HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

IAGen

Invariants-Assertion-Generation

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Frühjahrssemester 2012

Autor(en):	Frederick Egli, Dominik Mengelt
Betreuer:	Thomas Letsch
Experte:	Prof. Dr. Martin Zimmermann
Gegenleser:	Prof. Dr. Andreas Müller

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

Erklärung

Wir, Frederick Egli und Dominik Mengelt, erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Rapperswil, 11.06.2012

Frederick Egli

Dominik Mengelt

IAGen
Invariants-Assertion-Generation

Abstract

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	23.05.2012	f1egli	Erste Version des Abstract
1.0rc02	23.05.2012	dmengelt	Enterprise Architect Add-In Teil angepasst
1.0	24.05.2012	dmengelt, f1egli	Anpassungen gemäss Wochenbesprechung

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Abstract

Invariants-Assertion-Generation, kurz IAGen, ermöglicht es, Invarianten während der Laufzeit einer Java-Applikation zu überprüfen und bei einer Invariantverletzung entsprechend zu reagieren. Die Invarianten können auf verschiedene Arten definiert werden:

- Ein Java-Entwickler kann die Invarianten mit IAGen-Annotationen direkt in seinem Sourcecode definieren. Wenn die Applikation danach gestartet wird, werden die gesetzten Invarianten zur Laufzeit überprüft.
- Ein Entwickler kann im CASE Tool Enterprise Architect (nachfolgend EA) seine Applikation modellieren. Wenn der EA-Entwickler Modelle mit dem Sourcecode synchronisiert (Forward Engineering), sorgt das IAGen EA Add-In dafür, dass allfällig Invariantendefinitionen in den Sourcecode mittels IAGen-Annotationen gesetzt werden. Auch das Reverse Engineering von Sourcecode, welcher bereits IAGen-Annotationen gesetzt hat, wird vom IAGen EA Add-In unterstützt.
- Ein Entwickler kann im EA auch Java Bytecode importieren und das Modell mit IAGen-Invariantendefinitionen erweitern. Das IAGen EA Add-in bietet die Möglichkeit an, Invariantendefinitionen als XML-Datei zu exportieren, da der EA bestehenden Bytecode nicht verändern kann.

IAGen bietet eine Vielzahl von unterschiedlichen Annotationen an, um Invarianten zu definieren. Es lassen sich Wertebereiche, Multiplizitäten, Subsets (Teilmenge eines Containers), NotNull sowie Const-Invarianten (konstante Objekte) definieren. Ausserdem kann der Entwickler entscheiden, was bei einer Invariantenverletzung geschehen soll. So kann individuell festgelegt werden, ob eine Exception (Typ und Text der Exception kann ebenfalls gesetzt werden) geworfen werden soll, ob ein Swing Popup erscheinen soll, ob die Invariantenverletzungen geloggt werden sollen (Logfile kann gesetzt werden) und ob Debugmeldungen auf der Konsole erscheinen sollen. Diese Reaktionen können miteinander verknüpft werden.
Intern arbeitet IAGen mit AspectJ, welches aspekt-orientierte Programmierung in Java ermöglicht. Dadurch lassen sich generische Funktionalitäten über mehrere Klassen verwenden. Das Verweben dieser Funktionalitäten mit dem bestehenden Bytecode nennt sich Weaving. In IAGen wird AspectJ benötigt, um die Invarianten während der Laufzeit zu überprüfen und bei einer Invariantenverletzung entsprechend zu reagieren. Da AspectJ kein Runtimeweaving unterstützt, benutzt IAGen das Loadtimeweaving. Wird die annotierte Applikation gestartet, "weavt" sich IAGen in die Stellen der Applikation, wo die Invarianten überprüft werden sollen.
Das IAGen EA Add-In ist eine DLL, die mit C# programmiert ist und über einen Installer (.msi) installiert werden kann. Es wird über das Add-In-Menü im EA bedient und greift dabei direkt auf das EA Repository zu, um modellierte Änderungen in Form von Invarianten zu erkennen.

IAGen
Invariants-Assertion-Generation

Management Summary

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	28.05.2012	f1egli	Erste Version des Management Summary
1.0rc02	29.05.2012	f1egli	Version 1.0
1.0	04.06.2012	dmengelt	Enterprise Architect Add-In und Ausblick

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Management Summary

3.1 Ausgangslage

Modelliert ein Entwickler mit dem CASE Tool Enterprise Architect, können auch Invarianten definiert werden. Beispielsweise kann eine Multiplizität, welche aussagt, wie viele Objekte in einer Beziehung stehen können, auf einer Assoziation angegeben werden. Durch die Verwendung von Code-Synchronisierung gehen diese Informationen allerdings verloren, und eine Invariantenüberprüfung des Programmes zur Laufzeit findet nicht statt.

Ein Java-Entwickler kann Invariantenüberprüfungen programmatisch vornehmen. Dabei gibt es aber negative Effekte:

- Die Logik der Invariantenüberprüfung vergrössert den Code.
- Ändert eine Invariantendefinition, kann dies zu erheblichem Aufwand führen.
- Die Invariantenüberprüfung kann teilweise umgangen werden (öffentliche Felder).

Invariants-Assertion-Generation, kurz IAGen, soll es ermöglichen, Invarianten während der Laufzeit einer Java-Applikation zu überprüfen und bei einer Invariantverletzung entsprechend zu reagieren. IAGen soll ausserdem bei Code-Synchronisierung aus und in das Modellierungsprogramm Enterprise Architect helfen, damit keine Invariantendefinitionen verloren gehen.

3.2 Vorgehen

Das Vorgehensmodell während des Projekts basiert auf dem Unified Process von Craic Larman [Lar00]. Durch die Erstellung von Anforderungsspezifikationen wussten wir, was der Kunde will und wo die Rahmenbedingungen des Projektes liegen. Es wurde früh mit der Entwicklung eines Prototyps begonnen, da im Vorhinein nicht bekannt war, ob das Projekt im gewünschten Stil realisierbar ist. Dank der Dokumentation von Architektur und Design hatte man eine gute Diskussionsgrundlage und war immer im Bild, wie IAGen aufgebaut wurde.

3.3 Technologien

Durch die Aufgabenstellung, die Anforderungsspezifikation und den Prototyp wurden unter anderem diese Technologien vorgegeben bzw. ausgearbeitet:

- Java und Java-Annotationen [ann]
  Wurde von der Aufgabenstellung verlangt.
- Enterprise Architect [ent]
  Wurde von der Aufgabenstellung verlangt.

- AspectJ [Asp]
  Konnte durch den Prototyp als richtige Technologiewahl bestätigt werden.
- C# [add]
  Wurde als Technologie gewählt, um Enterprise Architect Add-Ins zu schreiben.

3.4 Ergebnisse

Mit IAGen können Invariantendefinitionen sowohl im Enterprise Architect wie auch im Java Code gesetzt werden. Das Verhalten bei einem Invariantenbruch kann dabei individuell gesteuert werden. Die Benutzung von IAGen ist dabei intuitiv und kann schnell erlernt werden. Unter anderem gibt es folgende Teilergebnisse:

- Invarianten im Java Code
  IAGen bietet eine Vielzahl von Invarianten an:
  - Wertebereiche
    Mittels *@Min, @Max, @Range* können Wertebereiche für Felder definiert werden. Diese Annotationen können auch in Parametern von Methoden und Konstruktoren gesetzt werden.
  - Multiplicity [mul]
    Mittels *@Multiplicity* kann angegeben werden, was die Unter- bzw. Obergrenze einer Multiplizität ist.
  - Subsets [Bal05]
    Mit der Annotation *@Subsets* kann eine Teilmenge einer anderen Menge definiert werden.
  - NotNull
    Mittels *@NotNull* kann angegeben werden, dass ein Feld nicht *null* sein darf. Diese Annotation ist auch bei Parametern einer Methode oder eines Konstruktors möglich.
  - Const
    Mit *@Const* können Felder, Methoden sowie Parameter einer Methode oder eines Konstruktors als *const* definiert werden.
- Mögliche Reaktionen bei Invariantenbruch:
  - Logging in eine vom Entwickler angegebene Datei.
  - Anzeige eines PopUps mit StackTrace.
  - Anzeige einer vom Entwickler definierten Exception inkl. eigener Message.
- Annotation Processing in der Entwicklungsumgebung:
  - Warnung, wenn Annotation auf nicht unterstütztem Typ gesetzt wurde.
  - Warnung bei falschen Kombinationen von Annotationen.

Listing 1: Beispiel einer IAGen Multiplicity Annotation auf einem Feld

```
1 @Multiplicity(lower=1, upper=10, log=true, logFile="players.txt")
2 private ArrayList<Player> players;
```

- Enterprise Architect IAGen Add-In
  - Generierung der @Multiplicity Annotationen, wenn Assoziationspfeile modelliert wurden.
  - Generierung der @Subsets Annotationen, wenn Subsets auf einer Assoziation modelliert wurden.
  - Exportieren der modellierten IAGen-Invarianten als XML-Datei. Somit können die IAGen-Invarianten auch bestehendem Bytecode (bereits kompilierter Code) hinzugefügt werden.
  - Das Forward sowie das Reverse Engineering im Enterprise Architect wird unterstützt.

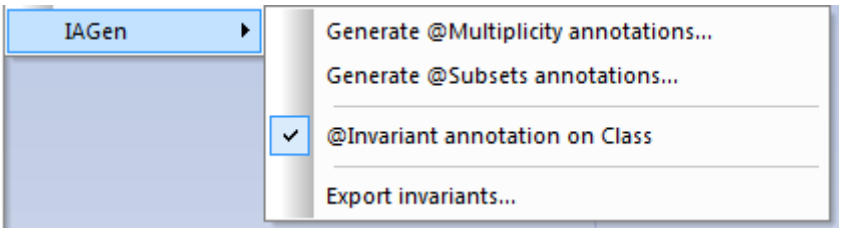


Abbildung 1: IAGen Enterprise Architect Add-In

3.5 Ausblick

Mithilfe von IAGen können sowohl bei bestehenden als auch bei neuen Java-Projekten Invarianten in Form von Annotationen hinzugefügt werden. Als Erweiterung wären weitere Annotationen wie *@NotEmpty* oder *@AssertTrue* bei booleans denkbar.

Literaturverzeichnis

[add] Enterprise Architect Add-in Framework. http://www.sparxsystems.com/uml_tool_guide/sdk_for_enterprise_architect/addins_2.htm.

[ann] Java Annotation. http://en.wikipedia.org/wiki/Java_annotation.

[Asp] AspectJ. Offizelle Aspectj Projekt Website. http://www.eclipse.org/aspectj/.

[Bal05] Prof. Dr. Heide Balzert. *Folienskript zur Dozenten-CD-ROM Lehrbuch der Objektmodellierung*. W3L-Verlag, 2005. Seite 20.

[ent] Enterprise Architect. http://www.sparxsystems.com/products/ea/index.html.

[Lar00] Craig Larman. *Applying UML and Patterns*. Prentice Hall, 2000. Seite 18.

[mul] Multiplizität. http://de.wikipedia.org/wiki/Multiplizit%C3%A4t_%28UML%29.

IAGen
Invariants-Assertion-Generation

Technischer Bericht

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	05.06.2012	dmengelt, f1egli	Gliederung und erste Version
1.0rc02	05.06.2012	dmengelt	Einleitung und Übersicht
1.0rc03	05.06.2012	f1egli	Schlussfolgerung
1.0rc04	05.06.2012	f1egli	Ergebnisse Java Sourcecode
1.0rc05	06.06.2012	dmengelt	Ergebnisse Add-In
1.0	06.06.2012	f1egli	Ergebnisse Bewertung der Ergebnisse
2.0rc01	07.06.2012	f1egli, dmengelt	Anpassung Einleitung
2.0	09.06.2012	f1egli, dmengelt	Finale Version 2.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Einleitung und Übersicht

3.1 Problemstellung

Im Modellierungsprogramm Enterprise Architect (EA) können Klassendiagramme erstellt werden. Diese Klassendiagramme beinhalten meist Invarianten in Form von Wertebereichen (Eine Person muss zwischen 18 und 65 Jahre alt sein), Multiplizitäten (Ein Parkhaus fasst maximal 500 Autos) oder Teilmengen (Ein Gewinner eines Turniers muss auch am Turnier teilgenommen haben). Ein Klassendiagramm im Enterprise Architect kann beispielsweise folgende Form haben:

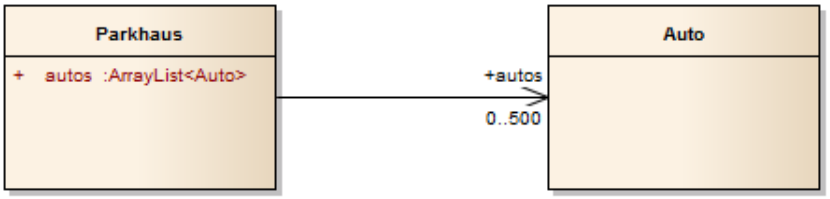


Abbildung 1: Modellierung einer Multiplizität im Enterprise Architect

Mit dem Enterprise Architect kann der Entwickler den Quelltext für die modellierten Klassen generieren lassen. Das Problem bei dieser Prozedur ist, dass die modellierten Invarianten verloren gehen. Es werden zwar die Klassen generiert, die Logik für die Zusicherung von Invarianten ist im Quelltext jedoch nicht vorhanden.

Arbeitet ein Entwickler ohne Modellierungsprogramm direkt mit dem Java-Quelltext, muss die Logik für die Zusicherung der Invarianten programmatisch vorgenommen werden. Einen Wertebereich für ein Integer-Datenfeld kann man nicht festlegen. Das gleiche gilt natürlich auch bei Multiplizitäten, Teilmengen und weiteren Invarianten.

3.2 Aufgabenstellung

Invariants-Assertion-Generation, kurz IAGen, soll es ermöglichen, dass modellierte Wertebereich-, Multiplizitäts- und Teilmengeninvarianten (sogenannte Subsets) beim Generieren des Quelltextes berücksichtigt werden. Weiter soll es auch dem Java-Entwickler ermöglicht werden, diese Invarianten mittels Annotationen [ann] direkt im Quelltext definieren zu können. Die gesetzten Annotationen sollen anschliessend zur Laufzeit von IAGen überprüft werden können. Folgende Grafik zeigt die mögliche spätere Verwendung von IAGen:

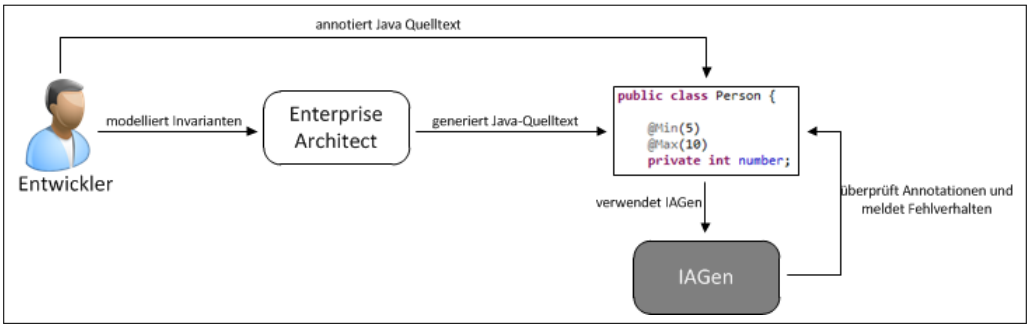


Abbildung 2: Mögliche Verwendung von IAGen

Im Kontext von IAGen existieren somit zwei Benutzergruppen:

- **Java-Entwickler**
  Definiert Invarianten mittels Annotationen direkt im Java-Quelltext.
- **Enterprise-Architect-Entwickler**
  Modelliert Invarianten auf Attributen, Klassen oder Assoziationen. Später sollen bei der Quelltext-Generierung diese Invarianten in den Programmcode einfliessen.

IAGen soll aus zwei Teilen bestehen. Zum einen aus einem Enterprise Architect Add-In (ermöglicht die Erweiterung der Funktionalitäten des Enterprise Architect) und zum anderen aus einer Programmbibliothek.

- **IAGen Enterprise Architect Add-In [add]**
  Das Add-In soll dafür sorgen, dass modellierte Invarianten während der Generierung des Quelltextes nicht verloren gehen.
- **IAGen Programmbibliothek**
  Die IAGen-Programmbibliothek soll die im Quelltext vorhandenen Invariantendefinitionen zur Laufzeit überprüfen und dem Entwickler gegebenenfalls das Fehlverhalten aufzeigen.

4 Ergebnisse

4.1 Enterprise Architect Add-In

Mit dem IAGen Enterprise Architect Add-In können vor der Generierung des Source-Codes (Forward Engineering) modellierte Multiplizitäts- und Subsetsinvarianten in die Tagged Values [tag] der jeweiligen Felder gespeichert werden. Zudem bietet es die Möglichkeit, modelllierte IAGen-Invarianten zu exportieren. Dies ist dann nützlich, wenn Invarianten auf bestehenden Bytecode gesetzt werden müssen.

4.1.1 Tagged Values: Wo werden Annotationen gespeichert?

Wird Java Sourcecode mit Annotationen (beispielsweise *@Override* bei einer Methode) im Enterprise Architect importiert (Reverse Engineering), werden diese in den sogenannten Tagged Values gespeichert:

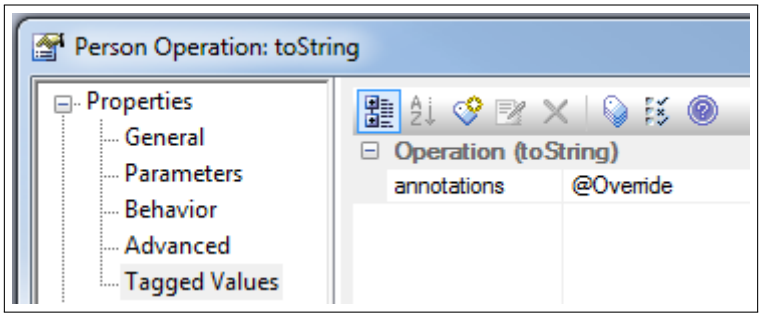


Abbildung 3: Tagged Value der Methode *toString()*

Tagged Values existieren unter anderem für Klassen, Methoden oder Felder. Die Tagged Value mit dem Namen “annotations“ wird bei der Generierung des Sourcecodes ausgelesen und die Annotation wird entsprechend in den Code geschrieben. Das IAGen Add-In generiert die Multiplizitäts- und Subsetsinvarianten als Annotationen in diese Tagged Values.

4.1.2 aop.xml: Wie können Invarianten auf Bytecode gesetzt werden?

Importiert ein Entwickler bereits kompilierte Java-Dateien (.class) in den Enterprise Architect (Bytecode Reverse Engineering), müssen die modellierten IAGen-Invariantendefinitionen exportiert werden können. Dies ist notwendig, da der Enterprise Architect bestehenden Bytecode nicht verändern kann. Das IAGen Add-In generiert beim Exportieren dieser Änderungen eine aop.xml-Datei [ltw], welche die Invariantendefinitionen beinhaltet. Die im aop.xml enthaltenen Invarianten werden dann während des AspectJ (Kapitel 4.2.1) Load Time Weaving Prozesses dem bestehenden Java Byte Code hinzugefügt.

4.2 Java-Sourcecode-Annotierungen

Invarianten können mit IAGen direkt in dem Sourcecode über Annotationen definiert werden. Es gibt folgende IAGen-Annotationen:

- *@Min* - Untergrenze, @Max - Obergrenze und @Range - Wertebereich
  Zulässige Typen: primitive Datentypen (ausser boolean), deren Wrapperklassen und String.
  Ort: Darf auf Felder, Konstruktorparameter und Methodenparameter gesetzt werden.

- *@Multiplicity* - Multiplizitätsinvarianten
  Zulässige Typen: Collections, Maps und Arrays [ ].
  Ort: Darf auf Felder gesetzt werden.

- *@Subsets* - Teilmengeninvariante
  Zulässige Typen: Objekte (welche in der grossen Menge gespeichert werden), Collections, Maps und Arrays [ ].
  Ort: Darf auf Felder gesetzt werden.

- *@NotNull* - Objekt darf nicht *null* sein.
  Zulässige Typen: Objekte.
  Ort: Darf auf Felder, Konstruktorparameter und Methodenparameter gesetzt werden.

- *@Const* - Objekt darf nicht verändert werden.
  Zulässige Typen: Objekte.
  Ort: Darf auf Methoden, Felder, Konstruktorparameter und Methodenparameter gesetzt werden.

Für alle IAGen-Annotationen (ausser *@Const*) kann angegeben werden, welche Reaktion bei einer Invariantenverletzung geschehen soll. Diese sind mit Standardwerten vordefiniert, können aber mit eigenen Werten überschrieben werden:

- Exception
  Einstellungsmöglickeiten:
  - Soll eine Exception geworfen werden?
  - Welche Exception soll geworfen werden?
  - Welche Meldung soll in der Exception vorhanden sein?

- PopUp
  Wenn eingeschaltet, erscheint ein Swing Popup, sobald eine Invariantenverletzung vorliegt. Der Stacktrace kann dann optional angezeigt werden.

- Logging
  Einstellungsmöglickeiten:
  - Soll ein Log geführt werden?
  - In welches Datei soll geloggt werden?

- Debug
  Wenn eingeschaltet, erscheint eine Debugmeldung auf der Konsole, sobald eine Invariante geprüft wird (also nicht nur bei einer Invariantenverletzung).

4.2.1 AspectJ: Bestehenden Code mit Invariantenüberprüfungen ergänzen

Laut Aufgabenstellung müssen die Invariantendefinitionen im Sourcecode mittels Annotationen gesetzt werden. Das Problem entstand, wie man zusätzliche Logik für die Überprüfung der Invarianten in den bestehenden Sourcecode hinzufügen kann.
Zur Problemlösung setzt IAGen AspectJ [Aspc] als Technologie ein.

4.2.2 Pointcuts: Wo werden IAGen Invarianten überprüft?

IAGen verwendet drei verschiedene Pointcuts [poi], um die Invariantenüberprüfungen durchzuführen:

- Nach einem externen Konstruktoraufruf.
- Nach einem externen Aufruf einer Methode.
- Bei Setzen eines externen Feldes.

Folgendes Beispiel soll dies verdeutlichen:

Listing 1: Externer Aufrufer

```
1  public class Main {
2
3      public static void main(String[] args) {
4          Auto a = new Auto(300);
5          a.setPs(150);
6          a.ps = 150;
7      }
8  }
```

Listing 2: Annotierte Klasse

```
1  @Invariant
2  public class Auto {
3      @Max("1000") public int ps;
4      //Überprüfe 'ps'. Von Listing 1; Zeile 6 ausgelöst.
5
6      public Auto(int ps) {
7          this.ps = ps;
8          //Überprüfe 'ps'. Von Listing 1; Zeile 4 ausgelöst.
9      }
10
11     public void setPs(int ps) {
12         this.ps = ps;
13         //Überprüfe 'ps'. Von Listing 1; Zeile 5 ausgelöst.
14     }
15 }
```

Bei Parameter Annotationen findet die Invariantenüberprüfung beim Eintritt in eine Methode oder beim Aufruf eines Konstruktors statt (Pre-Condition).

