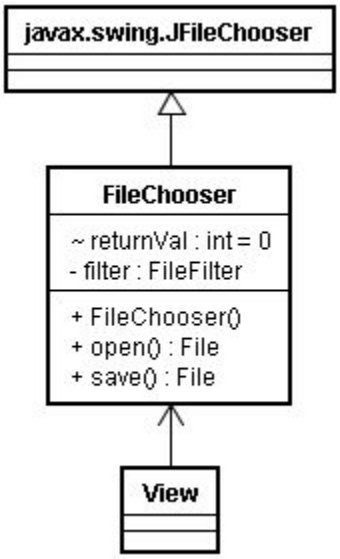


Fig. 4.11: Überblick FileChooser.java

Diese von javax.swing.JFileChooser abgeleitete und vom View instantiierte Klasse liefert das Menü zum Öffnen und Speichern von Frameworks.
Zur Übersicht wurde ein FileFilter *filter* eingerichtet, der nur .aaf – Dateien anzeigt und auch deaktiviert werden kann. Nach Auswahl der Dateien werden die **save ( )** – oder **open ( )** – Methoden aufgerufen, die die gewählte Datei als java.io.File zur Weiterverarbeitung an den View übergeben. Ein Controller ist hierbei nicht nötig, javax.swing.JFileChooser kümmert sich selbst um die Verarbeitung.

4.1.3 Die Controller (controller_AAF)

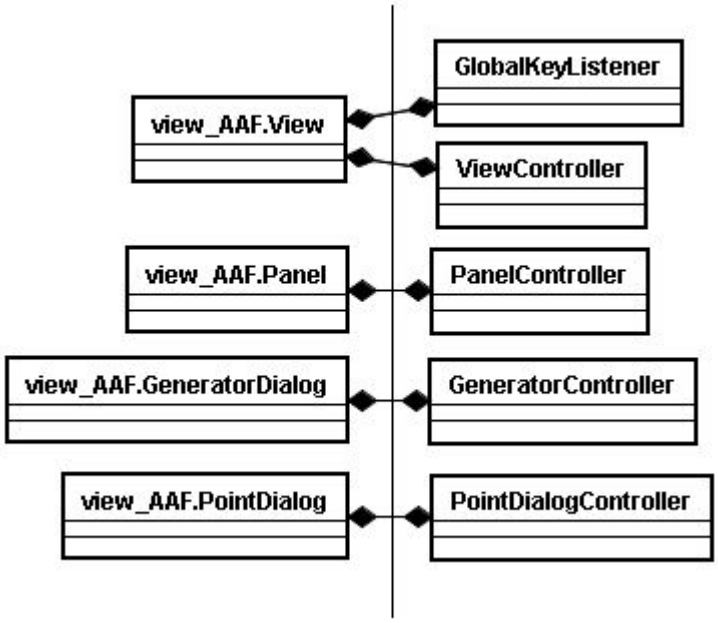


Fig. 4.12: Überblick controller_AAF

Dieses Paket stellt eine Reihe von Controllern in Form von Mouse –, Key – und ActionListenern für das Panel zur Verfügung. Für jede Klasse des Panels, abgesehen von der Klasse Point, wurde dabei ein passender Controller programmiert, um eine übersichtliche und auf die zugeordnete Klasse zugeschnittene Aufgabenverteilung zu gewährleisten. Innerhalb des Paketes sind die Controller voneinander unabhängig und stehen in keinerlei Verbindung zueinander. Die Verbindung zum Paket Model__AAF findet über die jeweilige Komponente von View_AAF statt, weswegen diese Verbindungen hier nicht direkt aufgeführt sind.
Grundsätzlich wurden die Controller so implementiert, dass zwischen dem Controller und der zugewiesenen Klasse eine beidseitige Aggregation vorliegt. Das heißt, der Controller trägt die jeweilige Klasse als Attribut und gleichzeitig trägt die Klasse den jeweiligen Controller als Attribut.
Dadurch wird eine 1:1 – Zuordnung zwischen diesen Klassen erreicht, die eine beidseitige Kommunikation ermöglicht.
Teilweise handelt es sich bei den folgenden Diagrammen um Mehrfachvererbungen, die JAVA eigentlich nicht erlaubt. Diese Einschränkung gilt allerdings nicht für Interfaces, mit denen eine Mehrfachvererbung umgesetzt werden kann. Diese erkennt man in den Diagrammen an dem Stereotype <<implements>>.

4.1.3.1 ViewController.java

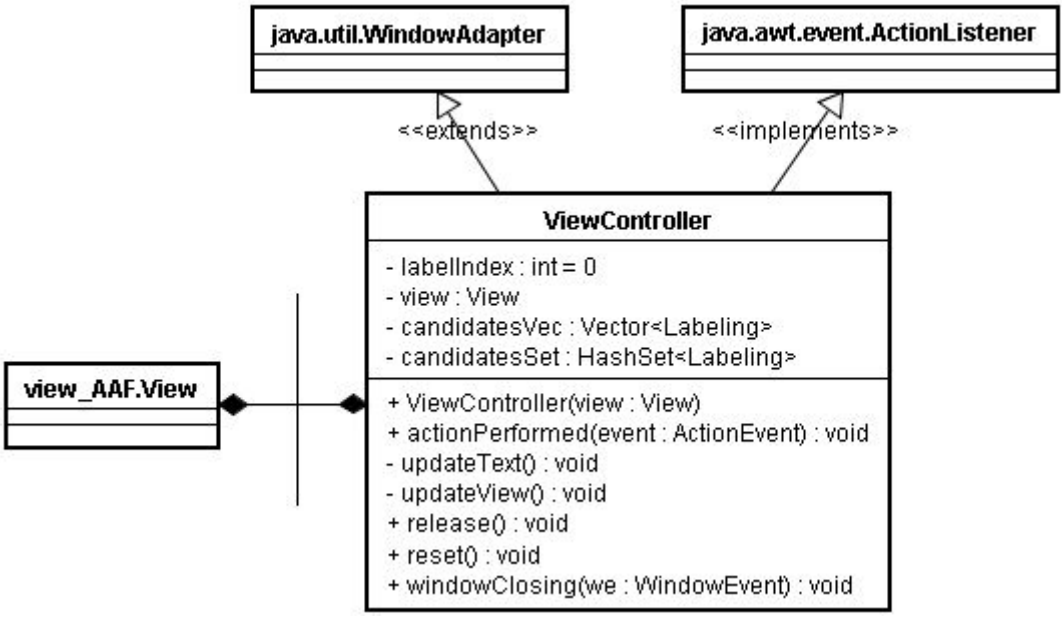


Fig. 4.13: Überblick ViewController.java

Diese Klasse stellt einen Controller für den View bereit. Er ist von java.util.WindowAdapter abgeleitet, womit zum Beispiel über die Methode **windowClosing (we : WindowEvent)** festgelegt werden kann, was bei Beenden des Programms passieren soll, und der View allgemein als Hauptfenster festgelegt wird. **windowClosing (we : WindowEvent)** zerstört dabei den *view* und den controller selbst mithilfe der **release ( )** – Methode.
Die beiden Collections *candidatesVec* und *candidatesSet* dienen dem Verarbeiten gefundener Labelings. Dabei wird einmal für einfache Aufgaben in einem HashSet gesammelt und zusätzlich für das Durchschalten der zu zeichnenden Labelings in einem Vector, welcher das direkte Ansprechen der Elemente mithilfe eines Indexes in Form einer Integer – Zahl ermöglicht.
Die Methode **reset ( )** dient dem Zurücksetzen des ViewControllers und leert sowohl *candidatesVec* als auch *candidatesSet*.