4.2.3 Command Pattern: Wie werden Invariantendefinitionen überprüft?

Sobald einer der drei Pointcuts den Advice [Aspb] aufruft, iteriert dieser über alle Felder der annotierten Klasse. Für jede Annotation auf dem Feld wird eine *execute()*-Methode auf dem richtigen Annotation Handler aufgerufen. Diese Logik wurde nach dem Command Pattern [com] implementiert. Im richtigen Annotation Handler muss für eine Invariantenüberprüfung der Typ des annotierten Feldes herausgefunden werden. Danach ruft der entsprechende Annotation Handler eine *executeInvariantAssertion()*-Methode auf dem richtigen Type Handler auf. Diese Logik wurde ebenfalls mit dem Command Pattern implementiert. Im richtigen Type Handler angelangt, kann entschieden werden, ob die Invariante verletzt wurde oder nicht.

5 Schlussfolgerungen

Mit IAGen ist es möglich, innert kürzester Einarbeitungszeit Invarianten im Enterprise Architect zu modellieren, sodass sie bei der Codegenerierung als IAGen-Annotationen im Sourcecode wiederzufinden sind. Neue wie auch bestehende Annotationen können im Code einfach erstellt bzw. ergänzt oder abgeändert werden.
Folgendes Beispiel zeigt einen typischen Ablauf. Die Klasse Parkhaus hat eine ArrayList, um Autos zu speichern:

Listing 3: Parkhaus und Auto ohne Invarianten

```
1  public class Auto {}
2
3  public class Parkhaus {
4
5      private ArrayList<Auto> autos;
6
7      public Parkhaus() {
8          autos = new ArrayList<Auto>();
9      }
10
11     public void fuelleMit(Auto a){
12         autos.add(a);
13     }
14 }
```

Man weiss, dass ein Parkhaus zwischen 0 und 500 Autos aufnehmen kann. Dies entspricht einer Multiplizitätsinvariante! Nachdem der Code im Enterprise Architect importiert worden ist, modelliert man die Invarianten:

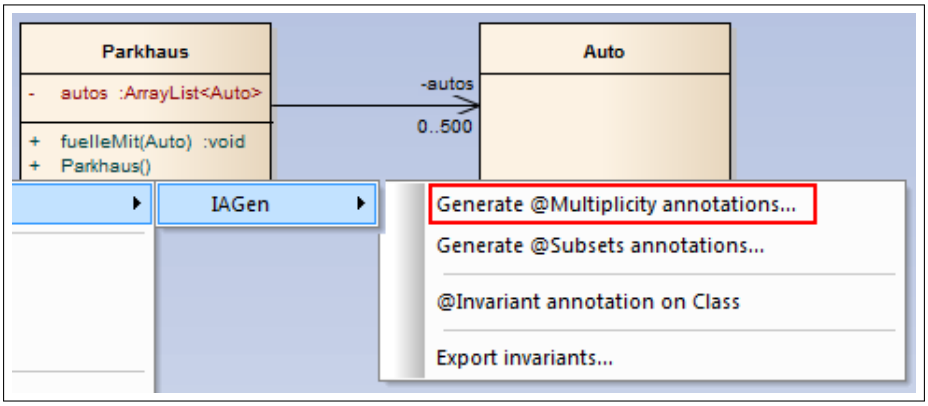


Abbildung 4: Generierung der *@Multiplicity*-Invariante mit dem IAGen Add-In

Der synchronisierte Code hat danach die IAGen-Annotationen gesetzt:

Listing 4: Parkhaus mit IAGen Invarianten

```
1  @Invariant
2  public class Parkhaus {
3
4      @Multiplicity(lower=0, upper=500)
5      private ArrayList<Auto> autos;
6
7      public Parkhaus() {
8          autos = new ArrayList<Auto>();
9      }
10
11     public void fuelleMit(Auto a){
12         autos.add(a);
13     }
14 }
```

Ab sofort werden die Invarianten von IAGen zur Laufzeit überprüft. Da die Invariantendefinition die Standardoptionen belassen hat, wird bei einer allfälligen Invariantenverletzung eine Exception geworfen:

Listing 5: Exception bei einer verletzten Invariante

```
1 Exception in thread "main" ch.iagen.exception.InvariantBrokenException:
      @Multiplicty contract broken!
2     // ...
3     at to.test.Main.main(Main.java:11)
```

Die bestehende Multiplicity-Invariante kann sowohl im Enterprise Architect wie auch im bestehenden Code angepasst werden. Ein Java-Entwickler möchte nicht, dass eine Exception geworfen wird, sondern dass die Verletzungen in ein Logfile gespeichert werden. Er ändert die Invariantendefinition auf:

```
@Multiplicity(lower=0, upper=500, raiseException=false, log=true)
private ArrayList<Auto> autos;
```

Tritt eine Invariantenverletzung auf, wird sie in die Datei log.txt protokolliert:

```
Tue Jun 05 20:36:32 CEST 2012 The size of field (autos) is not between 0 and
500! Found on Line: Main.java:11
```

5.1 Bewertung der Ergebnisse IAGen Add-In

5.1.1 Tagged Values

Die Generierung der Zusicherung für Invarianten aus dem Enterprise Architect kann auf zwei Varianten gemacht werden:

1. **Über die Tagged Values**
   Modellierte IAGen Invarianten werden vor dem Forward Engineering in die Tagged Values der betroffenen Klasse oder dem betroffenen Feld gespeichert.

2. **Über das Code Template Framework [ctf]**
   Die Art, wie der Sourcecode generiert wird, kann über das Enterprise Architect Code Template Framework angepasst werden.

Bei der zweiten Variante müssten die internen Java Code Templates [ct] des Enterprise Architects so verändert werden, dass bei der Generierung die IAGen-Invarianten als Annotationen in den Sourcecode geschrieben werden. Die angepassten Templates könnten dann über XML-Dateien in die jeweilige Enterprise Architect Installation importiert werden. Dies hat jedoch den Nachteil, dass dann bei jedem Enterprise Architect Projekt das veränderte Java Code Template verwendet werden würde. Zudem bestünde das Problem, dass die Annotationen, welche ein Java-Entwickler bereits im Code gesetzt hat, bei der Generierung überschrieben werden. Aus diesen Gründen setzen wir beim IAGen Add-In die Tagged-Values-Variante ein, da das Standard Java Code Template bereits die Tagged Value mit dem Namen “annotations“ bei der Generierung berücksichtigt und somit die Synchronität mit dem Sourcecode gewährleistet ist.

5.1.2 aop.xml

Für das Format der XML-Datei, welche die vom Enterprise Architect exportierten IAGen-Invarianten beinhaltet, wurden zwei Optionen in Betracht gezogen:

1. **AspectJ aop.xml**
   Beinhaltet die Definitionen der Aspekte, welche die Invarianten als Annotationen deklarieren:

```
<aspectj>
  <aspects>
    <concrete-aspect name="ch.iagen.aspects.reverse.Annotater">
      <declare-annotation
      type="Parkhaus"
      annotation="@ch.iagen.annotations.Invariant"
     />
      <declare-annotation
      field="* Parkhaus.autos"
      annotation="@ch.iagen.annotations.Multiplicity(upper=500)"
     />
    </concrete-aspect>
  </aspects>
</aspectj>
```

2. **IAGen.xml**
   Ein eigenes Format für die IAGen-Invarianten:

```
<iagen>
    <invariant>
        <class>Parkhaus</class
    </invariant>
    <multiplicity>
        <field>
            <name>autos</name>
            <class>Parkhaus</class>
            <package></package>
            <upper>500</upper>
        </field>
    </multiplicity>
</iagen>
```

Die Option mit dem eigenen XML-Format wurde trotz besserer Leserlichkeit verworfen, da AspectJ das Parsing der aop.xml-Datei übernimmt. Ein eigenes XML-Format hätte zuerst geparsed werden müssen. Zudem müssten beim Programmstart sowohl die eigene XML-Datei mit den Invarianten-Definitionen als auch ein aop.xml mit den IAGen Aspects angegeben werden. Dies wollten wir vermeiden, damit die Konfiguration für den Entwickler einfach bleibt.

5.2 Bewertung der Ergebnisse IAGen Library

5.2.1 AspectJ

Zu Beginn des Projektes wurden unterschiedliche Möglichkeiten evaluiert, wie die Invariantenüberprüfung auf bestehenden Java Code gemacht werden kann:

- **AspectJ**
  Mit AspectJ wird die Logik für die IAGen Invariantenüberprüfung zur Ladezeit (Load Time Weaving) in den bestehenden Bytecode verwoben.
- **APT - Annotation Processing Tool [apt]**
  Erzeugt beim Kompilieren neue Java-Source-Dateien, welche die Logik für die IAGen-Invariantenüberprüfung beinhalten würden.
- **AST - Abstract Syntax Tree Transformierung [ast]**
  Modifizierung des Java Abstract Syntax Trees zur Kompilierzeit.

Da gemäss Aufgabenstellung auch das Hinzufügen von Invariantenüberprüfungen auf Bytecode geprüft werden musste, haben wird uns für AspectJ entschieden. Sowohl APT als auch AST würden die Überprüfungen beim Kompilieren in den Code generieren, während AspectJ mit Load Time Waving (zur Ladezeit) arbeitet. Somit können auch bestehendem Bytecode-Überprüfungen hinzugefügt werden. Zudem lassen sich Pointcut-signaturen von AspectJ ideal mit Annotationen verknüpfen. Beispielsweise kann herausgefunden werden, ob eine Methode einen Parameter enthält, der mit einer bestimmten Annotation versehen ist.

5.2.2 Pointcuts

Es gäbe die Möglickeit, einen Pointcut zu erstellen, der nur auf das Setzen (*set() Pointcut*) von annotierten Feldern anspricht. Vorteile wären dann, dass die Überprüfungslogik nur an wenigen Stellen im Code eingewoben werden müsste und dass die Übersichtlichkeit in den Aspects [aspa] steigt. Eine Klasse darf aber laut Definition [inv] die eigene Invariante temporär verletzen, weshalb diese Variante nicht als Problemlösung verwendet werden konnte.

Die Variante mit drei Pointcuts hat den gravierenden Nachteil, dass die Überprüfungslogik in viele Stellen des Programms eingewoben werden muss. Für jeden externen Methodenaufruf, externen Konstruktoraufruf und für jedes externe Setzen von Feldern müsste die IAGen-Logik eingewoben werden, und zwar unabhängig davon, ob im Ziel Invarianten definiert sind oder nicht.

Wir haben aus diesem Grund die *@Invariant* Annotation eingeführt. Sie dient lediglich zur Markierung von Klassen, damit die Pointcuts nur auf Ziele reagieren, welche auch Invarianten enthalten. Durch diese Massnahme konnte die Performance von IAGen gesteigert werden.

5.2.3 Command Pattern

Ohne das Command Pattern müsste im Advice entschieden werden, durch welche Annotation iteriert wird. Mit sieben Annotationen wären das folglich schon sieben *instanceof*-Anweisungen. Jede Annotation kann auf verschiedene Datentypen gesetzt werden. Die Wertebereichannotationen (*@Min*, *@Max* und *@Range*) können je auf 15 verschiedenen Typen gesetzt werden. Insgesamt hätte der Advice über 50 *instanceof*-Anweisungen, was nicht mehr übersichtlich genug wäre.

Durch den Einsatz des Command Pattern muss pro Aspect einmalig eine *handlers* Map angelegt werden. Das gleiche gilt für die Annotation Handlers. Als Beispiel erstellt die Oberklasse *BaseRangeAnnotationHandler*, welche für die Wertebereich Annotationen zuständig ist, folgende *handlers* Map:

Listing 6: Handlers für Typen der Wertebereich Annotationen

```
1   protected void createHandlers() {
2       handlers = new HashMap<Class<?>, RangeHandler>();
3       handlers.put(Byte.class, new ByteHandler(this));
4       handlers.put(Short.class, new ShortHandler(this));
5       handlers.put(Integer.class, new IntegerHandler(this));
6       handlers.put(Long.class, new LongHandler(this));
7       handlers.put(Float.class, new FloatHandler(this));
8       handlers.put(Double.class, new DoubleHandler(this));
9       handlers.put(String.class, new StringHandler(this));
10      handlers.put(Character.class, new CharacterHandler(this));
11      handlers.put(null, new NullHandler(this));
12  }
```

Durch das Command Pattern entfallen die *instanceof*-Anweisungen.

5.2.4 Internes Logging

Wenn man sich in die aspektorientierte Programmierung einliest, wird als Beispiel immer das Logging der eigenen Applikation erwähnt. Normalerweise verursacht die Einführung eines Loggers eine höhere Kopplung und redundanten Code, was sich mit AspectJ vermeiden lässt. Es wurde ein Aspect geschrieben, um das interne Logging sicherzustellen. Aus Performanzgründen ist der Aspekt per Standard deaktiviert, damit die Logging-Methoden nicht permanent mit den IAGen-Klassen verwoben sind. Bei Bedarf kann der Aspekt im IAGen Projekt aktiviert werden. Alle wichtigen Ereignisse werden dann in die Datei “IAGenLog.txt“ protokolliert.

5.3 Weiteres Vorgehen

Als Erweiterung wären weitere Annotationen wie *@NotEmpty* oder *@AssertTrue* bei booleans denkbar. Ausserdem könnte die Popup-Reaktion (siehe Kapitel 4.2) mit einer Asynchron- und Quittierungsoption erweitert werden. Das Enterprise Architect Add-In könnte für die Behandlung weiterer Invarianten erweitert werden.

Literaturverzeichnis

[add] Enterprise Architect Add-in Framework. http://www.sparxsystems.com/uml_tool_guide/sdk_for_enterprise_architect/addins_2.htm.

[ann] Java Annotation. http://en.wikipedia.org/wiki/Java_annotation.

[apt] Annotation processor. http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html.

[aspa] Aspects. http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html#aspects.

[Aspb] AspectJ. Advice semantics. http://www.eclipse.org/aspectj/doc/released/progguide/semantics-advice.html.

[Aspc] AspectJ. Offizelle Aspectj Projekt Website. http://www.eclipse.org/aspectj/.

[ast] Abstract syntax tree. http://en.wikipedia.org/wiki/Abstract_syntax_tree.

[com] Command pattern. http://en.wikipedia.org/wiki/Command_pattern.

[ct] Code Templates. http://www.sparxsystems.com/enterprise_architect_user_guide/software_development/codetemplatesoverview.html.

[ctf] Code Template Framework. http://en.wikipedia.org/wiki/Abstract_syntax_tree.

[inv] Class invariant. http://en.wikipedia.org/wiki/Class_invariant.

[ltw] AspectJ Load Time Weaving mit aop.xml. http://www.eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html#configuring-load-time-weaving-with-aopxml-files.

[poi] Pointcuts semantics. http://www.eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html.

[tag] Enterprise Architect Tagged Values. http://www.sparxsystems.com/uml_tool_guide/modeling_tool_features/thetaggedvaluestab.htm.

IAGen
Invariants-Assertion-Generation

Persönliche Berichte

13. Juni 2012

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	05.06.2012	dmengelt	Template erstellt
1.0rc02	12.06.2012	dmengelt	Dominik Mengelt
1.0rc03	12.06.2012	f1egli	Frederick Egli
1.0	13.06.2012	f1egli	Finale Version 1.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Persönliche Berichte

3.1 Frederick Egli

In der Aufgabenstellung war festgehalten, dass Annotationen als Invariantendefinition im Sourcecode zu verwenden sind. Da wir keinerlei Erfahrungen mit Annotationen hatten, mussten wir uns zuerst in diese Thematik einlesen. Der Knackpunkt der Arbeit ist nicht das Erstellen einer Annotation, sondern wie man zusätzliche Logik (in unserem Fall die Invariantenüberprüfung) in ein annotiertes Programm einbauen kann. Es gibt verschiedene Lösungen für dieses Problem. Praktisch alle Ansätze binden aber die Zusatzlogik beim Kompilierzeitpunkt ein. Das stellte aber ein Problem dar, da in der Aufgabenstellung die optionale Anforderung für Bytecode-Invariantendefinition aufgeführt war. Wir haben uns aus diesem Grund für AspectJ entschieden, da dort die zusätzliche Logik während der Ladezeit der Klassen eingewoben werden kann. Mit AspectJ hatten wir bereits Erfahrungen in der Studienarbeit sammeln können.

Für das Enterprise Architect Add-In haben wir .NET C# eingesetzt, um allfällige Invarianten beim modellieren zu erkennen und diese dann beim jeweiligen Feld als Tagged Value zu setzen. Grosse C#-Erfahrung hatten wir nicht und mussten uns daher in die Programmiersprache einarbeiten.

Wir mussten uns zudem auch in die Invariantenthematik einarbeiten. Insbesondere der Zeitpunkt der Invariantenüberprüfung war besonders wichtig, da es Ausnahmen gibt, bei der eine Invariante temporär verletzt werden darf.

Mit über 940 verschiedenen Tests für die IAGen Library konnten wir uns jederzeit sicher sein, wenn Design-Änderungen oder Refactorings vorgenommen wurden. Nach dem Prototyp haben wir für neue Features das sogennante Test-driven development (TDD) angewendet. Durch die Bachelorarbeit habe ich den extremen Nutzen von JUnit-Tests kennen und schätzen gelernt.

Besonders gefreut hat mich, dass durch unsere Anfrage an die AspectJ Mailing-Liste ein neues Feature implementiert wurde. Unser Test-Code für die Anfrage ist (momentan) sogar noch im offiziellen Developer Release von AspectJ vorhanden.

Die Zusammenarbeit mit Dominik Mengelt war, wie gewohnt, sehr gut. Da wir bereits in jedem Projekt in der HSR zusammen gearbeitet haben, kennen und ergänzen wir uns hervorragend. Mit unserem Betreuer, Thomas Letsch, war die Zusammenarbeit, wie bereits in der Studienarbeit, äusserst konstruktiv.

Ich würde IAGen jederzeit wieder als Bachelorarbeit wählen. Invarianten, Annotationen im Zusammenspiel mit AspectJ, intensives testing und C# als neue Programmiersprache haben diese Arbeit äusserst spannend, herausfordernd und lehrreich gemacht.

3.2 Dominik Mengelt

Wie bereits bei der Studienarbeit mussten zu Beginn der Bachelorarbeit zuerst unterschiedliche Technologien getestet werden. Ob die Arbeit gemäss Aufgabenstellung vollständig realisierbar ist, wussten wir nicht. Wir begannen deshalb sehr früh mit Prototyping. Unterschiedliche Ansätze wurden getestet, bis wir uns schlussendlich für AspectJ bei der Library und .NET C# beim Enterprise Architect Add-In entschieden haben. Zudem mussten wir uns in das Thema Invarianten einlesen. Die Frage: "Wann werden Invarianten in der Programmierung überprüft?", musste geklärt werden.

Bei der Entwicklung des Add-Ins stellte sich schnell heraus, dass der Enterprise Architect standardmässig praktisch keine Eingabenüberprüfung macht. Beispielsweise kann bei einer Multiplizität auf einer Assoziation auch ein nicht UML-konformer String angegeben werden. Trotzdem konnten wir die in den Anforderungen gewünschten Funktionen implementieren.

Die Herausforderung bei der Implementation der IAGen Library war die Überprüfung von Invarianten bei bestehendem Bytecode. Mit dem Einsatz von AspectJ Load Time Weaving konnten wir dieses Problem jedoch lösen. Zusätzlich bietet AspectJ bereits sehr gute Unterstützung, wenn es darum geht, auf Annotationen im Java Code zu reagieren. Eine tolle Erfahrung war es auch, dass nach einer von uns gestellten Anfrage auf die AspectJ Mailing-Liste innert kürzester Zeit ein neues Feature implementiert wurde, welches wir gut gebrauchen konnten.

Als Vorgehensmodell zur Entwicklung der Software haben wir den Unified Process eingesetzt. Parallel zur Entwicklung des Add-Ins und der Library haben wir iterativ die entsprechenden Dokumente erstellt respektive angepasst.

Während des Ganzen Projekts war die Zusammenarbeit mit Frederick Egli sehr angenehm. Wir konnten die Verantwortlichkeiten gut aufteilen und setzten meistens Pair Programming ein. So konnten wir auch gleich intensiv über Architektur- und Designentscheide diskutieren.

Die Bachelorarbeit zeigte mir erneut wie mächtig die aspektorientiere Programmierung ist. Es hat grossen Spass gemacht, sich mit Themen wie dem Enterprise Architect Add-In SDK oder dem Java Annotation Processing auseinander zu setzen. Zusätzlich konnte ich meine Kentnisse in AspectJ weiter vertiefen. Unser Betreuer Thomas Letsch liess uns bei der Implementierung grosse Freiheiten, was ich sehr schätzte.

IAGen - Invariants-Assertion-Generation

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

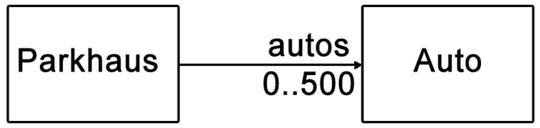
Bachelorarbeit Frühjahrssemester 2012
Themengebiet: Software


Frederick Egli


Dominik Mengelt

Betreuer: Thomas Letsch
Experte: Prof. Dr. Martin Zimmermann
Gegenleser: Prof. Dr. Andreas Müller

OHNE IAGEN

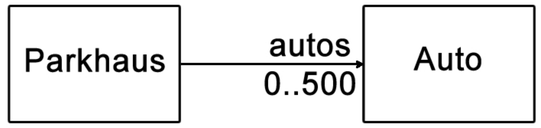


INVARIANTENDEFINITIONEN (IM BEISPIEL: 0..500) GEHEN BEI CODE-SYNCHRONISIERUNGEN VERLOREN.

```
public class Parkhaus {

  private List<Auto> autos;

  public Parkhaus() {
    autos = new ArrayList<Auto>();
  }

  public void fuelleMit(Auto a){
    if(autos.size() >= 500){
      throw new RuntimeException();
    }
    autos.add(a);
  }
}
```

INVARIANTENÜBERPRÜFUNG MUSS SELBER ERSTELLT WERDEN!

MIT IAGEN



INVARIANTENDEFINITIONEN WERDEN VON IAGEN ERKANNT, DAMIT DIESE BEI CODE-SYNCHRONISERIUNGEN ALS ANNOTATIONEN GESETZT WERDEN.

```
@Invariant
public class Parkhaus {

 @Multiplicity(upper=500)
 private List<Auto> autos;

 public Parkhaus() {
   autos = new ArrayList<Auto>();
 }

 public void fuelleMit(Auto a){
   autos.add(a);
 }
}
```

INVARIANTENÜBERPRÜFUNG WIRD VON IAGEN ZUR LAUFZEIT SICHERGESTELLT!

ENTERPRISE ARCHITECT

ENTERPRISE ARCHITECT

Java

Java

IAGEN ANNOTATIONEN

- @MIN, @MAX, @RANGE
- @MULTIPLICITY, @SUBSETS
- @NOTNULL, @CONST

WEITERE IAGEN FUNKTIONEN

- LOGGING, EXCEPTIONS & MESSAGES
- ANNOTATION PROCESSING
- NACHTRÄGLICHE ANNOTIERUNG VON BYTECODE!

IAGEN FAKTEN

- ASPEKTORIENTIERTE PROGRAMMIERUNG MIT ASPECTJ
- CUSTOMIZED ANNOTATION PROCESSOR
- C# ENTERPRISE ARCHITECT ADD-IN
- MEHR ALS 940 TESTS

IAGen
Invariants-Assertion-Generation

Vision

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	24.02.2012	f1egli	Erster Entwurf der Vision
1.0rc02	05.03.2012	f1egli	Vision erweitert
1.0rc03	05.03.2012	dmengelt	Entwickler Beschreibung
1.0rc04	12.03.2012	f1egli	Vision überarbeitet
1.0rc05	12.03.2012	dmengelt	Anpassung Entwickler Beschreibung
1.0rc06	14.03.2012	dmengelt	Überarbeitung & Umstellungen
1.0	15.03.2012	dmengelt	Verhalten bei Invarianten Verletzung
2.0	03.04.2012	dmengelt	Finale Version 2.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Einführung

Modelliert man im Enterprise Architect [ent] Klassendiagramme mit Multiplizitäten, Subsets oder Wertebereichen bei Datenfeldern, gehen diese Informationen beim Generieren von Sourcecode verloren. Diese Invarianten [inv] müssen vom Entwickler später im Sourcecode selber programmatisch überprüft werden.

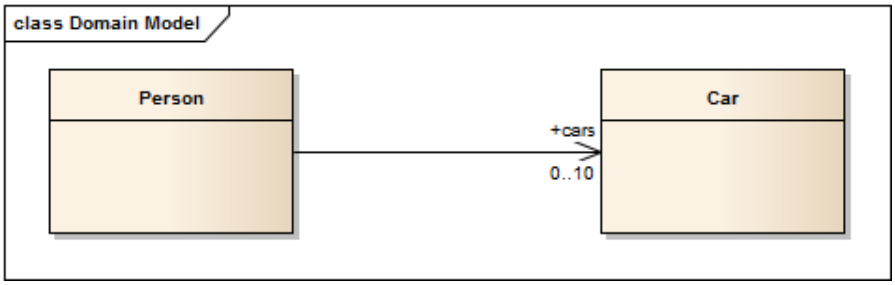


Abbildung 1: Modellierung einer Multiplizität im Enterprise Architect

IAGen soll es ermöglichen, das die modellierten Invarianten beim Generieren des Sourcecodes berücksichtigt werden. Weiter soll es auch dem Java-Entwickler ermöglicht werden, Invarianten mittels Annotationen direkt im Source Code zu definieren. Die definierten Annotationen sollen anschliessend zur Laufzeit von IAGen überprüft werden können.

Listing 1: Beispielannotation einer Multiplizität

```
1 public class Person {
2
3     @Multiplicity(0,10)
4     private ArrayList<Car> cars;
5
6     // code weggelassen
7 }
```

4 Problembeschreibung

Die Durchsetzung und Überprüfung der Invarianten könnten auch im bestehenden Code ausprogrammiert werden. Diese Vorgehensweise hat jedoch gravierende Nachteile:

- Aufgabentrennung [sep]:
  Die eigentliche Logik des Programms und die Überprüfung der Invarianten verschmelzen. Der Code wird dadurch unleserlich und schlechter wartbar.
- Boilerplate code [boi]:
  Durch die hinzugefügten Invariantenüberprüfungen wird der Code "aufgeblasen". Die Methoden, welche die Invarianten überprüfen, müssen immer wieder aufgerufen werden und man verletzt somit das *Don't repeat yourself (dry)* Prinzip. Gibt es Änderungen in den Invariantendefinitionen, müssen an verschiedenen Stellen des Codes diese Änderungen nachgeführt werden, was zu Fehlern führen kann.
- Vergessenheit:
  Die Invariantenüberprüfung immer wieder zu programmieren, kann mühsam werden. Beim Implementieren eines neuen Features wird diese Überprüfung schnell einmal nach hinten verschoben und kann anschliessend vergessen werden.

5 Beschreibung der Entwickler

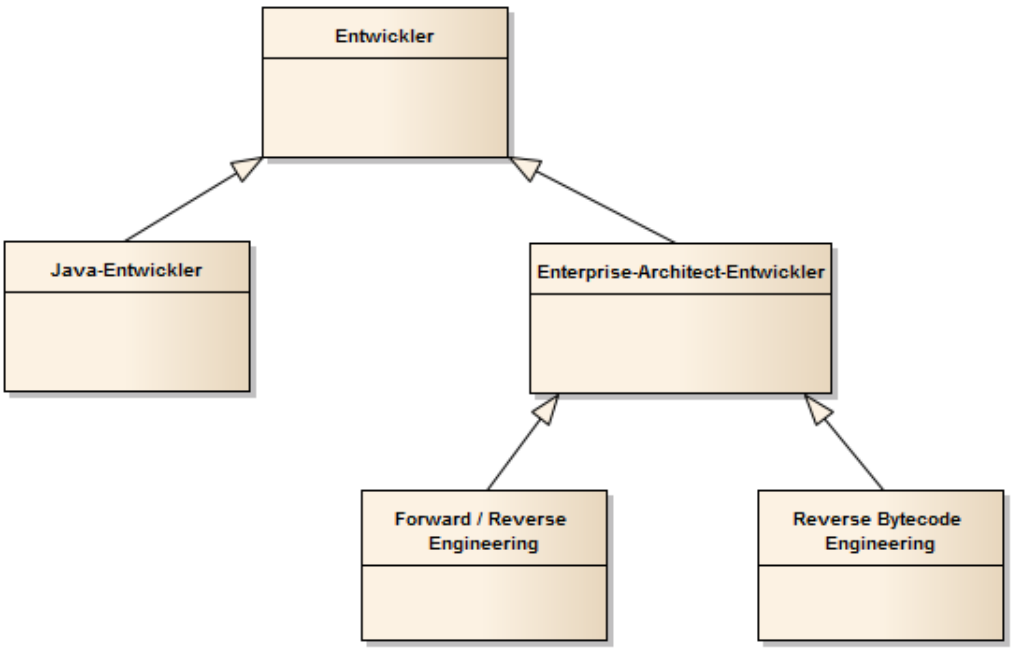


Abbildung 2: Potenzielle Entwickler

Invarianten können im Enterprise Architect modelliert oder direkt im Java Sourcecode mittels Annotationen definiert werden. Daraus ergeben sich folgende potenzielle Entwickler:

- EA - Forward / Reverse Engineering
  Das Ziel des EA-Anwenders ist es beispielsweise, Invarianten auf Attributen, Klassen oder Assoziationen zu modellieren. Später sollen bei der Code-Generierung diese Invarianten in den Sourcecode einfliessen.

- EA - Reverse Bytecode Engineering
  Bestehender Java Bytecode wird im Enterprise Architect importiert (Reverse Engineering). Der Enterprise-Architect-Entwickler modelliert dann Änderungen in Form von Invarianten wie zum Beispiel eine neue Multiplizität und möchte diese Änderungen dann dem bestehenden Bytecode hinzufügen.

- Java-Entwickler
  Ein Java-Entwickler programmiert in einer IDE [ide] und hat das Ziel, Invarianten möglichst schnell definieren zu können. Er will sich nicht darum kümmern müssen, diese selbst zu überprüfen. Die Invarianten sollten als Annotationen festgelegt werden können und anschliessend automatisch überprüft werden. Die Logik zur Überprüfung der Invarianten will der Java-Entwickler nicht selbst programmieren.

6 Poduktübersicht

6.1 Use-Case-Diagramm

Der Akteur kann entweder ein Enterprise-Architect- oder ein Java-Entwickler sein. Es ergeben sich drei Use Cases, welche dem folgenden Use-Case-Diagramm zu entnehmen sind:

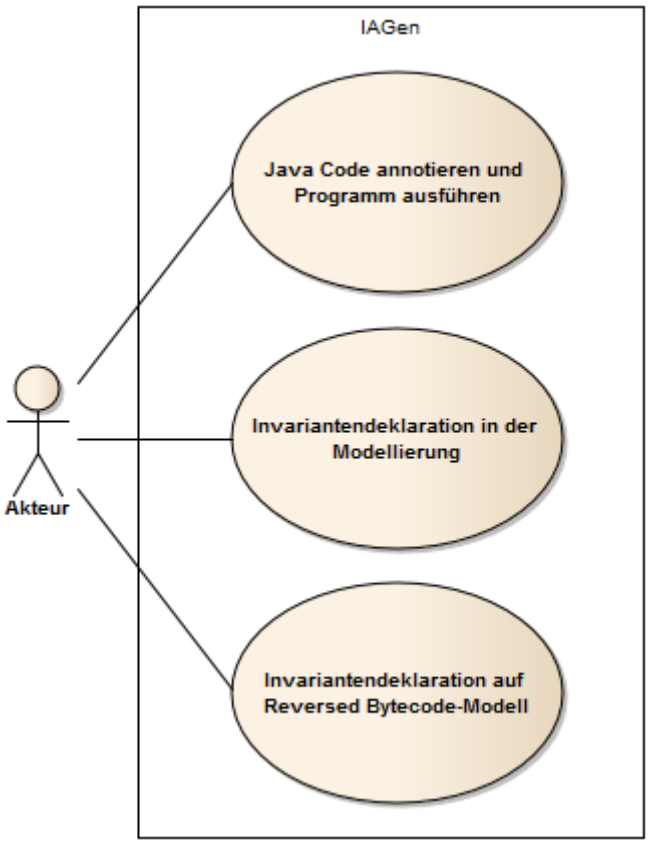


Abbildung 3: Use Case Diagramm von IAGen

6.2 Verhalten bei Invariantenverletzung

Werden zur Laufzeit die Invarianten verletzt, sollten folgende Reaktionen möglich sein:

- Benachrichtigung über Popup
- Logging in eine Datei
- Werfen einer Exception

6.3 Zusammenfassung von Vorteilen

Funktion	Kundennutzen
Enterprise Architect Add-In	Durch die Modellierung im Enterprise Architect sind viele Invarianten bereits gesetzt (beispielsweise Multiplizitäten bei Assoziationen). Durch ein Add-In von IAGen werden bei der Generierung von Code diese Invarianten mit den entsprechenden Annotationen automatisch gesetzt.
Bytecode Reverse Engineering	Im Enterprise Architect lassen sich Modelle auch aus Java-Class-Dateien erstellen. Mittels einem IAGen Add-In soll es möglich sein, die modellierten Änderungen zu konservieren und dem bestehenden Bytecode hinzuzufügen.
IAGen bietet verschiedene Annotationen an.	Der Entwickler kann diese dokumentierten Annotationen verwenden. Durch IAGen soll sichergestellt werden, dass die Annotationen am richtigen Ort (Klasse, Felder, Methoden) gesetzt werden. Denkbare Beispiele sind: • @Min(10), um einen garantieren Minimalwert eines Datenfeldes zu setzen. • @Max(100), um einen garantieren Maximalwert eines Datenfeldes zu setzen. Der Entwickler kann sich dabei vollständig der Implementierung seines Codes widmen. Die Kontrollen der Invarianten übernimmt IAGen. Der Code bleibt somit schlank.
Annotationen werden zur Kompilierzeit auf inhaltliche Korrektheit überprüft.	Beispielsweise macht es keinen Sinn, einen Wertebereich auf einem Datenfeld zu setzen, bei dem der Minimalwert höher ist als der entsprechende Maximalwert. Bei allfälligen Problemen wird der Entwickler bereits zur Kompilierzeit gewarnt.

Literaturverzeichnis

[boi] Definition von Boilerplate code. http://en.wikipedia.org/wiki/Boilerplate_code.

[ent] Enterprise Architect. http://www.sparxsystems.com/products/ea/index.html.

[ide] Definition IDE. http://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung.

[inv] Invarianten in der Informatik. http://de.wikipedia.org/wiki/Invariante_(Informatik).

[sep] Definition von Separation of concerns. http://en.wikipedia.org/wiki/Separation_of_concerns.

IAGen
Invariants-Assertion-Generation

Projektplan

12. Juni 2012

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	02.03.2012	dmengelt	Grobentwurf und Termine
1.0rc02	05.03.2012	dmengelt	Beschreibung Prototype Release
1.0rc03	07.03.2012	dmengelt	Risikomanagement angepasst
1.0rc04	14.03.2012	dmengelt	Erwähnung RUP, UML und AspectJ
1.0rc05	25.03.2012	dmengelt	Kleine Anpassungen
1.0rc06	26.03.2012	dmengelt	Externe Partner
1.0	29.03.2012	dmengelt	Anpassungen gemäss Wochenbesprechung
2.0rc01	10.04.2012	dmengelt	Release 2.0 Grobplanung
2.0	13.04.2012	dmengelt	Grobplanung Release 2.0 fertiggestellt
2.1	20.04.2012	f1egli	Planung Release 2.0 anhand Anforderungen spezifiziert
3.0rc01	30.04.2012	dmengelt	Anpassungen gemäss Wochenbesprechung
3.0	21.05.2012	dmengelt	Grobplanung Release 3.0 fertiggestellt
3.1	25.05.2012	dmengelt	Planung Release 3.0 anhand Anforderungen spezifiziert

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel 5.3 Artefaktenliste

3 Projekt Übersicht

3.1 Projektidee

Die grobe Idee des Projekts kann dem Vision Dokument entnommen werden:

`01_Project-Management/01_Vision/Vision_vX.X.pdf`

Es beschreibt die potenziellen Entwickler und die Probleme, welche ohne IAGen eintreffen können. Zudem zeigt die Vision auf, welche Vorteile der Einsatz von IAGen bringen kann.

3.2 Arbeitsaufwand und Dauer

Der Arbeitsaufwand für ein Projektmitglied während den ersten 14 Wochen beträgt 20 Stunden pro Woche. Anschliessend folgt ein 2-wöchiger Block mit einem Arbeitsaufwand von 45 Stunden pro Woche. Dies ergibt insgesamt 2 * 370 Stunden für das gesamte Projekt.

Das Projekt dauert vom 20.02.2012 bis zum 15.06.2012.

3.3 Vorgehensmodell

Das Vorgehensmodell zur Entwicklung der Software basiert auf dem Unified Process von Craig Larman. [Lar00]

3.4 Entscheid Domainmodell

Weil IAGen keine Datenschicht (Datenmodell) benötigt, wird auf die Erstellung eines Domainmodells verzichtet. Abläufe und Anforderungen an das Produkt können den Anforderungsspezifikationen bzw. dem Architektur und Design Dokument entnommen werden:

`03_Anforderungen/Anforderungsspezifikation_vX.X.pdf`

`04_Architektur_und_Design/Architektur_und_Design_vX.X.pdf`

3.5 Anforderungsspezifikationen

Die Anforderungen beinhalten zu Beginn vor allem das Big Picture des Projekts. Die Funktionalitäten aus der Aufgabenstellung repräsentieren gleichzeitig die wichtigsten Use Cases. Sobald der Prototyp steht, werden die Anforderungen genauer spezifiziert.

`03_Anforderungen/Anforderungsspezifikationen_vX.X.pdf`

Zudem beinhalten die Anforderungen die Wunschvorstellung des Kunden. Der effektive Umfang wird auf Basis der zu Verfügung stehenden Ressourcen definiert.

4 Projektorganisation

4.1 Organisationsstruktur

Name	Frederick Egli	Dominik Mengelt
Kürzel	f1egli	dmengelt
Verantwortung	• Implementation Java Annotationen • Testing • Qualitätsmassnahmen • Developer Guide Java • Architektur und Design Java • Vision • Domain Rules	• Implementation EA Add-In • Risikomanagement • Projektplanung • Anforderungsspezifikationen • Architektur und Design EA Add-In • Developer Guide Add-In • Zeitanalyse

4.2 Externe Partner

Projektbetreuer: Thomas Letsch
Projektexperte: Prof. Dr. Martin Zimmermann
Gegenleser: Prof. Dr. Andreas Müller

5 Management-Abläufe

5.1 Zeiterfassung

Für die Zeiterfassung wird die Projektmanagement Applikation Redmine [Red] eingesetzt. Die Zeit wird jeweils auf eine Aktivität rapportiert:

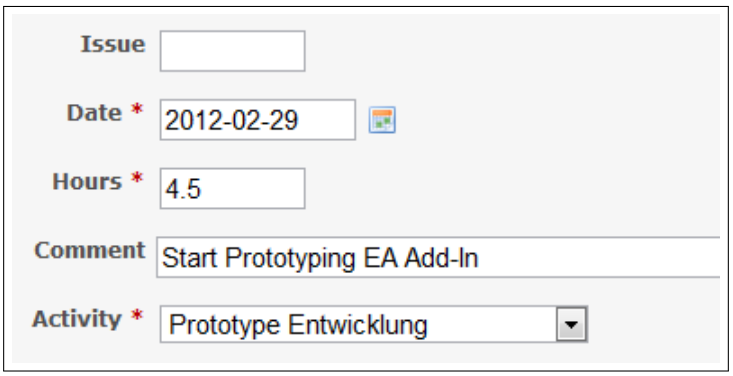


Abbildung 1: Zeiterfassung auf Aktivitäten in Redmine

Sämtliche Artefakte, die während des Projekts entstehen, werden als Aktivitäten erfasst. Die Implementation von IAGen wird auf die Aktivitäten “Prototype Entwicklung“ und “Entwicklung“ aufgeteilt.

Während des Projekts können fortlaufend neue Aktivitäten dazustossen. Dies kann beispielsweise bei erweiterten Funktionalitäten der Releases oder Anpassungen der Anforderungen der Fall sein.

5.2 Wochenbesprechung

Im Normalfall findet jeweils am Donnerstag um 10:10 Uhr eine Wochenbesprechung mit folgendem Aufbau statt:

- Rückblick
- Aktuell
- Ausblick
- Generelles

Das anschliessend zu erstellende Besprechungsprotokoll wird bis 24 Stunden nach der Besprechung an sämtliche Teilnehmer per E-Mail verschickt und abgelegt:

`01_Project-Management/05_Protokolle/YYYYMMDD.txt`

5.3 Artefaktenliste

Die während des Projekts erstellten Artefakte könnend der Projekt-Roadmap entnommen werden:

`01_Project-Management/00_Artefaktenliste/Artefaktenliste_vX.X.pdf`

Sie liefert ebenfalls einen Überblick über das gesamte Projekt. Geplante Releases und fertiggestellte Dokumente sind ersichtlich. Der Projektmitarbeiter kann sich an dieser Liste orientieren. Zusätzlich wird pro Artefakt der Nutzen und die Funktion erklärt. Beispiel:

Vision
Projektvision. Das Big Picture des Projekts.
v1.0 Erste Version mit Problembeschreibung und Zusammenfassung der Vorteile.
v2.0 Vollständige Vision inkl. Beschreibung der Entwickler.

5.4 Releases

Während des Projekts sind folgende 3 Releases geplant:

#	Datum	Name	Iteration/Phase
1.0	04.04.2012	Release 1.0 [Prototype]	Elaboration Woche 7
2.0	16.05.2012	Release 2.0	Construction 3 Woche 13
3.0	15.06.2012	Release 3.0 [final]	Transition 1 Woche 17

5.4.1 Releasepapers

Bei jedem Release wird ein Releaspaper erstellt. Dieses beinhaltet die folgenden Abschnitte:

- New Features
- Defects Fixed
- Known Issues

Sie werden unter folgendem Verzeichnis abgelegt:

`04_Architektur_und_Design/01_Releasepapers/Releasepaper_vX.X.pdf`

5.4.2 Release 1.0 [Prototype]

Im Enterprise Architect [ent] können Wertebereichannotationen [ann] auf primitive Datenfelder festgelegt werden. Wird der Sourcecode danach generiert, fügt IAGen die Annotationen automatisch am richtigen Ort ein. Zur Laufzeit des annotierten Programms werden die gesetzten Invarianten von IAGen überprüft.

Ausserdem ist es möglich, dass ein Java-Entwickler die Annotationen direkt im Java Sourcecode setzen kann.

5.4.3 Release 2.0

Geplante Use Cases und Funktionalitäten:

- **UC01: Java Code annotieren**
  Wertebereiche für sämtliche primitiven Datenfelder und deren Wrapper können festgelegt werden. Auf Java Collections, Maps und Arrays können Multiplizitäten definiert werden. Annotationen processing für *@Min*, *@Max* und *@Range* wird vollumfänglich unterstützt.
- **UC02: Invariantendeklaration auf Source Code aus EA**
  Die im Enterprise Architect modellierten Multiplizitäten werden bei der Generierung des Java Codes als Annotation auf das jeweilige Datenfeld gesetzt.
- **UC03: Invariantendeklaration auf Byte Code aus EA**
  Modellierte Multiplizitäten und Wertebereiche können aus dem Enteprise Architect exportiert werden.
- **Weitere Funktionalitäten**
  - Bei Invariantenverletzung und gesetztem Popup Flag wird der Java-Entwickler zusätzlich den StackTrace anzeigen lassen können.
  - Unicode-Definierung als Annotationswert bei Characters (\uD800).

5.4.4 Release 3.0 [final]

- **UC01: Java Code annotieren**
  Die Annotation *@Subsets* kann auf Datenfelder und Container (Collection, Map, Array) gesetzt werden. Annotation Processing für die *@Subsets* und *@Multiplicity* Annotation wird vollumfänglich unterstützt.
- **UC02: Invariantendeklaration auf Source Code aus EA**
  Die im Enterprise Architect modellierten Subsets werden bei der Generierung des Java Sourcecodes als Annotation auf das jeweilige Datenfeld gesetzt.
- **UC03: Invariantendeklaration auf Byte Code aus EA**
  Modellierte Subsets können aus dem Enterprise Architect exportiert werden.
- **Weitere Funktionalitäten**
  - *@NotNull*-Annotation auf Datenfelder, Methoden- und Konstruktorparameter.
  - *@Min*, *@Max* und *@Range*-Annotation auf Methoden- und Konstruktorparameter.
  - *@Const*-Annotation auf Methoden, Feldern und Methoden- und Konstruktorparameter.
  - Eigene Exceptions und Fehlermeldungen bei einer Invariantenverletzung können definiert werden.
  - Invariantenverletzungen können in eine vom Entwickler definierte Datei geloggt werden.
  - Internes Logging der IAGen Implementation wird möglich sein.

6 Risikomanagement

Für das Projekt werden die wichtigsten Risiken identifiziert und der Aufwand für die Massnahmen eingeschätzt. Die detaillierte Planung kann dem Risikomanagement Dokument entnommen werden:

`01_Project-Management/01_Risikomanagement/Risikomanagement_vX.X.pdf`

7 Issues

Am Ende der Phasen Elaboration, Construction 3 und Transition 1 folgt ein Release. Die geschlossenen Issues pro Release können der Projektmanagement Applikation Redmine entnommen werden. Eine Roadmap zeigt den aktuellen Stand eines Releases:

Abbildung 2: Roadmap mit offenen Issues

7.1 Issue-Kategorien

Issues werden in folgenden Kategorien eingeteilt:

- Feature
  Zeigt die zu implementierenden Funktionalitäten pro Release.
- Defect
  Zeigt die erfassten Bugs pro Release.
- ToDo
  Zeigt die offenen ToDos über das gesamte Projekt.

8 Infrastruktur

Den Projektmitglieder steht ein eigenes Arbeitszimmer mit festem Arbeitsplatz und einem Desktop-PC zur Verfügung. Als Entwicklungsumgebung dient Eclipse Indigo. Beim Versionsmanagement wird ein entsprechender Server eingesetzt.