Die Klasse implementiert das Interface java.awt.event.ActionListener. Dieses Interface fordert eine Implementierung der Methode **actionPerformed (ActionEvent event)** und ermöglicht es diese Klasse als ActionListener an beliebige JButtons zu hängen. Bei Aktivierung des Buttons wird dabei der String, welcher dem JButton zugewiesen wurde, mit der Methode **event.getActionCommand ( )** dem Controller zugänglich gemacht und kann weiterverarbeitet werden. Dieser String wird in der Variable *command* gespeichert. Über verschiedene If – Anfragen innerhalb der Methode **actionPerformed (ActionEvent event)** kann nun festgelegt werden, welcher *command* welche Funktionen nach sich zieht. Da die Strings, die den JButtons zugewiesen wurden, alle als statische Strings der View – Klasse implementiert wurden, lassen sich diese auch über diese statischen Variablen ansprechen und Änderungen dieser Strings am View beeinflussen die Funktionalität des Controllers nicht. Im Folgenden sind alle Strings, die in diesem Controller als *command* ankommen können, und deren Auswirkungen kurz geschildert:

View.OPEN / View.SAVE / View.NEW

→ Veranlasst den View die Methode **openFile ( ) / saveFile ( ) / reset ( )** aufzurufen.

View.NEXT

→ Erhöht den *labelIndex*, der vorgibt, welches Labeling aus dem *candidatesVec* gezeichnet werden soll.

View.IN / View.OUT / View.UNDEC

→ Ruft die Methoden **inPoints (HashSet<Point> set) / outPoints (HashSet<Point> set) / undecPoints (HashSet<Point> set)** des *view*, und übergibt ihnen das *grabbed* des *panels* des *view.* Damit werden die Argumente aller aktivierten Punkte für das Bedingungslabeling eingeordnet.

View.DONTCARE

→ Ruft die Methoden **dontCarePoints (HashSet<Point> set)** des *view* auf und übergibt ihnen das *grabbed* des *panels* des *view.* Damit werden alle Argumente aller aktivierten Punkte aus dem Bedingungslabeling entfernt.

View.TEXTUPDATE / View.UNIVERSAL / View.ANTI_ALIAS / View.SHOWNAME

→ Schaltet die Boolean –Variablen textUpdate / universal / antiAliasing / showName der statischen Options – Klasse[12] um.

View.GENERATOR

→ Startet einen neuen GeneratorDialog, mit dem zufällige Frameworks erzeugt werden können[13]

View.GROUNDED / View.PREFERRED / View.STABLE

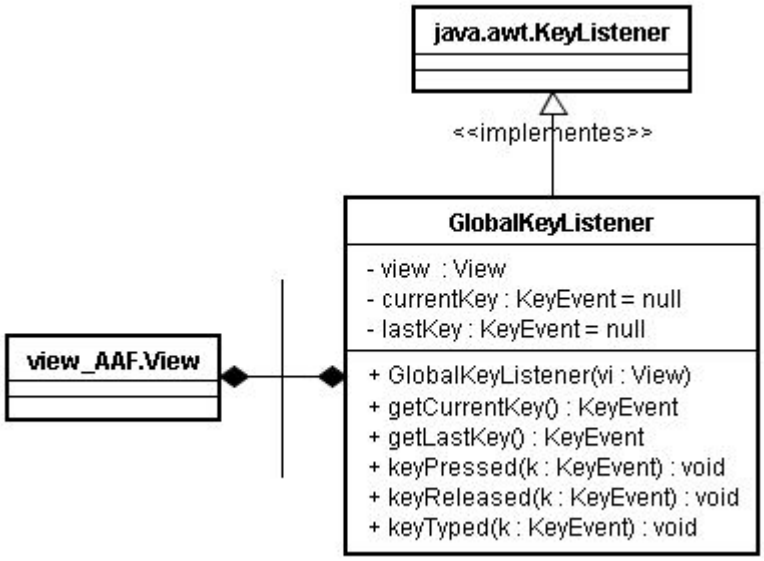
→ Erzeugt das Bedingungslabeling und startet damit die Funktionen **findGrounded (Labeling condition) / findPreferred (Labeling condition) / findStable (Labeling condition).** Die gefundenen Labelings werden dann in *candidatesSet* gespeichert.