Die Projektmanagement Applikation Redmine kann unter folgender URL erreicht werden:

`http://www.iagen.ch/`

9 Qualitätsmanagement

9.1 Besprechung & Pair Programming

Einmal in der Woche findet eine Wochenbesprechung mit dem Betreuer statt. Zusätzlich wird für die Qualität des Codes auf Pair Programming gesetzt. Mögliche Refactorings und schlechte Designentscheidungen werden somit schneller erfasst und korrigiert. Zusätzlich hilft es für den Know-how-Transfer innerhalb des Teams.

9.2 Versionsmanagement

Die projektrelevanten Dateien und Sourcecode Files werden mit einem Versionsmanagement Tool (git, svn oder ähnlich) verwaltet. Bei jedem commit ist ein aussagekräftiger Kommentar zu der getätigten Änderung notwendig. Dies wird die Erkennung von gemachten Änderungen vereinfachen und es wird auf einen Blick sichtbar, welcher Projektmitarbeiter für die betroffene Datei verantwortlich ist.

`04_Architektur_und_Design/02_Tools_und_Infrastruktur/Tools_und_Infrastruktur_vX.X.pdf`

9.3 Automatischer Build

Nach jeder Änderung am Sourcecode werden mittels Jenkins [Jen] Build-Server automatisch die JUnit Tests ausgeführt und eine neuer Build erstellt. Sollte eine solche Buildprozedur mittels ANT fehlschlagen, werden umgehend sämtliche Projektmitarbeiter per E-Mail benachrichtigt. Somit können Fehler in den Releases schneller bemerkt und schliesslich behoben werden.

9.4 Project Style Guide

9.4.1 Code Style Guide

Für die Softwareentwicklung gilt während des Projekt die bestehende Cody Style Guide von Oracle (Sun):

`http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`

9.4.2 Code Formatter Eclipse

Das ganze Team verwendet die gleiche Formattervorlage für die Programmierung. Diese muss als XML exportiert und auf das Repository verschoben werden, damit allfällige Änderungen sofort wirksam werden.

9.5 Unit Testing

Die Testabdeckung der internen IAGen-Logiken sollte mit mindestens 90% gedeckt sein. Das heisst, dass die Mechanismen (beispielsweise das Annotationen-Handling) getestet wird.

Literaturverzeichnis

[ann] Java Annotation. http://en.wikipedia.org/wiki/Java_annotation.

[ent] Enterprise Architect. http://www.sparxsystems.com/products/ea/index.html.

[Jen] Jenkins. Offizielle Jenkins-Projekt-Website. http://jenkins-ci.org/.

[Lar00] Craig Larman. *Applying UML and Patterns*. Prentice Hall, 2000. Seite 18.

[Red] Redmine. Offizielle Redmine-Projekt-Website. http://www.redmine.org/.

IAGen Artefaktenliste

HSR Abgaben	Bericht - Poster						v1.0		
	Bericht - Glossar					v1.0	v2.0		
	Bericht - Persönlicher Bericht							v1.0	
	Bericht - Technischer Bericht der Arbeit							v1.0	v2.0
	Bericht - Management Summary							v1.0	
	Bericht - Abstract						v1.0		
	Bericht - Eigenständigkeitserklärung							v1.0	
	Bericht - Aufgabenstellung						v1.0		
	Bericht - Titelblatt							v1.0	
Projekt Management (RUP)	Tools und Infrastruktur						v1.0		v2.0
	QS & Testauswertung								v1.0
	Developer Guide					v1.0			v2.0
	Releasepapers		v1.0			v2.0			v3.0
	Implementation		v1.0			v2.0			v3.0
	UML Modell				v1.0	v2.0			v3.0
	Architektur & Design			v1.0		v2.0			v3.0
	Projektplan		v1.0	v2.0	v2.1			v3.0	v3.1
	Domain Rules		v1.0						v2.0
	Vision	v1.0							
	Anforderungsspezifikationen		v1.0	v2.0					v3.0
	Zeitanalyse	**							
	Protokoll Besprechung	*							
	Woche	1 2	3 4 5 6 7	8 9	10 11	12 13	14 15	16 17	
	Phase	Inception	Elaboration	Construction 1	Construction 2	Construction 3	Construction 4	Transition 1	

Versionen und Beschreibung

Protokoll Besprechung [* = wöchentlich]
Das Protokoll der Wochenbesprechung mit Herrn Letsch. Enthält die diskuktierten Ansätze und Entscheide.

Zeitanalyse [= fortlaufend]**
Zeigt die erfassten Zeiten pro Arbeitspaket. Die Zeiten werden in Redmine erfasst.

Vision
Projektvision. Das Big Picture des Projekts.
v1.0 Erste Version mit Problembeschreibung und Zusammenfassung der Vorteilen.
v2.0 Vollständige Vision inkl. Beschreibung der Entwickler

Domain Rules (Business Rules)
Beschreibt die verwendeten Terminologien wie Invarianten,Subsets, Multiplizitäten, Wertebereiche
v1.0 IAGen Terminologie festgelegt
v2.0 Alle Rules und Terminologien beschrieben

Projektplan
Beschreibt wie das Projekt abgewickelt wird inkl. Termine und Organisation
v1.0 Termine und Abläufe festgelegt. Prototype Release beschrieben.
v2.0 Für den Release 2.0 alle Artefakte und Termine geplant.
v2.1 Beschreibt, welche Teile der Use Cases & Funktionalitäten von den Anforderungen 2.0 bearbeitet werden.
v3.0 Testkonzept und sämtliche Releases beschrieben.
v3.1 Beschreibt, welche Teile der Use Cases & Funktionalitäten von den Anforderungen 3.0 bearbeitet werden.

Anforderungsspezifikationen
Was muss das Produkt können. Enthält weder Termine noch Teamangaben. Ist die Wunschvorstellung aus Kundensicht.
Der effektive Umfang wird auf Grund der zu Verfügung stehenden Ressourcen im Projektplan definiert.
v1.0 Erste Use Cases ausgearbeitet.
v2.0 Weitere Anforderungen definiert.
v3.0 Sämtliche Use Cases und nichtfunktionale Anforderungen ausgearbeitet.

UML Modell
Anhand des Modell wird das Zusammenspiel der einzelnen Klassen und Interfaces aufgezeigt.
v1.0 Erste Version des UML Modells der Architektur.
v2.0 Implementation 2.0 und UML Modell synchron.
v3.0 Finale Implementation 3.0 und UML Modell synchron.

Architektur & Design
Das A&D zeigt Architektur und Design Entscheide innerhalb der Implementation.
v1.0 Planung und Entscheide mit Begründung, Grobe Package Struktur.
v2.0 Verfeinerung der Planung und erste Designentscheide.
v3.0 Alle Designentscheide begründet und sämtliche Diagramme aktualisiert.

Implementation
Source Code und Unit Tests von IAGen (Library und Add-In)
v1.0 Prototype Release. Setzen von Annotationen auf Wertebereiche im EA und im Java Source Code
v2.0 Wertebereiche, Multiplizitäten im EA und im Java Source Code. Annotation Processing.
v3.0 Subsets im EA und im Java Source Code. NotNull und Const Annotationen.

QS & Testauswertung
Enthält die Auswertung der Metriktools (FindBugs, CheckStyle, eCobertura, stan4j) sowies der Junit Tests (Eclemma)
v1.0 Auswertungen der Metriktools / Junit Tests für Release 2.0
v2.0 Auswertung aller genutzten Tools

Releasepapers
Beschreibt die Features zum jeweils aktuellen Release

Developer Guide
Erklärt die Verwendung von IAGen anhand von Beispielen für den Kunden.
Diese Beispiele und Texte können gegebenfalls von den Anforderungsspezifikation kopiert werden.
v1.0 Installation und Verwendung von IAGen. Wertebereiche erklärt.
v2.0 Verwendung aller Annotationen anhand von Beispielen erklärt.

Tools und Infrastruktur
Zeigt die vom Team eingesetzten Tools und Werkzeuge
v1.0 Vorgehen automatische Builds und GIT
v2.0 CheckStyle, FindBugs, eCobertura, stan4j

Bericht - Titelblatt
v1.0 Titelblatt gemäss Vorlage.

Bericht - Aufgabenstellung
v1.0 Die vom Betreuer abgegebene und unterschriebene Aufgabenstellung (eingescannt).

Bericht - Eigenständigkeitserklärung
v1.0 Erklärung gemäss Vorlage.

Bericht - Abstract
v1.0 Der Abstract richtet sich an den Spezialisten auf dem entsprechenden Gebiet und beschreibt daher in erster Linie die (neuen, eigenen) Ergebnisse und Resultate der Arbeit.

Bericht - Management Summary
v1.0 Das Management Summary richtet sich in der Praxis an die "Chefs des Chefs", d.h. an die Vorgesetzten des Auftraggebers. Es darf Texte von anderen Dokumenten enthalten.

Bericht - Technischer Bericht der Arbeit
Enthält Einleitung und Übersicht, Ergebnisse und Schlussfolgerungen.
Darf Texte und Code Abschnitte von anderen Dokumenten erhalten.
v1.0 Einleitung und Übersicht
v2.0 Schlussfolgerungen

Bericht - Persönlicher Bericht
v1.0 Persönliche Berichte einschliesslich (selbst-)kritische Reflexion der Studierenden zu ihren Erfahrungen bei der Arbeit.

Bericht - Glossar
Erklärung zu den unklaren Projektbegriffen und Abkürzungen
v1.0 Erste Begriffe definiert.
v2.0 Fertiggestelltes Glossar.

Bericht - Poster
v1.0 Zusammenfassung der Arbeit auf einem Poster. Muss nicht selbsterklärend sein.

Risikomanagement IAGen

Verantwortlich: Frederick Egli (f1egli) & Dominik Mengelt (dmengelt)

Legende
Infrastrukturelle Risiken
Risiken der Implementation

ID	Risiko	Auswirkung	Massnahme	max. Schaden [h]	Zeitlicher Aufwand der Massnahmen	Wahrscheinlichkeit des Eintreffens	Gewichteter Schaden in Stunden	Priorität
R01	**Server Absturz**	**Kein Zugriff mehr auf Remote Repository.**	**Kopien auf Arbeitsrechner alle 4h aktualisieren**	**4**	**1.0**	5%	0.2	4
R02	**Ausfall des persönlichen Laptops**	**Codepassagen die nicht eingecheckt wurden gehen verloren.**	**Alternative HW organisieren. (HSR Workstations)**	**2**	**1.0**	5%	0.1	5
R03	**Ausfall Netzwerkstruktur**	**Arbeiten am Projekt werden erschwert.**	**Mehrere Möglichkeiten für Datenaustausch innerhalb des Teams bereithalten (Memory-Stick, HD etc.).**	**2**	**4.0**	5%	0.1	5
R04	**Datenverlust**	**Wichtige Projektdaten werden unwiderruflich gelöscht.**	**Artefakte konsequent unter Versionsverwaltung stellen, lokale Kopien mindestens über eine Nacht behalten.**	**1**	**1.0**	10%	0.1	3
R05	**Einarbeitung in neue Technologien (Add-Ins EA, C#, Java Annotations) dauert länger als erwartet**	**Projektplanung gerät in Verzug**	**Know-How Transfers und kurze "Heads-Ups" jeweils wöchentlich.**	**10**	**2.0**	10%	1	2
R06	**Komplexität des Codes wird zu hoch.**	**Einarbeitung eines Projektmitglieds wird schwierig.**	**JavaDoc und vereinzelte Code Reviews machen.**	**4**	**1.0**	20%	0.8	3
R07	**Geplante Releases können nicht zeitgerecht abgeliefert werden**	**Zeitplan kann nicht eingehalten werden**	**Auf die Grundfunktionalitäten konzentrieren und nur die Requirements implementieren**	**25**	**5.0**	25%	6.25	2
R08	**Eingesetzte Technologie (AspectJ, Annotations) erfüllt die gewünschten Anforderungen nicht.**	**Projektplanung gerät in Verzug**	**Vorgängiges Studium alternativer Technologien und Frameworks**	**80**	**20.0**	50%	40	1
	Totaler Aufwand der Massnahmen [h]				**35.0**			
	Total Rückstellungen [h]						**48.55**	

IAGen
Invariants-Assertion-Generation

Domain Rules

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	24.02.2012	f1egli	Erste Version der Vision
1.0rc02	06.03.2012	f1egli, dmengelt	Definitionen und Erklärung
1.0rc03	07.03.2012	f1egli	Wertebereich, Assoziationen und Subsets programmatisch erklärt
1.0rc04	14.03.2012	f1egli	Wertebereich, Assoziationen und Subsets definiert
1.0rc05	26.03.2012	dmengelt	Enterprise-Architect-Erklärungen
1.0	29.03.2012	f1egli, dmengelt	Anpassungen gemäss Wochenbesprechung
2.0rc01	29.05.2012	dmengelt	Erklärung Invariante
2.0	29.05.2012	dmengelt	Finale Version 2.0

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Invarianten

Invarianten in der Programmierung sind wie folgt definiert:

> „In computer programming, specifically object-oriented programming, a class invariant is an invariant used to constrain objects of a class. Methods of the class should preserve the invariant. The class invariant constrains the state stored in the object. Class invariants are established during construction and constantly maintained between calls to public methods. Temporary breaking of class invariance between private method calls is possible, although not encouraged.“[1]

Invariantenüberprüfungen finden zu folgenden Zeitpunkten statt:

- Nach dem Ablauf des Objekt Konstruktors.
- Vor und nach jedem “externen“ Aufruf einer nicht privaten Methode.
- Vor und nach jedem Setzen eines nicht privaten Feldes.

4 Wertebereiche bei Datenfeldern

Ein Wertebereich kann auf drei verschiedene Arten aufgeteilt werden:

- Untere Grenze
  Es wird der minimale Wert angegeben, welcher nicht unterschritten werden darf.
- Obere Grenze
  Es wird der maximale Wert angegeben, welcher nicht überschritten werden darf.
- Bereich
  Es wird ein Bereich angegeben. Der Wert darf diesen Bereich weder über- noch unterschreiten.

[1] Class invariant - http://en.wikipedia.org/wiki/Class_invariant

4.1 Enterprise Architect

Zur aktuellen Version von EA gibt es keine Möglichkeit, einen Wertebereich auf ein Feld festzulegen.

4.2 Programmatische Zusicherung

Die Zusicherung eines Wertebereichs kann bei einer *setter*-Methode des entsprechenden Feldes erreicht werden:

Listing 1: Wertebereich eines Datenfeldes

```
1  public class Car {
2
3      private int numberOfDoors;
4
5      public Car(int numberOfDoors) {
6          try {
7              this.setNumberOfDoors(numberOfDoors);
8          } catch (Exception e) {
9              e.printStackTrace();
10         }
11     }
12
13     public int getNumberOfDoors() {
14         return numberOfDoors;
15     }
16
17     // Nur Türen zwischen zwei und fünf erlauben
18     public void setNumberOfDoors(int numberOfDoors) throws Exception {
19         if (numberOfDoors < 2 || numberOfDoors > 5) {
20             throw new Exception("Türanzahl nicht zwischen zwei und fünf.");
21         }
22         this.numberOfDoors = numberOfDoors;
23     }
24 }
```

4.3 Probleme

- Innerhalb der Klasse muss immer die *setter*-Methode verwendet werden.
- Zusicherung kann nicht garantiert werden, wenn das Feld *public* ist und dadurch von anderen Klassen verändert werden kann.

5 Multiplizitäten

Assoziationen können durch Multiplizitäten [mula] in ihrer Anzahl gesteuert werden. Durch die Angabe eines Bereiches kann man festlegen, ob ein Attribut optional, einwertig oder mehrwertig ist. Folgende Beispiele sollen dies genauer erläutern:

- Multiplizität *0..1* bedeutet, entweder keine oder eine Instanz ist assoziiert.
  Ist eine optionale Assoziation.
- Multiplizität * oder (gleichbedeutend) *0..** bedeutet, keine, eine oder mehrere Instanz/en sind assoziiert.

- Multiplizität *5* oder (gleichbedeutend) *5..5* bedeutet, dass genau fünf Instanzen assoziiert sind.
- Multiplizität *1..** bedeutet, eine oder mehrere Instanz/en sind assoziiert.
- Multiplizität *5..10* bedeutet, es sind zwischen fünf und zehn Instanzen assoziiert.

Seit UML2 ist es nicht mehr möglich, Multiplizitäten mit mehreren unzusammenhängenden Zahlenabschnitten zu kombinieren. So ist es nicht mehr möglich, eine Multiplizität von *0..4, 8..** (also alle Zahlen ausser 5,6 und 7) zu setzen. [mulb]

5.1 Enterprise Architect

Multiplizitäten werden im Enterprise Architect über eine Assoziation modelliert.

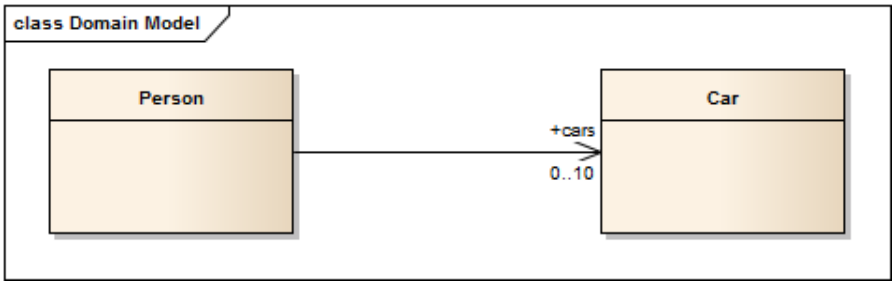


Abbildung 1: Modellierung einer Multiplizität im Enterprise Architect

Die *cars* Collection in der Klasse *Person* beinhaltet maximal 10 Instanzen der Klasse *Car*.

5.2 Programmatische Zusicherung

Man könnte die Zusicherung der Multiplizitäten folgendermassen einführen:

Listing 2: Multiplizität auf eine Liste

```
1  public class Person {
2
3      private List<Car> cars;
4
5      public Person() {
6          cars = new ArrayList<Car>();
7      }
8
9      // Eine Person darf nur zwischen null und zehn Autos besitzen
10     public void addCarToPerson(Car c) throws Exception {
11         if (cars.size() >= 10) {
12             throw new Exception();
13         }
14         cars.add(c);
15     }
16
17     public List<Car> getAllCars() {
18         return cars;
19     }
20 }
```

5.3 Probleme

- Innerhalb der Klasse muss immer die *add*-Methode (im Beispiel: *addCarToPerson()*) verwendet werden.
- Gibt es eine *get*-Methode, welche die Liste unverändert (weder geklont noch als immutable List) retourniert (im Beispiel: *getAllCars()*), kann diese herangezogen werden und ein entsprechendes *add* ausgeführt werden. Somit würde der Multiplizitätscheck verletzt werden:

Listing 3: Multiplizitätsüberprüfung ausgehebelt

```
1  public class Main {
2      public static void main(String[] args) throws Exception {
3          Person p = new Person();
4          for (int i = 1; i < 15; i++) {
5              ////Wirft eine Exception wenn i > 10
6              //p.addCarToPerson(new Car(2));
7
8              //keine Multiplizitätsüberprüfung
9              p.getAllCars().add(new Car(2));
10         }
11     }
12 }
```

- Zusicherung kann nicht garantiert werden, wenn die Liste *public* ist und dadurch von anderen Klassen direkt angesprochen werden kann.

6 Subsets

Ein Subset [Bal05] beschreibt eine Teilmenge von Objektbeziehungen. Wenn zwischen zwei Klassen mehr als eine Assoziation besteht, kann man auf eine der beiden Assoziationen einen Subset definieren. Als Beispiel: Man hat eine *Player* und eine *Tournament*-Klasse. Ein *Player* spielt an verschiedenen Turnieren (erste Assoziation) und ein *Tournament* hat (zumindest am Ende eines Turnier) einen *Player* als Gewinner (zweite Assoziation). Auf der zweiten Assoziation kann nun ein Subset definiert werden: Ein Gewinner eines Tournaments muss daran teilgenommen haben. Die zweite Assoziation ist also eine Teilmenge der ersten.

6.1 Enterprise Architect

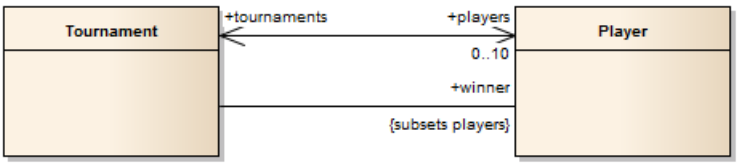


Abbildung 2: Modellierung eines Subsets im Enterprise Architect

Im Enterprise Architect können Subsets nicht speziell modelliert werden. Man verwendet die UML-konformen Curly Brackets (= Constraints) und referenziert auf ein Rollennamen:

{*subsets <role-name>*}

6.2 Programmatische Zusicherung

Man könnte die Zusicherung der Subsets folgendermassen lösen:

Listing 4: Player

```
1  public class Player {
2
3      private final String name;
4      private List<Tournament> tournaments;
5
6      public Player(String name) {
7          this.name = name;
8          tournaments = new ArrayList<Tournament>();
9      }
10
11     public void addTournament(Tournament t){
12         tournaments.add(t);
13     }
14
15     public List<Tournament> getTournaments(){
16         return tournaments;
17     }
18 }
```

Listing 5: Subset - ein Gewinner muss beim Tournament dabei gewesen sein

```
1  public class Tournament {
2
3      private final String tournamentName;
4      private List<Player> players;
5      private Player winner;
6
7      public Tournament(String tournamentName) {
8          this.tournamentName = tournamentName;
9          players = new ArrayList<Player>();
10     }
11
12     public void addPlayer(Player p) {
13         players.add(p);
14     }
15
16     // Subset - Ein Gewinner muss beim Tournament dabei gewesen sein
17     public void setWinner(Player p) throws Exception {
18         for (Player player : players) {
19             //vergleich über Referenz
20             if (p == player) {
21                 winner = p;
22                 return;
23             }
24         }
25         throw new Exception();
26     }
27 }
```

Folgender Code würde also zu einer Exception führen:

Listing 6: Exception - Hans hat nicht am Tournament teilgenommen

```
1  public class Main {
2      public static void main(String[] args) throws Exception {
3
4          Player hans = new Player("hans");
5          Tournament t1 = new Tournament("t1");
6
7          Tournament notWithHans = new Tournament("notWithHans");
8
9          hans.addTournament(t1);
10         t1.addPlayer(hans);
11         t1.setWinner(hans);
12
13         //Exception
14         notWithHans.setWinner(hans);
15     }
16 }
```

6.3 Probleme

- Innerhalb der Klasse muss immer die *setter*-Methode (im Beispiel: *setWinner(Player p)*) verwendet werden.
- Zusicherung kann nicht garantiert werden, wenn das Subset Feld *public* ist und dadurch von anderen Klassen verändert werden kann.

Literaturverzeichnis

[Bal05] Prof. Dr. Heide Balzert. *Folienskript zur Dozenten-CD-ROM Lehrbuch der Objektmodellierung*. W3L-Verlag, 2005. Seite 20.

[mula] Multiplizität. http://de.wikipedia.org/wiki/Multiplizit%C3%A4t_%28UML%29.