Am Ende der Methode werden mithilfe von **updateText ( )** und **updateView ( )** Veränderungen bearbeitet und der View auf den neuesten Stand gebracht. Dabei werden alle Labelings des *candidatesSet* in *candidatesVec* kopiert und das mit *labelIndex* in *candidatesVec* gefundene Labeling auf dem *panel* des *view* als *active* festgelegt und die Textausgabe für die *console* des *view* aufbereitet. Wurde kein Labeling gefunden, verlieren alle Argumente ihr (in der Verarbeitung erhaltenes) Label.

12 Siehe Kapitel 4.2.1.2

13 Siehe Kapitel 4.1.2.5

4.1.3.2 GlobalKeyListener.java



Um Tastatureingabe aufnehmen zu können, wird ein GlobalkeyListener zugewiesen, der das Interface java.awt.KeyListener implementiert.
Dieser nimmt Tastaturbefehle wahr und sendet diese Informationen mithilfe der Methoden **keyPressed (KeyEvent e)**, **keyTyped (KeyEvent e)** und **keyReleased (KeyEvent e)** an die Methode **notice (KeyEvent e)** des *view*, der diese dann weiterverarbeiten kann. Zusätzlich geben die Attribute *currentKey* und *lastKey* beziehungsweise deren get ( ) – Methoden die aktuell gedrückte und die zuletzt gedrückte Taste an
.

4.1.3.3 PanelController.java

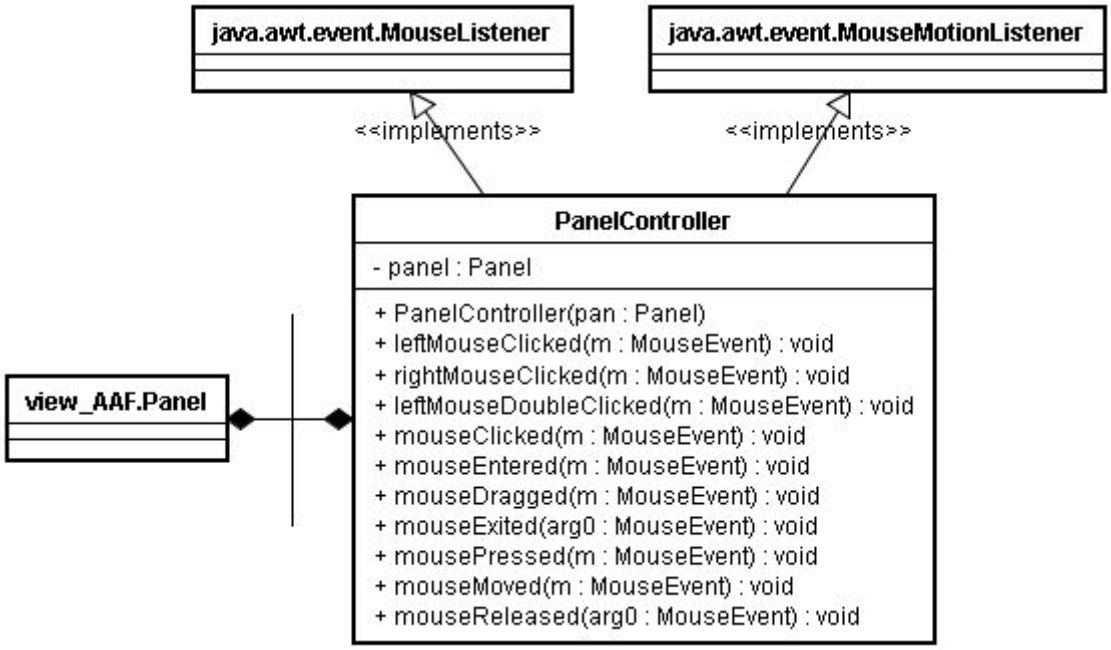


Fig. 4.14: Überblick PanelController.java

Um für das Panel eine Steuerung per Maus zu ermöglichen, wurde diesem ein PanelController, welcher die Interfaces java.awt.event.MouseListener und java.awt.event.MouseMotionListener implementiert, zur Seite gestellt. Diese Interfaces ermöglichen eben jene Steuerung mithilfe der zu implementierenden Methoden, die aus dem Diagramm ersichtlich sind. Einige Methoden übernehmen dabei keine Aufgabe, weshalb ich diese nicht erwähnen werde.
Ein Mausklick aktiviert die Methode **mouseClicked (MouseEvent m)**, welche die gedrückte Taste erkennt und je nach Taste die Methoden **rightMouseClicked (MouseEvent m)** oder **leftMouseClicked (MouseEvent m)** aufruft. Bei Doppelklick mit der linken Maustaste wird **leftMouseDoubleClicked (MouseEvent m)** aufgerufen.
rightMouseClicked (MouseEvent m) iteriert bei Aufruf über die Argumente der aktivierten Punkte und attackiert mit diesen das Argument des angeklickten Punktes, wenn es vom jeweiligen Argument nicht attackiert wird. Wenn diese Attacke bereits besteht, wird sie rückgängig gemacht. Wird dabei auf eine leere Fläche des Panels gedrückt und kein Zielpunkt gefunden, passiert nichts.
leftMouseDoubleClicked (MouseEvent m) versucht mit dem an der Position des Mausklicks gefundenen Punkt einen PointDialog zu öffnen, indem anschließend dem Argument des Punktes ein Name gegeben werden kann.
Wird an der angeklickten Position kein Punkt gefunden, wird ein neuer Punkt mit einem neuen Argument an dieser Position erzeugt.
leftMouseClicked (MouseEvent m) versucht den an der Stelle des Mausklicks ermittelten Punkt zu aktivieren. Dabei wird *collect* der Klasse Panel überprüft und bei Auswertung zu true der angeklickte Punkt zu *grabbed* hinzugefügt, bei Auswertung zu false wird *grabbed* geleert und anschließend der angeklickte Punkt hinzugefügt.
Wird an der angeklickten Position kein Punkt gefunden, wird wieder ein neuer Punkt

mit einem neuen Argument an dieser Position erzeugt.

Für das Bewegen der Punkte auf dem Panel sind die Methoden **mousePressed (MouseEvent m), mouseDragged (MouseEvent m)** und die oben erwähnten *distanceX* und *distanceY* der Point – Klasse wichtig. **mousePressed (MouseEvent m)** berechnet die Distanz der Punkte zu der Stelle, an der die Maus mit gedrückter Maustaste bewegt wird und teilt diese Distanz den Punkten mit. Parallel updatet die Methode **mouseDragged (MouseEvent m)** die Position der Punkte im Verhältnis von dessen Werten *distanceY* und *distanceX*. Dieses Vorgehen erwies sich als nötig, da die Punkte immer versetzt zur Position der Maus gezeichnet werden. Würden sie an der Position der Maus gezeichnet, würden sie immer rechts unterhalb der Maus „hängen“, da JAVA Kreise nicht an deren Mittelpunkt, sondern an deren oberen linken Ecke ausrichtet.
Gerät ein Punkt dabei an eine Position, die außerhalb des Panels liegt, und ist die entsprechende Option aktiviert, wird dieser aus dem Panel entfernt und dessen Argument aus dem Framework gelöscht.
Während **mouseDragged (MouseEvent m)** läuft, werden durchgehend **repaint ( )** – Aufrufe auf das Panel ausgeführt, sodass dieses immer aktuell bleibt. Sonstige Updates des Panels werden mit der Methode **mouseReleased (MouseEvent m)** aufgerufen, die einfach nach jedem Loslassen der Maus, welches nach jedem Mausklick gefeuert wird, einen **repaint ( )** aufruft.

Diese eigentlich sehr kompliziert erscheinenden Berechnungen und Aufrufe – insbesondere der ständige **repaint ( )** während des **mouseDragged (MouseEvent m)** – werden dabei erfreulich schnell durchgeführt, sodass eine komfortable und schnelle Steuerung auch bei vielen Argumenten und Attacken gegeben ist.