[mulb] Multiplizität mit UML2. http://de.wikipedia.org/wiki/Attribut_%28UML%29#Unterschiede_zur_UML_1.4.

IAGen
Invariants-Assertion-Generation

Anforderungsspezifikationen

12. Juni 2012

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	06.03.2012	dmengelt	Punkte aus Aufgabenstellung als Anforderungen
1.0rc02	07.03.2012	dmengelt	Annahmen und Einschränkungen angepasst
1.0rc03	08.03.2012	dmengelt	IAGen Highlevel Ansicht
1.0rc04	25.03.2012	f1egli	Wertebereich
1.0rc05	25.03.2012	dmengelt	Überarbeitung Produktperspektive
1.0rc06	26.03.2012	f1egli	Use Case Java Code annotieren
1.0	29.03.2012	f1egli, dmengelt	Anpassungen gemäss Wochenbesprechung
2.0rc01	19.04.2012	f1egli, dmengelt	Weitere Anforderungen definiert
2.0rc02	22.04.2012	f1egli	Multiplicity
2.0rc03	22.04.2012	f1egli	Use Case mit Multiplicity erweitert
2.0rc04	22.04.2012	dmengelt	Nichtfunktionale Anforderungen
2.0	22.04.2012	dmengelt	Anforderungen Release 2.0
3.0rc01	30.04.2012	dmengelt	Anpassungen gemäss Wochenbesprechung
3.0rc02	21.05.2012	f1egli	Use Case ergänzt
3.0rc03	21.05.2012	dmengelt	Enterprise Architect Use Cases abgeändert
3.0rc04	23.05.2012	dmengelt	Weitere optionale Funktionen
3.0rc05	29.05.2012	dmengelt	@Const auf Felder
3.0	31.05.2012	dmengelt, f1egli	Finale Version 3.0

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Produktperspektive

3.1 Ausgangslage

3.1.1 Enterprise Architect

Der Entwickler modelliert mit dem CASE-Tool [cas] Enterprise Architect [ent] ein Klassendiagramm:

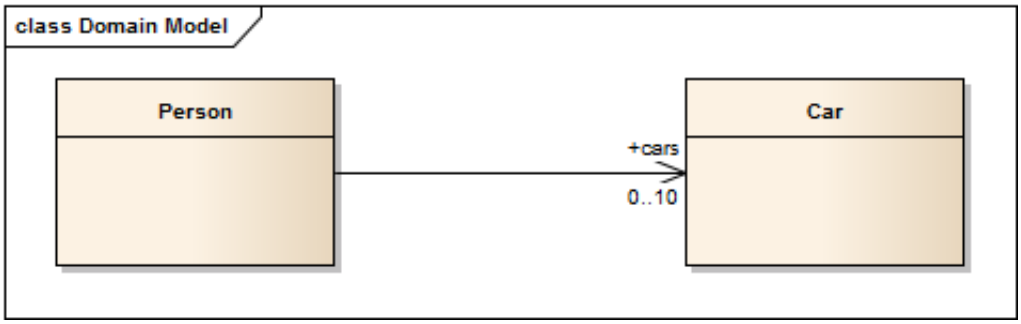


Abbildung 1: Zwei Klassen modelliert im Enterprise Architect

Anschliessend werden mittels “Code Generation“ [cod] diese modellierten Klassen generiert. Definierte Invarianten wie Multiplizitäten bei Assoziationen oder Wertebereiche bei primitiven Datenfeldern gehen bei diesem Prozess jedoch verloren.

3.1.2 Java Sourcecode

Eine ähnliche Problematik entsteht, wenn man ohne CASE-Tool arbeitet und versucht, diese Invarianten direkt im Java Sourcecode zu überprüfen. Einen Wertebereich für ein Integer-Datenfeld kann man nicht festlegen. Die Überprüfung muss also programmatisch vorgenommen werden. Das gleiche gilt natürlich auch bei Multiplizitäten und weiteren Invarianten. Die programmatische Einhaltung und die daraus resultierenden Probleme dieser Invariantenüberprüfung wird in den Domain Rules aufgezeigt:

`01_Project-Management/02_Domain_Rules/Domain_Rules_vX.X.pdf`

3.2 Verwendung von IAGen

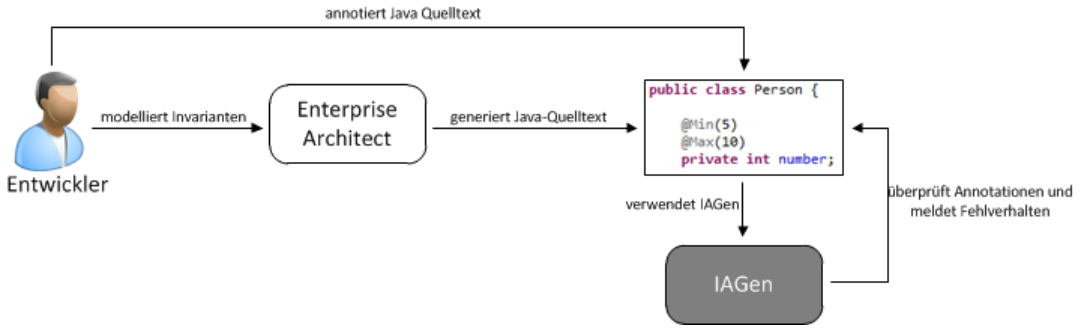


Abbildung 2: Verwendung von IAGen durch Entwickler

IAGen soll als Library in die bestehende Entwicklungsumgebung eingebunden werden können. Die im Programmcode enthaltenen Annotationen (in der Grafik @Min und @Max) werden dann zur Laufzeit ausgewertet. Sollten diese nicht eingehalten werden, wird dem Benutzer das Fehlverhalten aufgezeigt. Die Annotationen können vom Entwickler im Enterprise Architect oder direkt im Programmcode definiert werden.

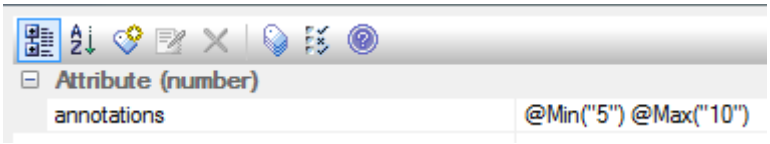


Abbildung 3: Erfassen von Invarianten im Enterprise Architect

Es soll dabei keine Rolle spielen, ob ein neues Klassendiagramm modelliert wird oder ob bereits vorhandener Java Code importiert wird und darauf nachträglich Invarianten definiert werden.

4 Anforderungen

4.1 Hauptfunktionalitäten

Die IAGen Library soll zur Laufzeit die im Code vorhandenen Annotationen überprüfen. Dabei sollten folgende Typen von Annotationen definiert werden können:

- Wertebereiche bei allen primitiven Datentypen (inklusive deren Wrapper-Klassen) und dem Datentyp String
- Multiplizitäten bei Assoziationen
- Subsets bei Assoziationen

Die Annotationen können auf zwei unterschiedliche Wege definiert werden:

- Generierung der Annotationen aus dem Enterprise Architect

  Der Entwickler modelliert die Invarianten direkt im Enterprise Architect und verwendet anschliessend die “Code Generation“-Funktion, um den Sourcecode mit den Annotationen zu erstellen.

- Definition der Annotationen im Java Sourcecode

  Möchte der Entwickler Enterprise Architect nicht verwenden, soll man die Annotationen auch direkt im Java Sourcecode vornehmen können.

4.1.1 Wertebereiche

Es soll drei Annotierungsarten für die Invarianten bei Wertebereichen geben. *@Min* für die Untergrenze, *@Max* für die Obergrenze und *@Range* für einen Bereich (Unter- und Obergrenze). Alle primitiven Datentypen (ausser boolean) und die String-Felder müssen sich mit den oben genannten Annotierungen versehen lassen können. Diese sind namentlich:

char, Character, byte, Byte, short, Short, int, Integer, long, Long, float, Float, double, Double und String

Setzt man Wertebereiche auf ein String-Feld, so wird die Länge des Strings als Bereich interpretiert.

Listing 1: model darf maximal 20 Zeichen lang sein

```
1 @Invariant
2 public class Car {
3
4     @Max(value="20")
5     private String model;
6 }
```

Bei den Wrapperklassen/String und einer Grenze von 0 darf die Klasse auch null sein, ohne dass es zu einer Invariantenverletzung kommt:

Listing 2: Keine Invariantenverletzung

```
1 @Invariant
2 public class Car {
3     @Min("0")
4     @Max("10")
5     public Integer i = null;
6 }
```

Der Auftraggeber wünscht, dass die Werte innerhalb der Annotation als String übergeben werden.

Listing 3: Wert von @Min als String übergeben

```
1 @Invariant
2 public class Car {
3
4     @Min(value="2")
5     private int numberOfDoors;
6 }
```

Die Alternative wäre, für jeden Datentyp eine eigene Annotierung festzuelgen. So gäbe es: *@MaxInt, @MaxDouble* etc.

Da die Annotierungen auf verschiedenen primitiven Datentypen und ihren Wrapperklassen gesetzt werden können, wird der Wert als String gesetzt und zur Kompilierzeit auf Parsbarkeit mit dem Annotationprocessor [apt] überprüft.
Die Verwendung einer einzigen Annotation für alle unterstütze Datentypen ist für einen Entwickler wesentlich einfacher und die Typensicherheit wird beim Kompilieren dennoch

sichergestellt.
Durch die Wertübergabe als String gibt es jedoch eine Limitation für den Datentyp char bzw. Character, da man einen Wert mit Apostrophen oder als Zahl definieren kann:

- public char c1 ='1';
- public char c2 = 1;

Durch die Übergabe als String könnte so nicht mehr zwischen '1' und 1 unterschieden werden. Aus diesem Grund soll IAGen die Werte für chars und Characters so interpretieren, als würden die sie mit Apostrophen (' ') gesetzt. Zudem soll die Unicodenotation (4.3.3) unterstützt werden.

Eine IDE [ide], die Annotationprocessing unterstützt, soll zur Kompilierzeit folgende Warnungen bzw. Errors ausgeben können:

- Error: Annotierung auf einem nicht unterstütze Datentyp.
- Error: Wert der Annotation (welcher als String übergeben wird) kann nicht zum entsprechenden Feldtyp geparsed werden (Beispiel: "1.5" auf int Feld, "128" auf byte Feld).
- Warning: Minimalwert ist grösser gesetzt als der Maximalwert.
- Warning: Es sind alle drei Annotierungen gesetzt (Beispiel: *@Min("1") @Max("2") @Range(lower="3", upper="4")*).

4.1.2 Multiplizitäten

Multiplizitäten sollen mit der Annotation *@Multiplicity* definiert werden können. Man soll diese Annotation auf Felder vom Typ *Collection, Map* und *Array* (primitive arrays - [ ]) setzen können. Die untere Grenze einer Multiplizität soll mit dem Argument *lower*, die obere mit dem Argument *upper* gesetzt werden.

Listing 4: Beispiel einer Multiplizität von 1..10

```
1 @Invariant
2 public class Person {
3
4     @Multiplicity(lower=1, upper=10)
5     public ArrayList<Car> cars;
6 }
```

Für lower sowie für upper müssen Standardwerte definiert werden:

- lower hat den Standardwert 0
- upper hat den Standardwert *Integer.MAX_VALUE*, was einem * entspricht.

Dadurch kann eine Multiplizitätsdefinition vereinfacht werden:

Listing 5: Beispiel einer Multiplizität von 3..*

```
1 @Invariant
2 public class Person {
3
4     @Multiplicity(lower=3)
5     public ArrayList<Car> cars;
6 }
```

Der Datentyp für upper und lower wird als *int* definiert, da die *size()* bzw. *length* Methoden von *Collection*, *Map* und *[ ]* einen *int* zurückgeben.

Wenn lower als 0 definiert wird, darf das annotierte Feld auch *null* sein:

Listing 6: Keine Invariantenverletzung nach Konstruktoraufruf

```
1 @Invariant
2 public class Person {
3
4     @Multiplicity(lower=0, upper=1)
5     public ArrayList<Car> cars;
6
7     public Person() {
8         cars = null;
9     }
10 }
```

Bei einer Multiplicity-Definition von: *lower=1*, *upper=1* wäre allerdings die Invariante nach dem Konstruktoraufruf verletzt gewesen.

Eine IDE, die Annotationprocessing unterstützt, soll zur Kompilierzeit folgende Warnungen bzw. Errors ausgeben können:

- Error: Annotierung auf einem nicht unterstützten Datentyp.
- Error: Der Wert von lower/upper ist nicht im Bereich zwischen 0 und *Integer.MAX_VALUE*.
- Warning: Der Wert von lower ist grösser gesetzt als der Wert von upper.

4.1.3 Subsets

Subsets werden mit der Annotation *@Subsets* definiert. Man soll diese Annotation auf den Feldern vom Typ *Collection, Map* und *Array* (primitive arrays - [ ]) setzen können. Es soll auch möglich sein, diese Annotation auf einem Feld zu setzen, das zum Typ passt, welche in der "grösseren" *Collection/Map/Array* gespeichert wird. In der *@Subsets*-Annotation wird man mit einem Stringparameter den Namen der grösseren Liste angeben können. Als Beispiel dient eine Turnier-Klasse, welche eine Liste mit Spielern enthält ("grössere" Liste) und bei der ein Gewinner, welcher auch ein Spieler ist (gleicher Typ wie "grössere" Liste enthält), gesetzt werden kann.

Listing 7: Keine Invariantenverletzung, wenn Gewinner bei Tournament mitgespielt hat

```
1  @Invariant
2  public class Tournament {
3
4      private List<Player> players;
5
6      @Subsets(value="players") // winner muss in der players-Liste vorkommen
7      private Player winner;
8
9
10     public Tournament() {
11         players = new ArrayList<Player>();
12     }
13
14     public void addPlayer(Player p) {
15         players.add(p);
16     }
17
18     public void setWinners(Player winner) {
19         this.winner = winner;
20     }
21
22 }
```

Werden mehrere Gewinner für ein *Tournament* möglich sein, so wird das Feld winner zu einer Liste. Die Annotation soll auch dann unterstützt werden:

Listing 8: winners-Liste muss in players-Liste enthalten sein

```
1  @Invariant
2  public class Tournament {
3
4      private List<Player> players;
5
6      @Subsets(value="players")
7      private List<Player> winners;
8
9
10     public Tournament() {
11         players = new ArrayList<Player>();
12         winners = new ArrayList<Player>();
13     }
14
15     public void addPlayer(Player p) {
16         players.add(p);
17     }
18
19     public void setWinners(Player p) {
20         winners.add(p);
21     }
22 }
```

Es wird nicht verlangt, dass das annotierte Feld und die grössere Collection/Map/Array vom gleichen Typ sind. Das heisst, dass winners im oberen Beispiel auch ein Array [ ] sein dürfte:

```
@Subsets(value="players")
private Player[] winners;
```

Es wird keine Invariantenverletzung geben, wenn das annotierte Feld als *null* gesetzt wird. Wenn das annotierte Feld vom Typ *Collection/Map/Array* ist, sollen gespeicherte *null*-Werte ebenfalls zu keiner Invariantenverletzung führen.
Eine IDE, die Annotationprocessing unterstützt, soll zur Kompilierzeit folgenden Error ausgeben können:

- Feldname von Parameter von *@Subsets* wurde nicht gefunden.

4.2 Verhalten bei Invariantenverletzung

Wird eine Invariante verletzt, soll der Benutzer von IAGen folgende Möglichkeiten im Vorhinein definieren können:

- Exception
  Es wird eine Exception geworfen, wenn die Invariante verletzt wird. Dies soll die Standardreaktion sein. Es soll möglich sein, eigene Exceptions anzugeben.
- Popup
  Ein Swing Popup erscheint, sobald die Invariante verletzt wurde. Diese soll synchron sein, das heisst, dass Programm arbeitet erst weiter, wenn die Meldung quittiert wurde.
- Logging
  IAGen soll in der Lage sein, Verletzungen in eine Datei zu protokollieren.
- Debug
  Debuginformationen sollen über die Konsole ausgegeben werden können.

Diese Reaktionen lassen sich miteinander verknüpfen. Wird keine Reaktion konfiguriert, so wird nur eine sinnvolle Exception geworfen.

4.3 Optionale Funktionalitäten

4.3.1 Enterprise Architect Reverse Engineering

Der Enterprise Architect bietet auch Reverse-Engineering-Möglichkeiten [rev] an. Bestehender Java Bytecode (.class-Dateien) kann importiert werden. Anschliessend kann der Benutzer Änderungen modellieren.

IAGen soll die Möglichkeit bieten, die modellierten Invarianten aus dem Enterprise Architect zu exportieren. Die bestehende Applikation (Bytecode) soll anschliessend zusammen mit diesen Änderungen gestartet werden können.

4.3.2 Verhalten bei ungültigen Annotationen

Die modellierten Invarianten eines Enterprise-Architect-Entwicklers können erst zur Laufzeit überprüft werden. Wird beispielsweise auf ein Integer-Datenfeld eine ungültige Annotation wie *@Min(“1.5“)* gemacht, so soll der Entwickler zur Laufzeit eine Exception erhalten.

4.3.3 Unicode Character als Annotationswert

Bei den Wertebereichen (*@Min, @Max und @Range*) sollte es möglich sein, den Wert eines char/Character-Feldes mittels Unicodenotation zu setzen:

Listing 9: Wert als Unicode

```
1 @Invariant
2 public class Person {
3
4     @Min("\u0033")
5     @Max("\u00ff")
6     public char c;
7 }
```

4.3.4 StackTrace-Anzeige bei Popup

Bei einer Invariantenverletzung soll das Popup eine Möglichkeit bereitstellen, den momentanen StackTrace anzeigen zu lassen.

4.3.5 PopUps quittieren

Bei einer Invariantenverletzung soll das Popup eine Möglichkeit bereitstellen, die aktuelle Invariante quittieren zu lassen. Danach würde diese Invariantenverletzung nicht mehr angezeigt werden.

4.3.6 Asynchrone Popup

Es sollte die Möglichkeit geben, die Popup-Meldung asynchron zu starten. Das Programm würde in seinem Thread weiterlaufen, ohne dass die Meldung akzeptiert werden müsste.

4.3.7 @NotNull

Die *@NotNull* Annotation soll dem Entwickler garantieren, dass ein beliebiges Objekt niemals den Wert *null* annehmen kann.

Listing 10: @NotNull Annotation auf ein Feld

```
1 @Invariant
2 public class Person {
3
4     @NotNull
5     private Car car;
6
7 }
```

4.3.8 @Const

Ausführliche Beispiele zur *@Const* Annotation sollen im Developer Guide festgehalten werden:

`04_Architektur_und_Design/00_Developer_Guide/Developer_Guide_vX.X.pdf`

Wird ein Datenfeld mit *@Const* annotiert, dann sollen in diesem Feldobjekt keine Werte geändert werden dürfen. Es sollen ausserdem nur Methoden auf diesem Feld aufgerufen werden können, die mit *@Const* gekenzeichnet sind (Erklärung zu *@Const*-Methode folgt in diesem Kapitel). Als Beispiel dient eine Personklasse. Eine *Person* hat ein *Car* Datenfeld, welches mit *@Const* definiert ist.

Listing 11: @Const Annotation auf einem Feld

```
1  @Invariant
2  public class Person {
3
4      @Const private Car car;
5
6      public Person() {
7          car = new Car();
8      }
9
10     public void modifyCar(){
11         car.setNumberOfDoors(6);     // Invariantenverletzung! Kein @Const!
12     }
13
14     public int getCarPs(){
15         return car.getPs();      // Keine Invariantenverletzung!
16     }
17 }
18
19 @Invariant
20 public class Car {
21
22     private int doors;
23     private int ps;
24
25     public void setNumberOfDoors(int doors) {
26         this.doors = doors;
27     }
28
29     @Const public int getPs() {
30         return ps;
31
32     }
33 }
```

Eine Methode, die mit *@Const* annotiert ist, soll *this* nicht verändern können. Wenn eine Const-Methode eine andere Methode innerhalb der gleichen Klasse aufruft, so soll diese ebenfalls Const sein. Als Beispiel dient eine Personklasse.

Listing 12: setName(String name) führt zu einer Exception

```
1  @Invariant
2  public class Person {
3
4      private String name;
5
6      public Person(String name) {
7          this.name = name;
8      }
9
10     @Const public void setName(String name) {
11         this.name = name; //This wird verändert, obwohl Const gesetzt ist.
12     }
13 }
```

Wenn ein Argument eines Konstruktors oder einer Methode mit *@Const* annotiert ist, sollen auf diesem Argument keine Änderungen vollzogen werden können. Es sollen auch nur *@Const*-Methoden auf diesem Argument aufgerufen werden können. Als Beispiel dienen erneut die Klassen Car und Person:

Listing 13: @Const Annotation auf einem Methodenparameter

```
1  @Invariant
2  public class Person {
3
4      private Car c;
5
6      public void setCar(@Const Car c){
7          c.getPs();                  // Keine Invariantenverletzung!
8          c.setNumberOfDoors(4)       // Invarientenverletzung! Kein @Const!
9      }
10 }
11
12 @Invariant
13 public class Car {
14
15     private int ps;
16     private int doors;
17
18     @Const private int getPs(){
19         return ps;
20     }
21
22     private void setNumberOfDoors(int doors){
23         this.doors = doors;
24     }
25 }
```

4.3.9 Annotationen auf Parameter einer Methode

Die Annotationen *@Min*, *@Max*, *@Range* und *@NotNull* sollen auch auf Methoden- oder Konstruktorparameter gesetzt werden können:

Listing 14: Annotationen auf Methoden- und Konstruktorparameter

```
1  @Invariant
2  public class Person {
3
4      private String name;
5      private int age;
6
7      public Person(@Min("18") int age) {
8          this.age = age;
9      }
10
11     public void setName(@Range(min="3", max="50") String name){
12         this.name=name;
13     }
14 }
```

Parameterannotationen sollen von IAGen gleich behandelt werden wie Annotationen auf ein Feld.

5 Use Cases

5.1 Einleitung

Das Format der einzelnen Use Cases [Lar00a] entspricht der Vorgehensweise von Craig Larman im Unified Process. [Lar00b]

5.2 UC01: Java Code annotieren und Programm ausführen

Umfang	IAGen
Ebene	Anwenderziel
Primäraktor	Java-Entwickler
Stakeholder und Interessen	Java-Entwickler • will IAGen möglichst einfach in der IDE seiner Wahl integrieren. • will Annotierungen an dem für ihn logischen Ort im Programmcode platzieren, um Invarianten während der Laufzeit zu überprüfen. • will Rückmeldung bei falsch gesetzten Annotationen (am falschen Ort, auf falschen Datentyp) oder wenn Annotationen inhaltlich falsch sind (Widersprüche, falscher Wert für Datentyp.)
Vorbedingungen	• IAGen.jar wurde als Bibliothek in die Entwicklungsumgebung eingebunden. • IAGen.jar wurde als Annotation Processor in der Entwicklungsumgebung eingebunden.
Standardablauf	1. Java-Entwickler annotiert den Programmcode entsprechend seiner Invarianten. 2. Java-Entwickler entscheidet pro Annotation, wie sich IAGen bei einer Invariantenverletzung verhalten soll. 3. Java-Entwickler kompiliert annotiertes Programm mit dem Annotation Processor. 4. Java-Entwickler startet das annotierte Programm. 5. IAGen überprüft alle gesetzten Invarianten und reagiert bei Verletzungen, wie es der Java-Entwickler vorgängig konfiguriert hat.

Erweiterungen (oder alternative Abläufe)	*a. Jederzeit, wenn Java-Entwickler eine Annotation auf falschem Datentyp oder am falschen Ort gesetzt hat: 1. Java Compiler wird mit dem Annotation Processor gestartet und kompiliert den annotierten Code. 2. IDE bzw. Konsole meldet, dass Annotation auf falschem Datentyp beziehungsweise am falschen Ort gesetzt wurde. 1-3a. Java-Entwickler nahm als Annotation einen Wertebereich (*@Max*, *@Min*, *@Range*), setzte einen Wert als String, der aber nicht zum Feldtyp passte (Beispiel: "1.5" auf int). 1. Beim Kompilieren parst der Annotation Processor alle gesetzten Stringwerte der Wertebereiche. 2. IDE gibt Fehler aus, dass dieser Wert nicht zum entsprechenden Datentyp passt. 1-3b. Java-Entwickler nahm alle drei Wertebereich (*@Max*, *@Min*, *@Range*) als Annotation. (Beispiel: *@Min("1") @Max("2") @Range(lower="3", upper="4")*) 1. Der Annotation Processor gibt eine entsprechende Warnmeldung aus. 1-3c. Java-Entwickler nahm als Annotation einen Wertebereich (*@Min* und *@Max* oder *@Range*) und setzte als Untergrenze einen höheren Wert als die Obergrenze. 1. Der Annotation Processor überprüft, ob inhaltliche Widersprüche gefunden werden, und gibt bei Vorhandensein eine entsprechende Warnmeldung aus. 1-3d. Java-Entwickler nahm als Annotation eine Multiplizität (*@Multiplicity*) und setzte als Untergrenze (lower) einen höheren Wert als die Obergrenze (upper). 1. Der Annotation Processor überprüft, ob inhaltliche Widersprüche gefunden werden, und gibt bei Vorhandensein eine entsprechende Warnmeldung aus 1-3e. Java-Entwickler nahm als Annotation eine Multiplizität (*@Multiplicity*) und setzte einen Wert, der nicht im Bereich zwischen 0 und *INTEGER.MAX_VALUE* liegt. 1. Der Annotation Processor überprüft, ob der Bereich der Multiplizitätswerte nicht stimmt, und gibt dann bei Vorhandensein eine entsprechende Fehlermeldung aus. 1-3f. Java-Entwickler nahm als Annotation ein Subsets (*@Subsets*) und setzte als Wert ein Feld, das nicht gefunden werden konnte.

	1. Der Annotation Processor überprüft, ob das Feld (Wert der Annotation) vorhanden ist, und gibt bei Nichtvorhandensein eine entsprechende Fehlermeldung aus.
Spezielle Anforderungen	Wenn das annotierte Programm gestartet wird, muss das Java-Argument -javaagent mit Pfad zum IAGen.jar gesetzt sein.
Liste der Technik- und Datenvariationen	-
Häufigkeit des Auftretens	Fortlaufend

5.3 UC02: Invariantendeklaration in der Modellierung

Umfang	IAGen
Ebene	Anwenderziel
Primäraktor	Enterprise-Architect-Entwickler
Stakeholder und Interessen	Enterprise-Architect-Entwickler • Will die modellierten Invarianten als Annotationen in den Java Sourcecode generieren lassen. • Will das IAGen Enterprise Architect Add-In einfach installieren.
Vorbedingungen	IAGen wurde als Enterprise Architect Add-In (Extension) installiert.
Standardablauf	1. Der Enterprise-Architect-Entwickler modelliert Diagramme mit diversen Invarianten wie Wertebereiche für Datenfelder oder Multiplizitäten bei Assozationen. 2. Der Enterprise-Architect-Entwickler generiert mittels IAGen Add-In die modellierten Invarianten als Annotationen in die jeweiligen Datenfelder und Klassen (*@Invariant*). 3. Der Enterprise-Architect-Entwicklerentscheidet pro Invariante, wie sich IAGen bei einer Verletzung verhalten soll. 4. Der Enterprise-Architect-Entwickler generiert den Sourcecode, wobei die von IAGen generierten Annotationen automatisch in den Sourcecode gesetzt werden.
Erweiterungen (oder alternative Abläufe)	*a. Jederzeit, wenn Enterprise-Architect-Entwickler Multiplizitäten oder Subsets bei Assoziationen modelliert: 1. IAGen stellt die Möglichkeit zur Verfügung, dass die Annotationen nur dann generiert werden, wenn die betroffene Klasse bereits eine Markerannotation gesetzt hat. 1-3a. Enterprise-Architect-Entwickler hat keine Standard Collection für Multiplizitäten definiert. 1. Das IAGen Add-In verwendet die Collection *ArrayList<TYPE>* als Standard Container.
Spezielle Anforderungen	
Liste der Technik- und Datenvariationen	-

Häufigkeit des Auftretens	Fortlaufend
Verschiedenes	-

5.4 UC03: Invarianten Deklaration auf reversed Bytecode Modell

Umfang	IAGen
Ebene	Anwenderziel
Primäraktor	Enterprise-Architect-Entwickler
Stakeholder und Interessen	Enterprise-Architect-Entwickler • Will die modellierten Änderungen in den bestehenden Java Byte Code einfliessen lassen.
Vorbedingungen	• Bestehender Java Bytecode (.class-Dateien) wurde in den Enterprise Architect importiert. • IAGen wurde als Enterprise Architect Add-In (Extension) installiert. • Beim Exportieren der Änderungen gelten die gleichen Erweiterungen / alternativen Abläufe wie bei UC02.
Standardablauf	1. Der Enterprise-Architect-Entwickler ergänzt Diagramme mit diversen Invarianten wie Wertebereiche für Datenfelder oder Multiplizitäten bei Assoziationen. 2. Der Enterprise-Architect-Entwickler entscheidet pro Ivariante, wie sich IAGen bei einer Invariantenverletzung verhalten soll. 3. Der Enterprise-Architect-Entwickler kann die modellierten Änderungen exportieren.
Erweiterungen (oder alternative Abläufe)	-
Spezielle Anforderungen	Die exportierten Änderungen müssen zum Java Classpath des Programms hinzugefügt werden.
Liste der Technik- und Datenvariationen	-
Häufigkeit des Auftretens	Fortlaufend
Verschiedenes	-

6 Nichtfunktionale Anforderungen

6.1 Interoperabilität

6.1.1 IAGen.jar

IAGen soll als Java Library (.jar-Datei) zur Verfügung gestellt und kann deshalb in alle gängigen IDEs eingebunden werden.

6.1.2 Enterprise Architect Add-In

Das Add-In wird mit der Version 8 und 9 des Enterprise Architekt kompatibel sein.

6.2 Benutzbarkeit

Sowohl für den Java- als auch für den Enterprise-Architect-Entwickler soll die Bedienung von IAGen möglichst einfach gemacht werden. Die Entwickler sollen sich nicht um die Invariantenüberprüfung kümmern müssen. Die verfügbaren Annotationen sind selbsterklärend und können einfach und schnell definiert werden. Im Enterprise Architect wird das IAGen Add-In klein und übersichtlich gehalten.

6.3 Effizienz

Die gesetzten Annotation sollen wann immer möglich bereits zur Ladezeit überprüft werden. Dies reduziert die Notwendigkeit von Reflection, um zur Laufzeit Informationen auszulesen. Es gibt keine “Worst Case“ obere Limite betreffend Performance.

04_Architektur_und_Design/Architektur_und_Design_vX.X.pdf

6.4 Lizenzanforderungen

Es werden keine Lizenzen vergeben oder benötigt.

6.5 Technologien

- Entwickler verwendet Java 1.6 oder höher
- IDE, welche Annotation Processing unterstützt.
- Entwickler verwendet Enterprise Architect 8.0 - 9.0
- Modelliert wird mit UML 2.0
- Entwickler verwendet Eclipse Indigo

7 Einschränkungen

7.1 Modellierung von Multiplizitäten im Enterprise Architect

Der Enterprise Architect bietet unterschiedliche Möglichkeiten an, Multiplizitäten bei Assoziationen zu modellieren:

- Variante 1: Auf dem Feld

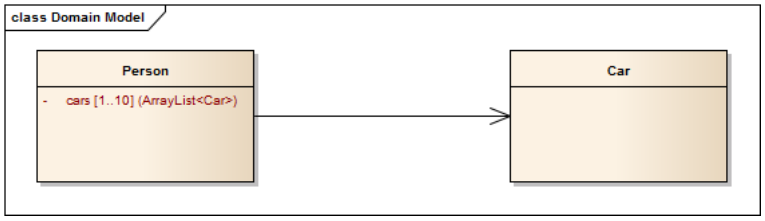


Abbildung 4: Multiplizität auf Feld

- Variante 2: Auf der Assoziation

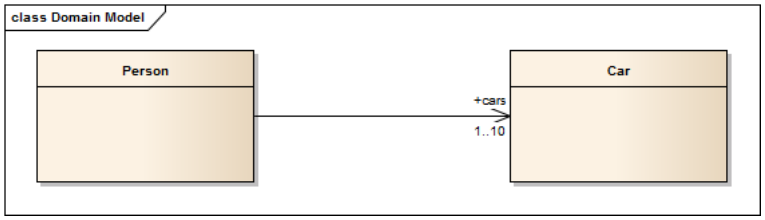


Abbildung 5: Multiplizität auf Assoziation

IAGen soll die Annotationen basierend auf der gegebenen Assoziation (Variante 2) generieren und somit die Multiplizitäten, welche direkt auf das Feld erfasst wurden, ignorieren (Variante 1).

Literaturverzeichnis

[apt] Annotation processor. http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html.

[cas] Definition Case-Tools. http://de.wikipedia.org/wiki/Computer-aided_software_engineering.

[cod] Enterprise Architect Code Engineering. http://www.sparxsystems.com/enterprise_architect_user_guide/software_development/codeengineering.html.

[ent] Enterprise Architect. http://www.sparxsystems.com/products/ea/index.html.

[ide] Definition IDE. http://de.wikipedia.org/wiki/Integrierte_Entwicklungsumgebung.

[Lar00a] Craig Larman. *Applying UML and Patterns.* Prentice Hall, 2000. Seite 61.

[Lar00b] Craig Larman. *Applying UML and Patterns.* Prentice Hall, 2000. Seite 18.

[rev] Reverse Engineering. http://de.wikipedia.org/wiki/Reverse_Engineering.

IAGen
Invariants-Assertion-Generation

Architektur und Design

13. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	06.03.2012	dmengelt	Erste Version
1.0	11.04.2012	dmengelt	Erste Architekturentscheide
2.0rc01	25.04.2012	f1egli, dmengelt	Gliederung angepasst
2.0rc02	08.05.2012	dmengelt	Deployment-Sicht
2.0rc03	11.05.2012	f1egli, dmengelt	Anpassungen Gliederung
2.0rc04	15.05.2012	dmengelt	Grobübersicht Packages
2.0rc05	15.05.2012	f1egli	Annotation und AspectJ
2.0	17.05.2012	dmengelt	Version 2.0
3.0rc01	18.05.2012	dmengelt	Anpassungen gemäss Wochenbesprechung
3.0rc02	21.05.2012	dmengelt	Multiplicity und Subsets Finder
3.0rc03	28.05.2012	dmengelt	Übersicht und Packages
3.0rc04	29.05.2012	dmengelt	Einschränkungen
3.0rc05	30.05.2012	f1egli	Designmodell und Beispielablauf
3.0rc06	04.06.2012	f1egli	Aspects, Annotation Handlers
3.0rc07	04.06.2012	f1egli	Type Handlers
3.0rc08	05.06.2012	f1egli	Testing
3.0rc09	05.06.2012	dmengelt	Verworfene Ideen
3.0rc10	05.06.2012	f1egli	Einschränkungen
3.0rc11	07.06.2012	f1egli, dmengelt	Packages Grobübersicht angepasst
3.0rc12	08.06.2012	f1egli	Annotation Processing
3.0	09.06.2012	f1egli, dmengelt	Finale Version 3.0

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Architektur

3.1 Einführung & Übersicht

IAGen besteht aus zwei Teilen. Einem Add-In für den Enterprise Architect und einer Java Library, welche direkt in eine IDE eingebunden werden kann:

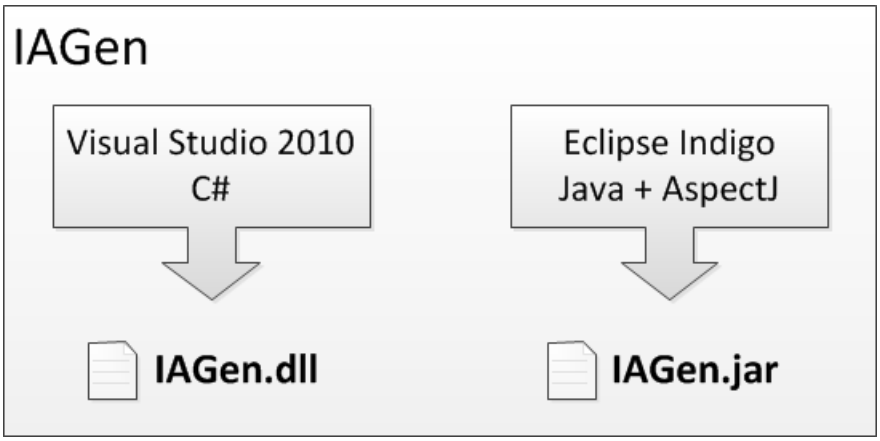


Abbildung 1: Teile von IAGen

Mit dem Add-In für den Enterprise Architect können die modellierten IAGen-Invarianten als Annotationen generiert und anschliessend mittels Forward Engineering in den Java Source Code generiert werden. Vorhandene Annotationen im Sourcecode können mittels Reverse Engineering in den Enterprise Architect importiert werden. Zusätzlich kann das Add-In die Annotationen auch als XML-Datei (aop.xml) exportieren. Dies wird dann notwendig, wenn der bestehende Programmcode nur als Byte Code vorliegt. Die IAGen Library (IAGen.jar) überprüft dann zur Laufzeit die definierten IAGen-Invarianten, welche als Annotationen gesetzt wurden.

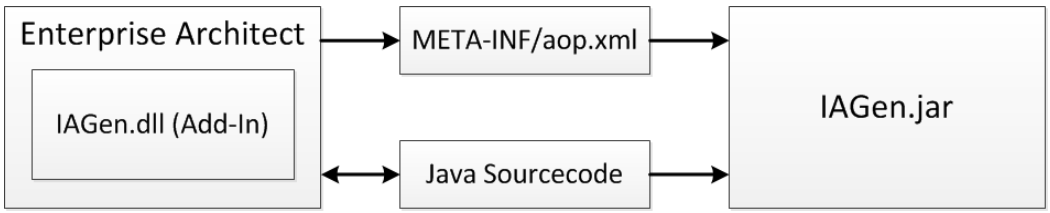


Abbildung 2: IAGen-Übersicht

3.2 Logische Sicht der Implementation - IAGen.jar

Beschreibt den architektonischen Grobaufbau der IAGen Library. Die wichtigsten Packages werden hier beschrieben.

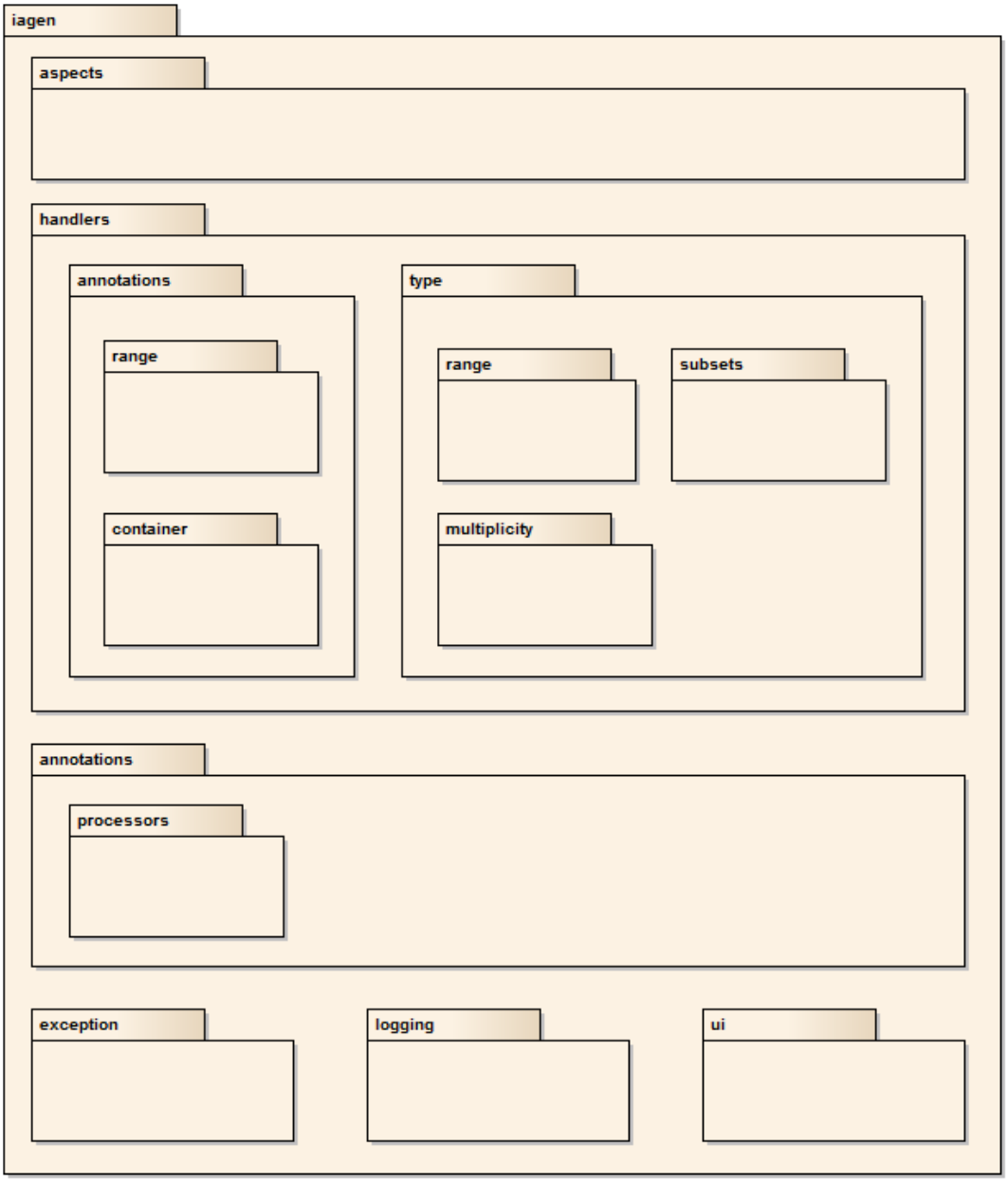


Abbildung 3: Wichtigste Packages der Implementation

Die drei wichtigsten Packages von IAGen sind *aspects*, *handlers* und *annotations*. Sie werden nachfolgend genauer beschrieben. Die packages *exception*, *logging* und *ui* sind für die Reaktionen bei allfälligen Invariantenverletzungen zuständig.

3.2.1 Package aspects

Im *aspects* Package befinden Sich die einzelnen .aj-Dateien (AspectJ). Der *FieldAspect* ist zuständig für alle IAGen-Annotationen, welche auf ein Feld vorgenommen wurden. Der *ParameterAspect* ist für IAGen-Annotationen verantwortlich, welche auf Methoden- oder Konstruktorparameter gesetzt wurden. Für die Annotation *@Const* wurden wegen der Übersichtlichkeit nochmals drei neue Aspects mit dem gleichen Prinzip eingeführt.

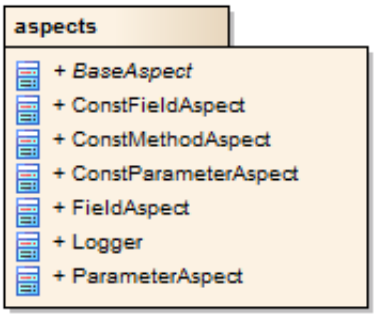


Abbildung 4: IAGen Aspects

3.2.2 Package handlers

Das Package *handlers* beinhaltet die Subpackages *annotations* und *type*. Das Package *annotations* wird von den Aspects verwendet, um die richtige Annotation verarbeiten zu können. Für die Invariantenüberprüfung benötigt *annotations* das Package *type*.

Abbildung 5: IAGen Aspects

3.2.3 Package annotations

Das Package *annotations* beinhaltet die IAGen Annotationen und das Package *processors*, in welchem sich die Logik für das Annotation Processing befindet.

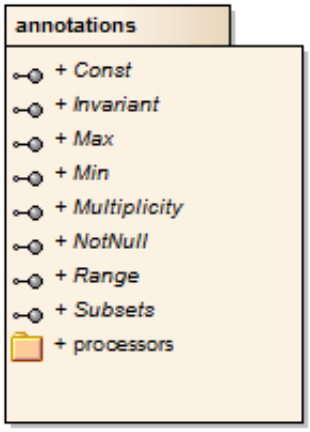


Abbildung 6: IAGen Annotationen

3.3 Logische Sicht der Tests

Beim Testing gibt es die zwei Hauptpackages *cases* und *resources*. Das *cases* Package beinhaltet die Testklassen, welche die internen IAGen-Logiken testen. Im Package *resources* befinden sich "Dummy" Klassen, die mit IAGen-Annotationen annotiert sind. Beim Ausführen der JUnit Tests verwenden die Testklassen direkt die annotierten "Dummy" Klassen.

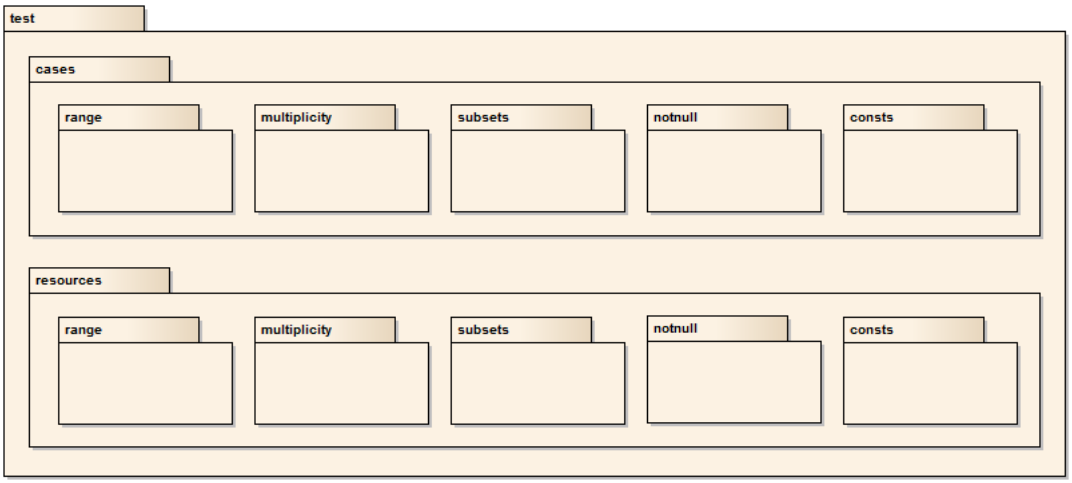


Abbildung 7: Wichtigste Packages des Testings

3.4 Logische Sicht der Implementation - IAGen.dll

Das IAGen Enterprise Architect Add-In (IAGen.dll) besteht lediglich aus dem Namespace IAGen. Sämtliche Klassen des Add-Ins befinden sich in diesem Namespace.