4.1.3.4 GeneratorDialogController.java

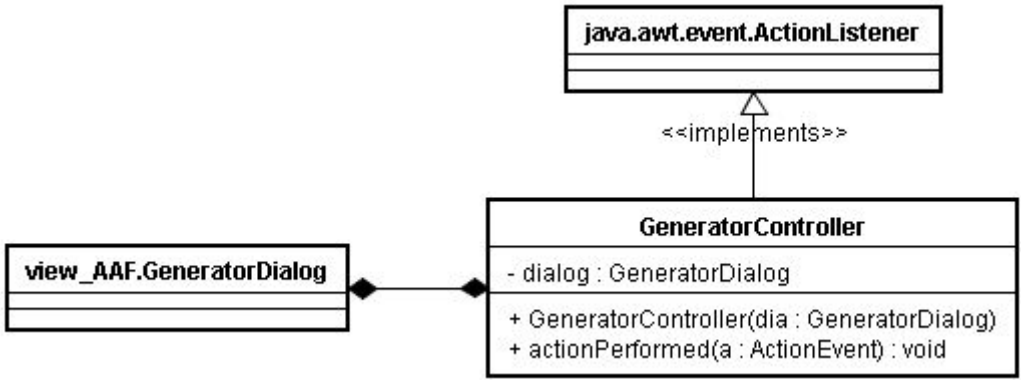


Fig. 4.15: Überblick GeneratorController.java

Dem GeneratorDialog wurde ein ActionListener in Form der Klasse GeneratorController zur Seite gestellt. Die Aufgabe dieses Controllers besteht lediglich daraus, die Methode **actionPerformed (ActionEvent a)** bei Druck auf einen der Buttons des übergebenen GeneratorDialoges *dialog* auszuführen. Dabei wird wieder der übergebene Befehl aus dem ActionEvent extrahiert und entweder – bei GeneratorDialog.OK – die Methode **submit ( )** des *dialog* aufgerufen oder – bei

GeneratorDialog.CANCEL – dieser *dialog* zerstört.

4.1.3.5 PointDialogController.java

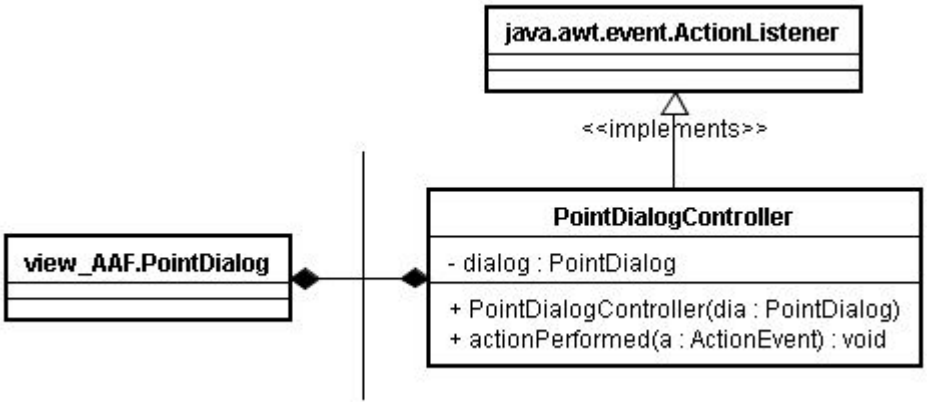


Fig. 4.16: Überblick PointDialogcontroller.java

Dem PointDialog wird ebenfalls ein Listener in Form des PointDialogController zur Seite gestellt.
Dieser ist identisch mit dem in 4.1.3.4 beschriebenen GeneratorDialogController, nur mit dem Unterschied, dass diesem ein PointDialog als *dialog* übergeben wird.
Diese Redundanz hätte sich per Vererbung vermeiden lassen können, indem man einen übergeordneten DialogController für diese beiden Klassen implementiert, allerdings wollte ich die einzelnen Controller als separate Klassen implementieren, um diese unabhängig voneinander betrachten zu können und eine unterschiedliche Nutzung der beiden Dialoge offen zu halten.

4.2 Sonstige Klassen

Abgesehen von den Klassen des MVC – Modells wurden 2 statische Klassen innerhalb des Paketes main_AAF implementiert, die ich nun kurz erläutern werde.

4.2.1 Main (main_AAF)

4.2.1.1 Main.java

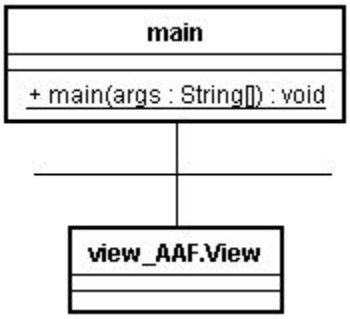


Fig. 4.17: Überblick main.java

JAVA benötigt zur Ausführung eines Programms einen Einstiegspunkt in Form einer statischen Methode namens **public static void main(String[] args)**. Über das übergebene String – Array *args* können dabei Startparameter per Konsole an das Programm übergeben und zum Programmstart der Programmlogik bekannt gemacht werden. Da dieses Programm komplett über eine GUI gesteuert wird, gestaltet sich die **main ( )** – Methode hier sehr kurz und instantiiert nur einen neuen View, der dann alles weitere übernimmt.

4.2.1.2 Options.java

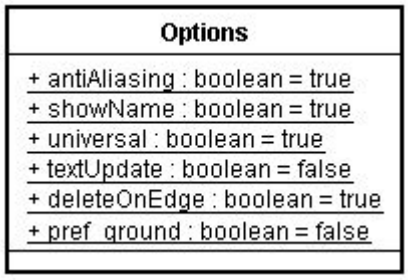


Fig. 4.18: Options.java

Diese Klasse stellt einige statische Boolean – Variablen zur Aktivierung und Deaktivierung bestimmter Optionen bereit. Dieses Vorgehen ermöglicht es, die festgelegten Optionen jederzeit jeder Klasse zugänglich zu halten, ohne diese erst (zum Beispiel als Objekt) übergeben zu müssen. Das Auslesen und Bearbeiten der Optionen ist dadurch von jeder Klasse aus möglich. Die Standardwerte dieser Variablen werden in der Options – Klasse bei Initialisieren der Variablen festgelegt und werden vom ViewController bei Änderungen im Options – Menü entsprechend geändert. Um die Option 'Argumentnamen anzeigen' zu deaktivieren, genügt es also ein Options.*showName = false* an beliebiger Stelle des Quellcodes zu setzen
Das Auslesen funktioniert dann unabhängig von der Klasse mithilfe von Options.optionsname und mittels einer if – Anfrage lässt sich dann beliebiger

Programmcode an eine Options – Variable binden.
So wurde zum Beispiel das Zeichnen des Strings, der den Namen eines Argumentes innerhalb der Methode **paintPoint (Graphics2D g, Point p)** mit

```
if ( Options.showName )
{
        //Zeichne den Namen auf das Panel
}
```

auf einfache Weise mit der zugehörigen Option verbunden.
Auf analoge Art und Weise sind die anderen Optionen mit den jeweiligen Stellen im Quellcode verbunden.

Diese Auflistung zeigt, welche Attribute mit welchen Optionen verbunden sind:

antiAliasing → Anti – Aliasing aktivieren
showName → Argumentnamen anzeigen
universal → Allgemeingültiges Labeling berechnen
textUpdate → Konsole bei Änderungen löschen
deleteOnEdge → Argumente per Ziehen löschen
pref_ground → Grounded Labeling für Preferred und Stable nutzen

Um eine neue Option hinzuzufügen, genügt es, diese als statisches Attribut in der Options – Klasse zu deklarieren, eine Möglichkeit der Beeinflussung durch den Nutzer bereitzustellen (zum Beispiel über View und ViewController) und die jeweilige Stelle im Quellcode von diesem Attribut abhängig zu machen.