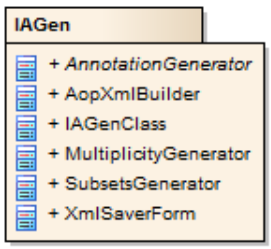


Abbildung 8: Klassen des IAGen Add-In

3.5 Deployment-Sicht

3.5.1 Enterprise Architect Add-In - IAGen.dll

Das IAGen Add-In ist eine DLL, welche beim Starten vom Enterprise Architect gelesen wird. Die Enterprise Architect Add-In Architektur [eaa] ist wie folgt aufgebaut:

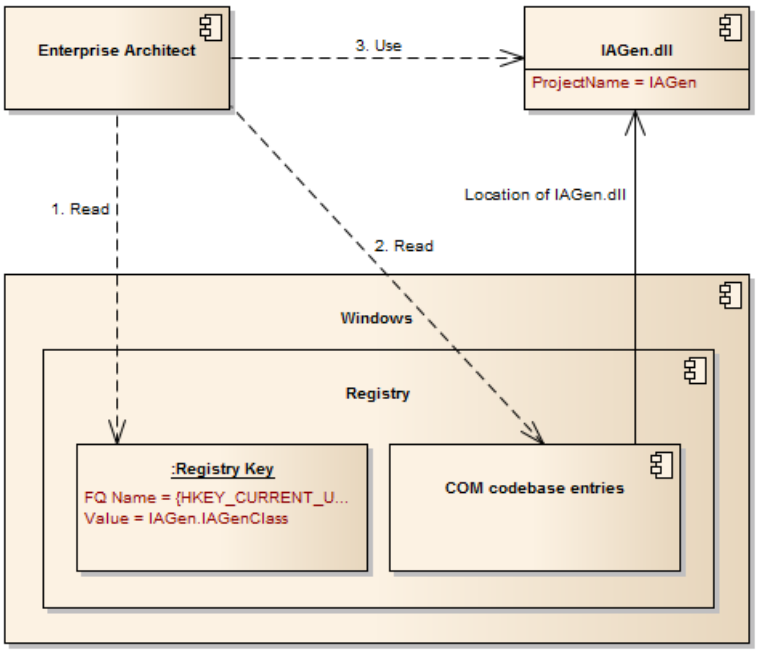


Abbildung 9: Enterprise Archictect Add-In Architektur

1. Der Enterprise Architect liest den vorhandenen Windows Registry Key aus. Der Key wird mit der Installation des IAGen Add-Ins automatisch erzeugt.
2. Anschliessend erfragt der Enterprise Architect den Namen des Assembly. Diese Information ist in der COM Codebase [coma] gespeichert.

3. Schlussendlich verwendet der Enterprise Architect die *public*-Methoden in der Add-In Klasse.

3.5.2 IAGen Library - IAGen.jar

Die IAGen Library, eingesetzt in einer Tomcat Webprojekt (WebApp.war) Umgebung.

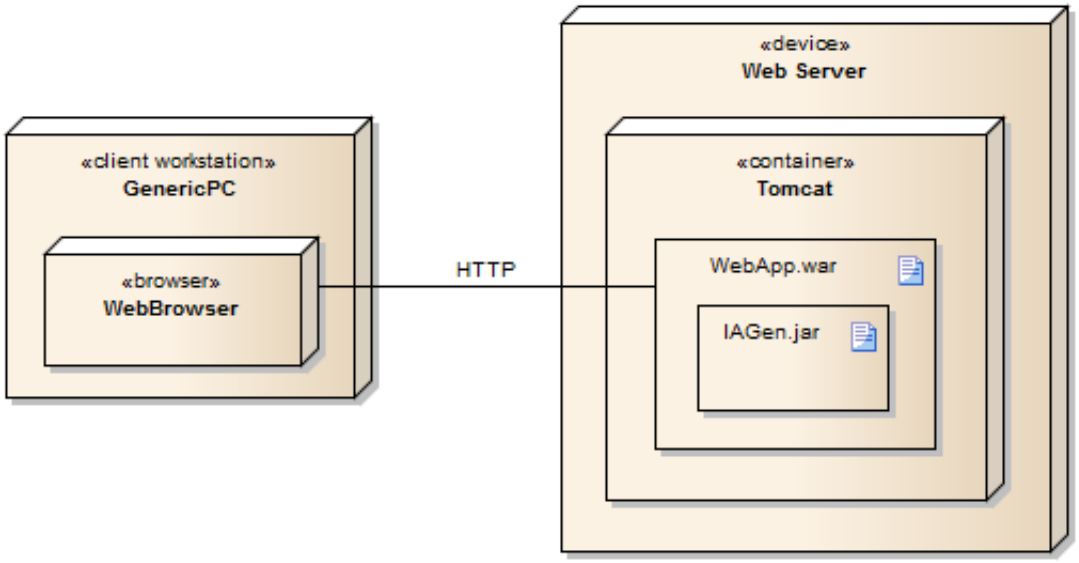


Abbildung 10: Mögliche Verwendung der IAGen Library

Die Webapplikation bindet die IAGen Library als externe .jar-Datei ein und hat dann Zugriff auf die IAGen-Annotationen.

3.6 Prozess Sicht

IAGen verwendet keine Nebenläufigkeiten in Form von Threads oder Prozessen.

3.7 Technologiewahl

Für die Entwicklung der IAGen Library und des Enterprise Architect Add-Ins werden folgenden Technologien verwendet, welche nicht bereits in den Anforderungen festgehalten sind:

3.7.1 IAGen Library

- AspectJ-DEVELOPMENT-20120411173900
  Die Pointcutsignaturen von AspectJ lassen sich ideal mit Annotationen verknüpfen. Beispielsweise kann herausgefunden werden, ob eine Methode einen Parameter enthält, der mit einer bestimmten Annotation versehen ist. Aus diesem Grund verwendet IAGen AspectJ.
- Java Annotation Processing (APT)

3.7.2 Enterprise Architect Add-In

- Visual Studio 2010 (.Net 4.0)
  Damit dem Enterprise Architect Funktionalitäten in Form eines Add-Ins hinzugefügt werden können, muss entweder Borland Delphi, Microsoft Visual Basic oder Microsoft Visual Studio .Net eingesetzt werden. Für das IAGen Add-In haben wir uns für .Net entschieden.

3.7.3 UML-Modell

Mit dem Enterprise Architect wird ein UML-Modell geführt, welches mit den Programmsourcen synchron ist. Informationen zu anderen eingesetzten Tools können dem Dokument “Tools und Infrastruktur“ entnommen werden:

`04_Architektur_und_Design/02_Tools_und_Infrastruktur/Tools_und_Infrastruktur_VX.X.pdf`

4 Design

4.1 Designmodell

Das Designmodell der IAGen Library. Das Designmodell zeigt die wichtigsten Klassen und deren Zugehörigkeiten.

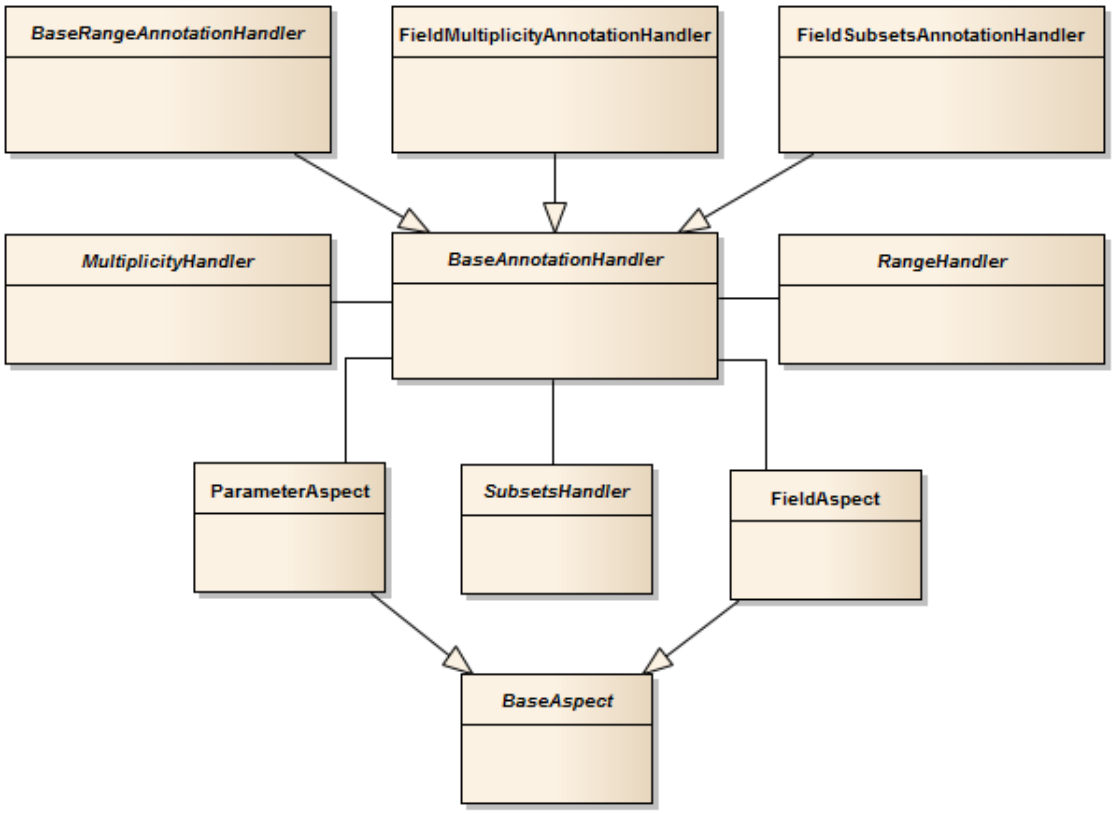


Abbildung 11: Highlevel Ansicht IAGen

Es zeigt, wie die verschiedenen Aspects die richtigen Annotation Handlers auswählen (ausgehend von der Annotation) und diese dann den jeweiligen Typ Handler ansprechen (ausgehend vom annotierten Typ). Das genaue Zusammenspiel der verschiedenen Klassen wird in den folgenden Kapiteln erklärt.

4.2 Beispielablauf

Um ein Verständnis der klassenübergreifenden Abfolgen zu erhalten, wird folgendes Beispiel gegeben. Im Java Code des Entwicklers gibt es die Klasse Person:

Listing 1: Multiplicity Annotation auf einem Feld

```
1  @Invariant
2  public class Person {
3
4      @Multiplicity(lower=0, upper=100)
5      List<Car> cars;
6
7      public Person() {
8          cars = new ArrayList<Car>();
9      }
10 }
```

Seitens IAGen entsteht folgendes Sequenzdiagramm:

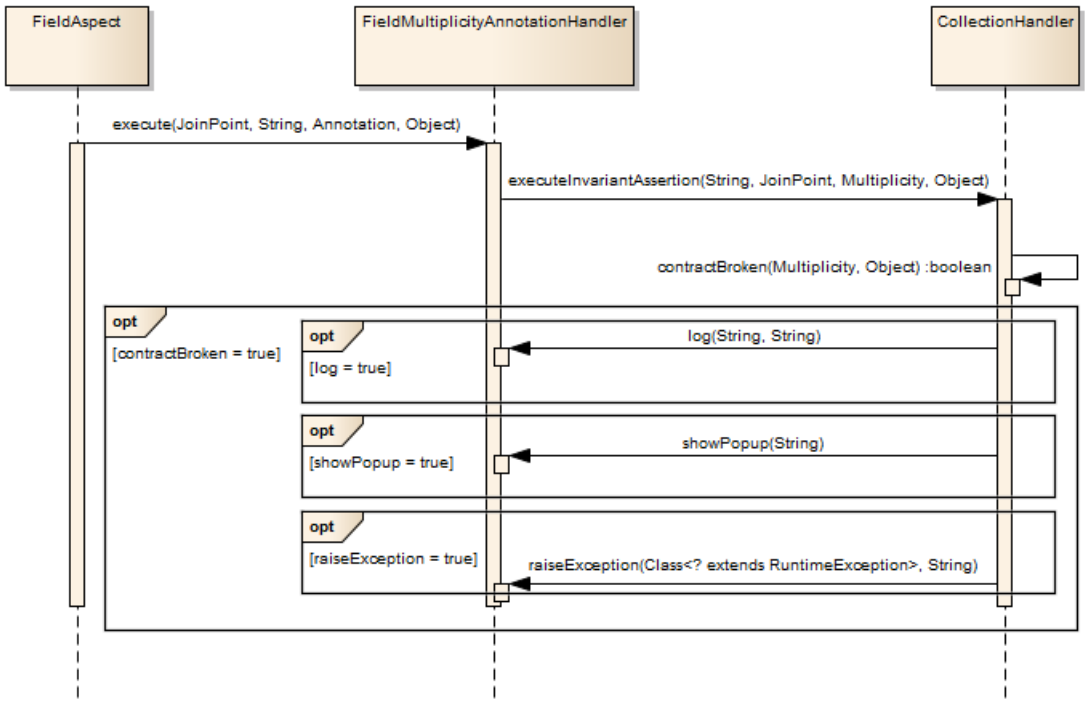


Abbildung 12: Sequenzdiagramm des Beispielablaufs

FieldAspect erkennt nach dem Konstruktoraufruf von Person das annotierte Feld cars. Da es eine *@Multiplicity*-Annotation ist, wird der Annotation Handler FieldMultiplicityAnnotationHandler aufgerufen. Dieser überüft den Typ von cars, was in unserem Fall einer Collection entspricht. Aus diesem Grund wird der CollectionHandler angesprochen, welcher von MultiplicityHandler erbt. Der CollectionHandler überprüft, ob die Invariante verletzt ist.

- Bei einer Verletzung werden die gesetzten Einstellungen der Annotation ausgewertet und dem FieldMultiplicityAnnotationHandler gemeldet. Dieser reagiert dann

entsprechend, beispielsweise im Werfen einer Exception.

- Wenn keine Verletzung vorliegt, werden keine weiteren Massnahmen getroffen (mit Ausnahme der debug-Einstellungsmöglichkeiten der IAGen-Annotationen, welche in jedem Fall eine Konsolenmeldung ausgibt).

4.3 Annotations

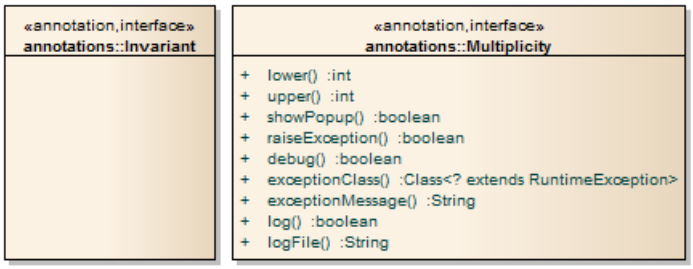


Abbildung 13: Die Annotationen @Invariant und @Multiplicity

In IAGen gibt es zwei verschiedene Annotationstypen:

- Die Marker Annotierungen.
- Die Annotationen für die Invariantendefinitionen.

Die Marker Annotierungen besitzen keinerlei Methoden. Eine solche Annotierung ist *@Invariant*. Sie dient lediglich der Markierung im Code.

Entscheid: Die *@Invariant* Annotation werden bei den Pointcuts gebraucht. Durch die Überprüfung des Vorhandenseins der Annotation muss man sich nicht bei allen Konstruktoren, Methodenaufrufe und Sets (das Setzen eines Feldes) einweben, was eine erhebliche Performancesteigerung mit sich bringt.

Annotationen wie *@Multiplicity* haben dagegen mehrere Methoden, welche mit Defaults vordefiniert werden können. Im Code können sie dann so gesetzt werden:

```
@Multiplicity(lower=0, upper=100, raiseException=false, showPopup=true)
List<Car> cars;
```

Entscheid: Die Methoden:

- boolean showPopup() default false;
- boolean raiseException() default true;
- boolean debug() default false;
- Class<? extends RuntimeException> exceptionClass() default InvariantBrokenException.class;

- String exceptionMessage() default “@Min contract broken!“;
- boolean log() default false;
- String logFile() default “log.txt“;

sind bei allen Invariantenannotationen gegeben und vordefiniert (mit Ausnahme von *@Const*). Da sich Annotationen nicht vererben lassen [ann], mussten diese Methoden bei den genannten Annotationen einzeln gesetzt werden und es konnte keine gemeinsame Basisklasse erstellt werden.

4.4 Aspects

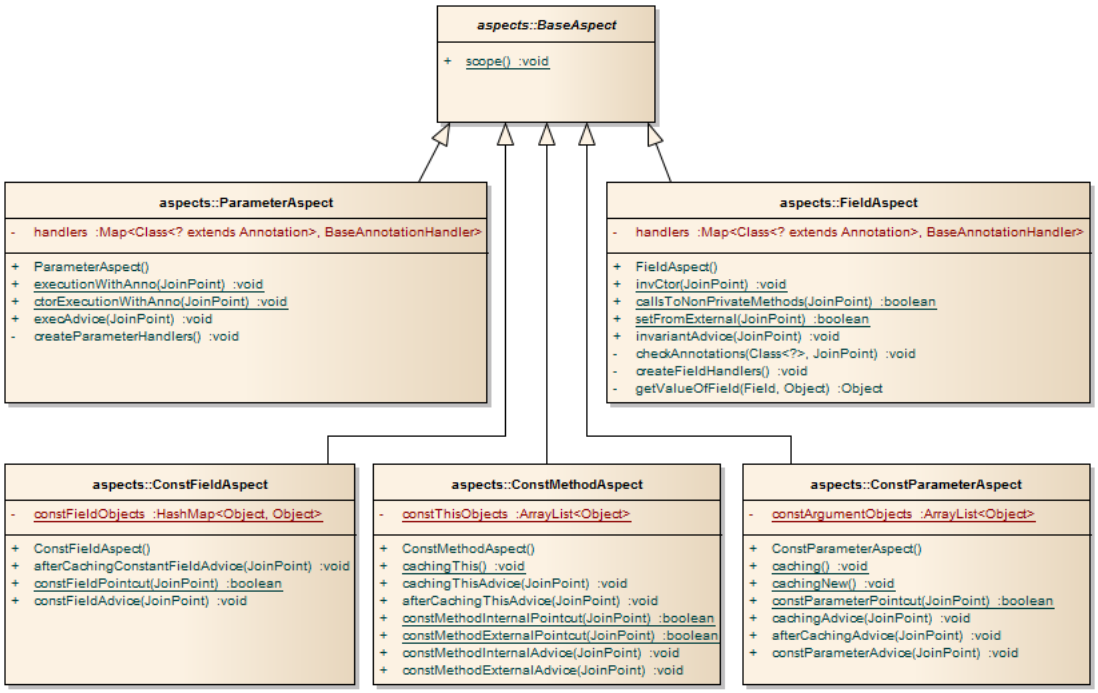


Abbildung 14: IAGen Aspects

4.4.1 BaseAspect

Durch den *scope* Pointcut im BaseAspect lassen sich rekursive Aufrufe in den Aspects verhindern. Ausserdem soll IAGen selber nicht auf IAGen-Annotationen reagieren.

Listing 2: Gültigkeitsbereich für Pointcuts

```
1 @Pointcut("!within(ch.iagen.aspects..*) && !within(ch.iagen.annotations..*)
      && !within(ch.iagen.handlers..*) && !within(ch.iagen.exception..*) &&
     !within(ch.iagen.logging..*) && !within(ch.iagen.ui..*)")
2 public static void scope() {
3 }
```

Dieser Aspect dient also lediglich der Einschränkung auf Invariantenüberprüfungen seitens IAGen. Alle konkreten Aspects erben von BaseAspect.

4.4.2 FieldAspect

Für die IAGen Field Annotationen *@Min*, *@Max*, *@Range*, *@Multiplicity*, *@Subsets* und *@NotNull* ist der Aspect *FieldAspect* zuständig. Es gibt drei Pointcuts in diesem Aspect, welche folgend erklärt werden:

4.4.2.1 Konstruktor

Die Invarianten müssen nach dem Erstellen eines Objektes überprüft werden. Dafür wurde der *invCtor(...)* Pointcut erstellt:

Listing 3: Pointcut für Konstruktoren

```
1 @Pointcut("initialization((@ch.iagen.annotations.Invariant *).new(..)) &&
      scope()")
2 public static void invCtor(JoinPoint jp) {
3 }
```

Sobald eine Initialisation eines Objektes stattfindet und diese Klasse auch die *@Invariant*-Annotation besitzt, greift dieser Pointcut.

4.4.2.2 Method calls

Wenn Methoden von aussen aufgerufen werden und das Ziel mit *@Invariant* versehen ist, so müssen die darin gesetzten Invarianten überprüft werden. Folgender Pointcut reagiert auf solche externen Calls:

Listing 4: Pointcut für externe Methodenaufrufe

```
1 @Pointcut("if() && call(!private * (@ch.iagen.annotations.Invariant *)
      .*(..)) && @target(ch.iagen.annotations.Invariant) && scope()")
2 public static boolean callsToNonPrivateMethods(JoinPoint jp) {
3    return jp.getThis() != jp.getTarget();
4 }
```

- *!private*, da interne Calls nicht überprüft werden müssen.
- *@target(ch.iagen.annotations.Invariant)* bedeutet, dass das Ziel initialisiert sein muss. Ein externer Aufruf auf eine *static*-Methode würde also nicht reagieren.
- *jp.getThis() != jp.getTarget()* ist ein Laufzeittest. Es können auch interne calls auf nicht private Methoden ausgeführt werden. Diese müssen nicht überprüft werden. Bei einem internen Aufruf ist *jp.getThis() != jp.getTarget() false* und der Pointcut reagiert nicht.

4.4.2.3 Set

Nicht private Felder können auch direkt von extern gesetzt werden. Dabei muss eine Invariantenüberprüfung stattfinden, sofern das Ziel mit *@Invariant* annotiert ist:

Listing 5: Pointcut für das Setzen von Feldern

```
1 @Pointcut("if() && set(!private * (@ch.iagen.annotations.Invariant *).*)
      && @target(ch.iagen.annotations.Invariant) && scope()")
2    public static boolean setFromExternal(JoinPoint jp) {
3       return jp.getThis() != jp.getTarget();
4    }
```

- *!private*, da intern gesetzte Felder nicht überprüft werden müssen.
- *@target(ch.iagen.annotations.Invariant)* bedeutet, dass das Ziel initialisiert sein muss. Ein externer Aufruf auf ein *static*-Feld würde also nicht reagieren.
- *jp.getThis() != jp.getTarget()* ist ein Laufzeittest. Es können auch interne Felder, welche nicht private sind, von der eigenen Klasse gesetzt werden. Diese müssen nicht überprüft werden. Bei einem internen Setzen eines Feldes ist *jp.getThis() != jp.getTarget() false* und der Pointcut reagiert nicht.