5 Ausblick und Zusammenfassung

Abstrakt Argumentations – Frameworks ließen sich mithilfe der zur Verfügung gestellten Algorithmen zuverlässig und strukturiert implementieren. Die Möglichkeiten, welche die Objektorientierung bietet, ließen dabei eine Implementierung zu, welche dieses Modell unabhängig von seiner Umgebung gestaltet und gleichzeitig die Zusammenarbeit dieses Modells mit dieser Umgebung ermöglicht. Daher eignet sich das Programm als Möglichkeit der Visualisierung dieser Frameworks und liefert gleichzeitig ein unabhängiges Paket, mit dessen Hilfe sich die zugrunde liegenden Berechnungen zuverlässig durchführen lassen.
Bei der Arbeit mit Frameworks zeigten sich einige charakteristische Eigenschaften für die einzelnen Semantiken. Grounded Labelings lassen sich schnell berechnen, geben dabei aber bei einem ungünstig aufgebauten Framework einem großen Anteil an Argumenten das Label UNDEC. Diese gegebenenfalls geringe Aussagekraft wird von den Semantiken Preferred und Stable umgangen, indem hier die Möglichkeit besteht, mehrere Labelings als Lösung zu liefern. Diese Labelings variieren (wenn nicht UNDEC) dann in den Argumenten, die bei Grounded das Label UNDEC erhalten haben. Allerdings gestaltet sich die Berechnung dieser Labelings als sehr aufwändig, sodass bei ungünstigen Frameworks die Berechnung nicht sinnvoll werden kann.
Ein Framework mit 8 Argumenten und 64 Attacken (Jedes Argument attackiert jedes Argument einschließlich sich selbst) benötigt für die Berechnung der Preferred Semantik 109 601 Rekursionsschritte (2 284 Millisekunden). Bei 9 Argumenten und 81 Attacken sind es schon 986 410 Rekursionsschritte (22 703 Millisekunden). Ein Framework mit 10 Argumenten und 100 Attacken liefert erst nach 253 585 Millisekunden und 9 864 101 Rekursionsschritten das Preferred Labeling.
Die Anzahl der Millisekunden ist dabei Hardwareabhängig und variiert je nach sonstiger Ausnutzung der Hardware ein wenig, die Anzahl der Rekursionschritte kann aber als repräsentativ angesehen werden, da diese unabhängig von der Leistungsfähigkeit der Hardware gleich bleibt.
Bei jedem dieser Frameworks dauerte die Berechnung des (in diesem Fall identischen) Grounded Labelings nur 1 Rekursionsschritt. Stable Labelings existieren nicht, da alle Argumente UNDEC wurden und die Stable Semantik keine Argumente erlaubt, die UNDEC sind.

Folgende Tabelle stellt die Vor – und Nachteile der einzelnen Semantiken gegenüber:

	positiv	**negativ**
Grounded	Immer vorhanden, schnell	Nur 1 Labeling, deshalb unter Umständen geringe Aussagekraft
Preferred	Immer vorhanden, Mehrere Labelings als Ergebnis möglich	Langsam
Stable	Keine UNDEC – Labels, höchste Aussagekraft	Nicht immer vorhanden, langsam (dabei schneller als Preferred)

Ein Ansatz den Aufwand der Berechnungen zu reduzieren – das sogenannte Splitting – besteht darin, die Frameworks in sinnvolle Komponenten aufzuteilen, deren Label getrennt zu berechnen und dann die Label der Komponenten für die weitere Berechnung zu verwenden. Diese Möglichkeit lässt sich im vorliegenden Programm zwar aktivieren, die Implementierung steht aber noch aus und ist Teil einer anderen Arbeit.

Literaturverzeichnis

1. Baroni, P., and Giacomin, M.(2009), „Semantics of Abstract Argument Systems“ in *Argumentation in Artificial Intelligence* eds. I. Rahwan and G. Sinari, Berlin:Springer, pp. 25-44.

2. Modgil, S., and Caminada, M. (2009), „Proof Theories and Algorithms for Abstract Argumentation Frameworks“ in *Argumentation in Artificial Intelligence* eds. I.Rahwan and G. Sinari, Berlin:Springer, pp. 105-129.

3. Dung, P. (1995), „On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games.“ *Artificial Intelligence*, 77(2) 321-357.

4. JAVA API http://download.oracle.com/javase/1.4.2/docs/api/

5. Ullenboom, C. (2009), „Java ist auch eine Insel“ 8.Auflage, html – Version http://openbook.galileocomputing.de/javainsel8/

Ich versichere, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Ort Datum Unterschrift