4.4.2.4 Advice

Alle drei Pointcuts sind mit dem gleichen Advice verknüpft:

Listing 6: After Advice im FieldAspect

```
1 @After("invCtor(jp) || callsToNonPrivateMethods(jp) || setFromExternal(jp)"
      )
2 public void invariantAdvice(JoinPoint jp) {
3    checkAnnotations(jp.getTarget().getClass(), jp);
4 }
```

Die Methode *checkAnnotations(...)* überprüft, welche Field-Annotationen gesetzt sind.

Listing 7: checkAnnotations()

```
1  private void checkAnnotations(Class<?> target, JoinPoint jp) {
2     if(target.getSuperclass() != null && target.getSuperclass().
          isAnnotationPresent(Invariant.class)){
3        checkAnnotations(target.getSuperclass(), jp);
4     }
5
6     if (target.getDeclaredFields() != null) {
7        for (Field f : target.getDeclaredFields()) {
8           for (Annotation a : f.getAnnotations()) {
9              if(handlers.get(a.annotationType()) != null){
10                handlers.get(a.annotationType()).execute(jp, f.getName(), a,
                       getValueOfField(f, jp.getTarget()));
11             }
12          }
13       }
14    }
15 }
```

Im ersten Abschnitt der Methode wird kontrolliert, ob Oberklassen überprüft werden müssen. Wenn eine Oberklasse mit *@Invariant* annotiert ist, wird die Methode rekursiv aufgerufen. Der richtige Handler zur Annotation wird über die *handlers* Map herausgefunden und die entsprechende *execute(...)*-Methode wird auf dem Handler aufgerufen. Die *handlers* Map wird beim Konstruktor des *FieldAspects* über die Methode *createFieldHandlers()* erstellt:

Listing 8: Erstellen der handlers Map

```
1 private void createFieldHandlers() {
2     handlers = new HashMap<Class<? extends Annotation>,
          BaseAnnotationHandler>();
3     handlers.put(Multiplicity.class, new FieldMultiplicityAnnotationHandler
          ());
4     handlers.put(Min.class, new MinAnnotationHandler());
5     handlers.put(Max.class, new MaxAnnotationHandler());
6     handlers.put(Range.class, new RangeAnnotationHandler());
7     handlers.put(Subsets.class, new FieldSubsetsAnnotationHandler());
8     handlers.put(NotNull.class, new NotNullAnnotationHandler());
9 }
```

4.4.3 ParameterAspect

Die Annotationen *@Min*, *@Max*, *@Range* und *@NotNull* können auch auf Parameter in Methoden und Konstruktoren gesetzt werden. Für die Parameterannotationen ist der *ParameterAspect* zuständig.

Entscheid: Parameterannotationen werden unmittelbar beim Entritt in eine Methode oder bei einem Konstruktoraufruf überprüft (Pre-Condition).

Es gibt zwei Pointcuts in diesem Aspect, welche folgend erklärt werden:

4.4.3.1 Konstruktor

Sobald ein Konstruktor, welcher eine Parameterannotation besitzt, ausgeführt wird und die Klasse dieses Konstruktors mit *@Invariant* annotiert ist, muss die Parameterannotation überprüft werden:

Listing 9: Pointcut für Konstruktoren

```
1 @Pointcut("execution((@ch.iagen.annotations.Invariant *).new(..,@ch.iagen.
      annotations.Min (*) || @ch.iagen.annotations.Max (*) || @ch.iagen.
      annotations.Range (*) || @ch.iagen.annotations.NotNull (*),..)) &&
      scope()")
2 public static void ctorExecutionWithAnno(JoinPoint jp) {
3 }
```

4.4.3.2 Methoden

Sobald ein Methode, welche eine Parameterannotation besitzt, ausgeführt wird und die Klasse dieser Methode mit *@Invariant* annotiert ist, muss die Parameterannotation überprüft werden:

Listing 10: Pointcut für Methoden

```
1 @Pointcut("execution(* (@ch.iagen.annotations.Invariant *).*(..,@ch.iagen.
      annotations.Min (*) || @ch.iagen.annotations.Max (*) || @ch.iagen.
      annotations.Range (*) || @ch.iagen.annotations.NotNull (*),..)) &&
      scope()")
2 public static void executionWithAnno(JoinPoint jp) {
3 }
```

4.4.3.3 Advice

Beide Aspects sind mit dem gleichen Advice verknüpft:

Listing 11: Before Advice im ParameterAspect

```
1  @Before("executionWithAnno(jp) || ctorExecutionWithAnno(jp)")
2  public void execAdvice(JoinPoint jp) {
3      Object[] args = jp.getArgs();
4      Annotation[][] parameterAnnotations = null;
5      String[] paramNames = null;
6
7      if (jp.getSignature() instanceof MethodSignature) {
8          MethodSignature ms = (MethodSignature) jp.getSignature();
9          paramNames = ms.getParameterNames();
10         parameterAnnotations = ms.getMethod().getParameterAnnotations();
11     } else if (jp.getSignature() instanceof ConstructorSignature) {
12         ConstructorSignature cs = (ConstructorSignature) jp.getSignature();
13         paramNames = cs.getParameterNames();
14         parameterAnnotations = ((Constructor<?>) cs.getConstructor()).
               getParameterAnnotations();
15     }
16
17     for (int i = 0; i < parameterAnnotations.length; i++) {
18         Annotation[] annotations = parameterAnnotations[i];
19         for (Annotation annotation : annotations) {
20             handlers.get(annotation.annotationType()).execute(jp, paramNames[i
                   ], annotation,
21                     args[i]);
22         }
23     }
24 }
```

Der Advice übeprüft, ob es sich um eine Konstruktor- oder Methodensignatur handelt, und holt die entsprechenden Informationen. Der richtige Handler zur Annotation wird über die *handlers* Map herausgefunden und die entsprechende *execute(...)*-Methode wird auf dem Handler aufgerufen. Die *handlers* Map wird beim Konstruktor des *ParameterAspects* über die Methode *createParameterHandlers()* erstellt:

Listing 12: Erstellen der handlers Map

```
1 private void createParameterHandlers() {
2     handlers = new HashMap<Class<? extends Annotation>,
          BaseAnnotationHandler>();
3     handlers.put(Min.class, new MinAnnotationHandler());
4     handlers.put(Max.class, new MaxAnnotationHandler());
5     handlers.put(Range.class, new RangeAnnotationHandler());
6     handlers.put(NotNull.class, new NotNullAnnotationHandler());
7 }
```

4.4.4 ConstAspects

Die Annotation *@Const* kann auf Feldern, Methoden und Parametern gesetzt werden. Für jeden Ort gibt es einen entsprechenden Aspect:

4.4.4.1 ConstParameterAspect

Im ConstParameterAspect gibt es zwei Schritte:

- Caching des annotierten Parameters.
- Auswertung innerhalb der Methode oder des Konstruktors.

Das annotierte Parameter Objekt muss zuerst gecached werden. Dies geschieht über die Pointcuts *caching()* und *chachingNew()*:

Listing 13: Pointcut für Caching

```
1 @Pointcut("execution(* (@ch.iagen.annotations.Invariant *).*(..,@ch.iagen.
      annotations.Const (*),..)) && scope()")
2     public static void caching() {
3     }
4
5 @Pointcut("execution((@ch.iagen.annotations.Invariant *).new(..,@ch.iagen.
      annotations.Const (*),..)) && scope()")
6     public static void cachingNew() {
7     }
```

Im Before Advice zu diesen Pointcuts werden die Objekte in einer Liste gespeichert und im entsprechenden After Advice wieder gelöscht.

Im zweiten Schritt werden alle calls und sets, welche die Methode oder der Konstruktor mit dem annotierten Parameter auslösen, ausgewertet. Wird auf dem Parameter Objekt eine nicht konstante Methode oder ein set ausgeführt, wird der Advice *constParameterAdvice(...)* ausgeführt, welcher dann eine entsprechende Exception wirft:

Listing 14: Werfen der Exception

```
1 @Before("constParameterPointcut(jp)")
2 public void constParameterAdvice(JoinPoint jp) {
3     if (jp.getKind() == JoinPoint.FIELD_SET) {
4         throw new InvariantBrokenException("Changed a field of a const
              parameter: " + jp);
5     } else {
6         throw new InvariantBrokenException("Called a non-const method of a
              const parameter: " + jp);
7     }
8 }
```

4.4.4.2 ConstFieldAspect

Im ConstFieldAspect gibt es zwei Schritte:

- Caching von *this* und dem konstanten Feld.
- Auswertung aller calls und sets auf das konstante Feld Objekt.

Das annotierte Feld wird gecached, sobald der Konstruktor der Klasse abgearbeitet ist. Im einem Before Advice wird das annotierte Feld und *this* in einer Map gespeichert. Im entsprechenden After Advice werden diese Informationen wieder aus der Map gelöscht.

Im zweiten Schritt werden alle calls und sets auf das annotierte Feld ausgewertet. Wenn der Aufrufer *this* ist und ein call auf eine nicht konstante Methode vom annotierten Feld aufgerufen wird oder wenn von *this* ein set auf das annotierte Feld ausgeführt wird, wirft der *constFieldAdvice(...)* eine entsprechende Exception:

Listing 15: Werfen der Exception

```
1 @After("constFieldPointcut(jp)")
2 public void constFieldAdvice(JoinPoint jp) {
3     if (jp.getKind() == JoinPoint.FIELD_SET) {
4         throw new InvariantBrokenException("Changed a value of a constant
              field: " + jp);
5     } else {
6         throw new InvariantBrokenException("Called a non-const method of a
              const field: " + jp);
7     }
8 }
```

4.4.4.3 ConstMethodAspect

Im ConstMethodAspect gibt es drei Schritte:

- Caching von *this*, welche die konstante Methode enthält.
- Auswertung aller internen calls und internen sets, welche von der konstanten Methode ausgelöst werden.
- Auswertung aller calls mit einem Parameterargument *this*, welche von der konstanten Methode ausgelöst werden.

This muss gecached werden, sobal eine Methode mit *@Const* annotiert ist:

Listing 16: Pointcut für Caching

```
1 @Pointcut("execution(@ch.iagen.annotations.Const * (@ch.iagen.annotations.
      Invariant *).*(..)) && scope()")
2 public static void cachingThis() {
3 }
```

Im Before Advice zu diesem Pointcut wird *this* in einer Liste gespeichert und im entsprechenden After Advice wieder gelöscht.

Im zweiten Schritt werden alle internen calls und sets, welche die konstante Methode auslöst, ausgewertet. Wenn ein interner call auf eine nicht konstante Methode oder eine set ausgeführt wird, wirft *constMethodInternalAdvice(...)* eine entsprechende Exception:

Listing 17: Werfen der Exception

```
1 @After("constMethodInternalPointcut(jp)")
2 public void constMethodInternalAdvice(JoinPoint jp) {
3    if (jp.getKind() == JoinPoint.FIELD_SET) {
4       throw new InvariantBrokenException("Changed 'this' object within a
            const method: " + jp);
5    } else {
6       throw new InvariantBrokenException("Called a non-const method from
            within a const method: " + jp);
7    }
8 }
```

Im dritten Schritt werden alle calls (Methoden und Konstruktoren) mit einem Argument *this* ausgewertet, die von einer konstanten Methode ausgelöst wurden. Wenn die Methode oder der Konstruktor das *this* Argument nicht mit *@Const* gekennzeichnet hat, wird der Advice *constMethodExternalAdvice(...)* ausgeführt, welcher dann eine entsprechende Exception wirft:

Listing 18: Werfen der Exception

```
1  @After("constMethodExternalPointcut(jp)")
2  public void constMethodExternalAdvice(JoinPoint jp) {
3      if (jp.getKind() == JoinPoint.METHOD_CALL) {
4          throw new InvariantBrokenException(
5                  "Called a method with an argument 'this' from a const method
                       but in the signature of the called method is no const
                       parameter present. "
6                      + jp);
7
8      } else {
9          throw new InvariantBrokenException(
10                 "Called a ctor with an argument 'this' from a const method but
                       in the signature of the called ctor is no const parameter
                       present. "
11                     + jp);
12     }
13 }
```

4.5 Annotation Handlers

Alle Annotation Handlers erben von *BaseRangeAnnotationHandler*, welcher das Interface *IReaction* implementiert. Darin befindet sich die Logik für Reaktionen bei einer Invariantenverletzung:

- Werfen von Exceptions - *raiseException(Class<? extends RuntimeException> clazz, String message).*
- Anzeigen von Popups - *showPopup(String message).*
- Logging in eine Datei - *log(String message, String logFile).*

4.5.1 Handlers für @Min, @Max und @Range

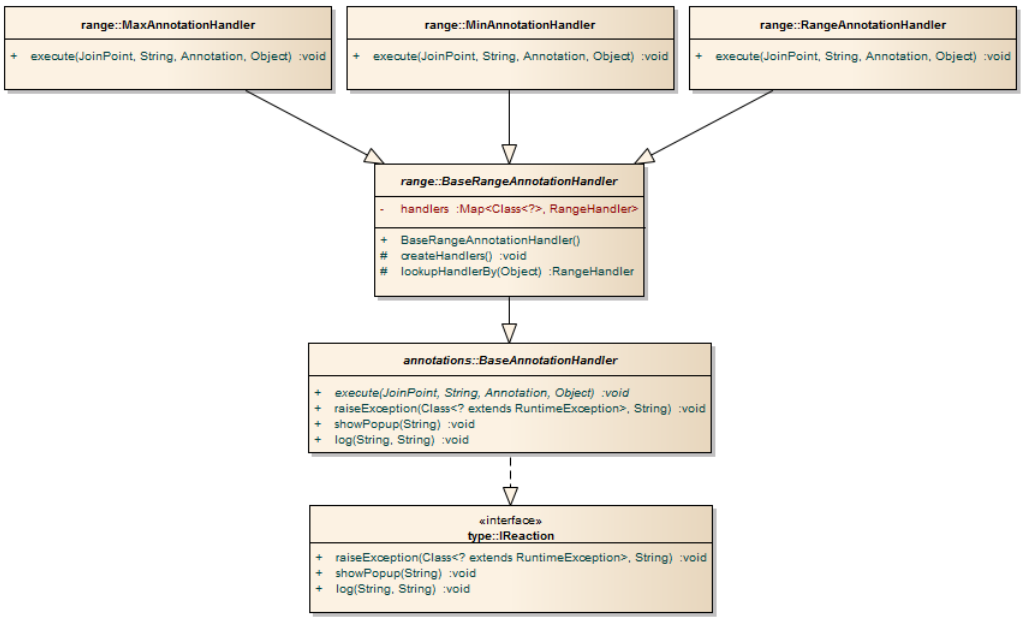


Abbildung 15: IAGen Range Handlers

Wird von einem der Aspects (siehe Kapitel 4.4) eine *@Min*, *@Max* oder *@Range*-Annotation erkannt, wird die *execute(...)*-Methode des entsprechenden Handlers (*MinAnnotationHandler*, *MaxAnnotationHandler*, *RangeAnnotationHandler*) aufgerufen.
In der Klasse *BaseRangeAnnotationHandler* gibt es eine handlers Map, welche für den annotierten Typ (Double, Float, Integer etc.) zuständig ist. Die *execute(...)*-Methode ruft den richtigen Typ Handler auf, damit die Invariante überprüft werden kann. Siehe Kapitel 4.6 für weitere Ausführungen der Typ Handlers.

Die Methode *createHandlers()* in *BaseRangeAnnotationHandler* wird beim Konstruktor aufgerufen, um die Map mit den unterstützten Typen zu erstellen:

Listing 19: Erstellung der Map

```
protected void createHandlers() {
    handlers = new HashMap<Class<?>, RangeHandler>();
    handlers.put(Byte.class, new ByteHandler(this));
    handlers.put(Short.class, new ShortHandler(this));
    handlers.put(Integer.class, new IntegerHandler(this));
    handlers.put(Long.class, new LongHandler(this));
    handlers.put(Float.class, new FloatHandler(this));
    handlers.put(Double.class, new DoubleHandler(this));
    handlers.put(String.class, new StringHandler(this));
    handlers.put(Character.class, new CharacterHandler(this));
    handlers.put(null, new NullHandler(this));
}
```

4.5.2 Handlers für @Multiplicity, @Subsets und @NotNull

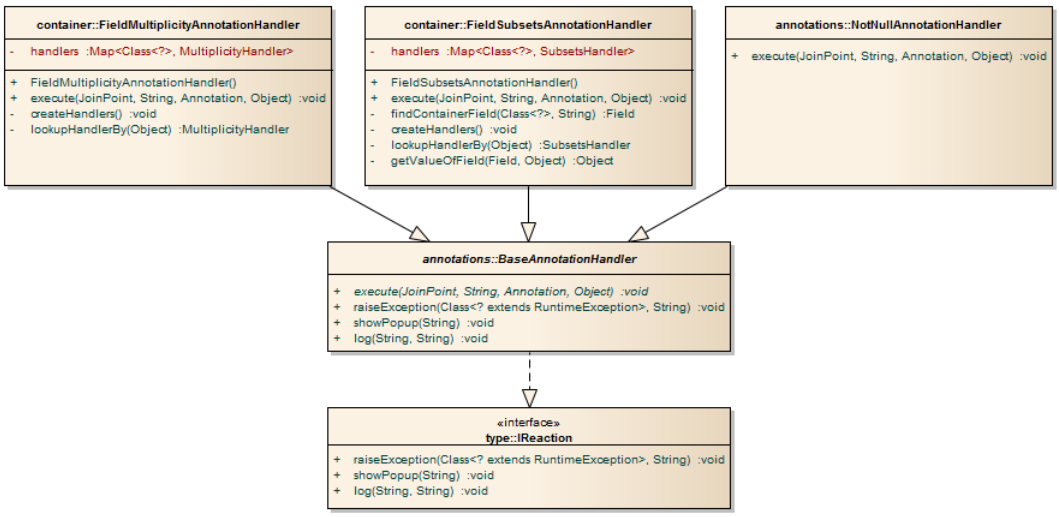


Abbildung 16: IAGen Container Handlers

Wird vom *FieldAspect* (siehe Kapitel 4.4) eine *@Subsets*- oder eine *@Multiplicity*-Annotation erkannt, wird die *execute(...)*-Methode des entsprechenden Handlers (*FieldSubsetsAnnotationHandler*, *FieldMultiplicityAnnotationHandler*) aufgerufen.

Beide Handlers haben eine handlers Map, damit die Invariantenüberpüfung für den richtigen Typ(Collection, Map, Array [ ], Null) stattfinden kann. Siehe Kapitel 4.6 für weitere Ausführungen der Typ Handlers.

Wird von einem der Aspects (siehe Kapitel 4.4) eine *@NotNull*-Annotation erkannt, wird die *execute(...)*-Methode des *NotNullAnnotationHandler* aufgerufen. Diese Methode überprüft, ob die Invariante verletzt ist oder nicht, und reagiert entsprechend den gesetzten Optionen der Annotation.

4.6 Type Handlers

Die Oberklassen der Type Handlers besitzen die Methode *executeInvariantAssertion(...)*. Diese wird von einem der Annotation Handler (siehe Kapitel 4.5) aufgerufen. Die *executeInvariantAssertion(...)*-Methode ruft die richtige *contractBroken(...)*-Methode auf, welche die Invariante überprüft.
Ist die Invariante verletzt, werden die gesetzten Optionen auf der Annotation ausgewertet (innerhalb der *executeInvariantAssertion(...)*-Methode) und dem Annotation Handler gemeldet, damit dieser die gesetzten Reaktionen ausführen kann.

4.6.1 Handlers für unterstützte Typen mit @Min, @Max oder @Range

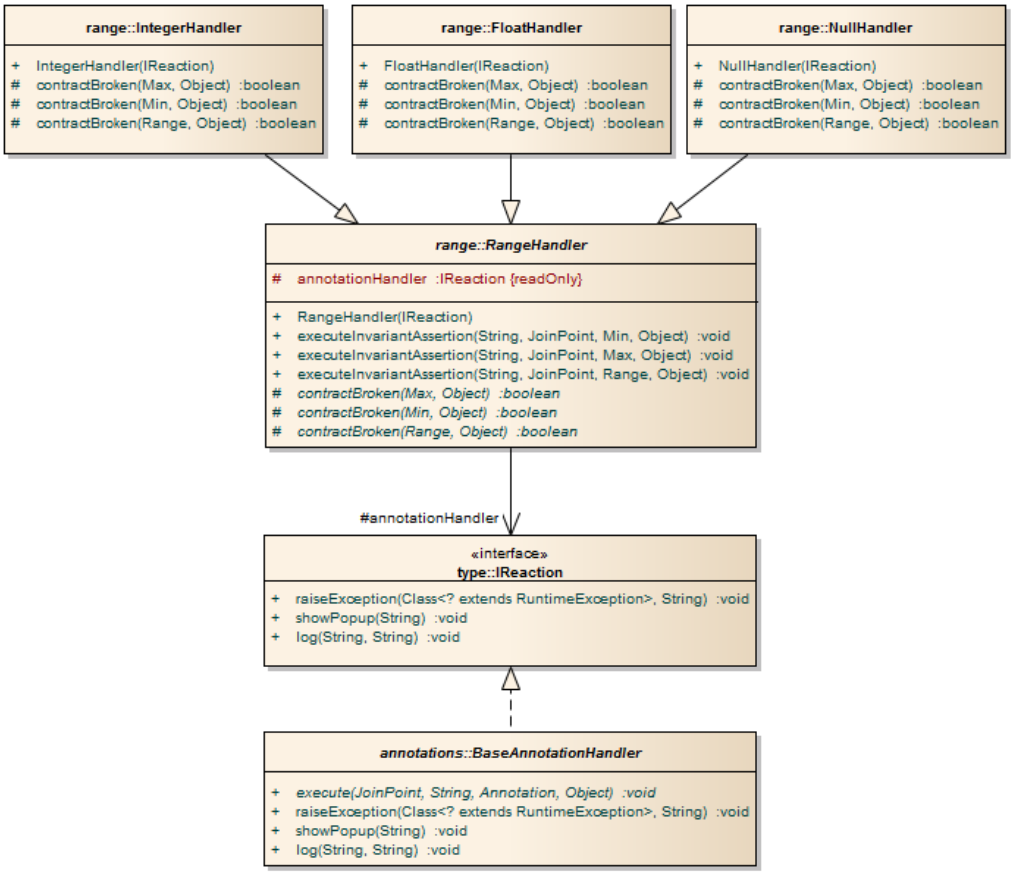


Abbildung 17: Auszug der Type Handlers

Es gibt die Type Handlers: *ByteHandler*, *CharacterHandler*, *DoubleHandler*, *FloatHandler*, *IntegerHandler*, *LongHandler*, *NullHandler*, *ShortHandler* und *StringHandler*, welche alle von *RangeHandler* erben.

Als Beipiel sieht die *contractBroken(...)*-Methode im *IntegerHandler* folgendermassen aus:

Listing 20: contractBroken() für Min auf einem Integer

```
1 @Override
2 protected boolean contractBroken(Min min, Object value) {
3     return (Integer) value < Integer.parseInt(min.value());
4 }
```

4.6.2 Handlers für unterstützte Typen mit @Multiplicity

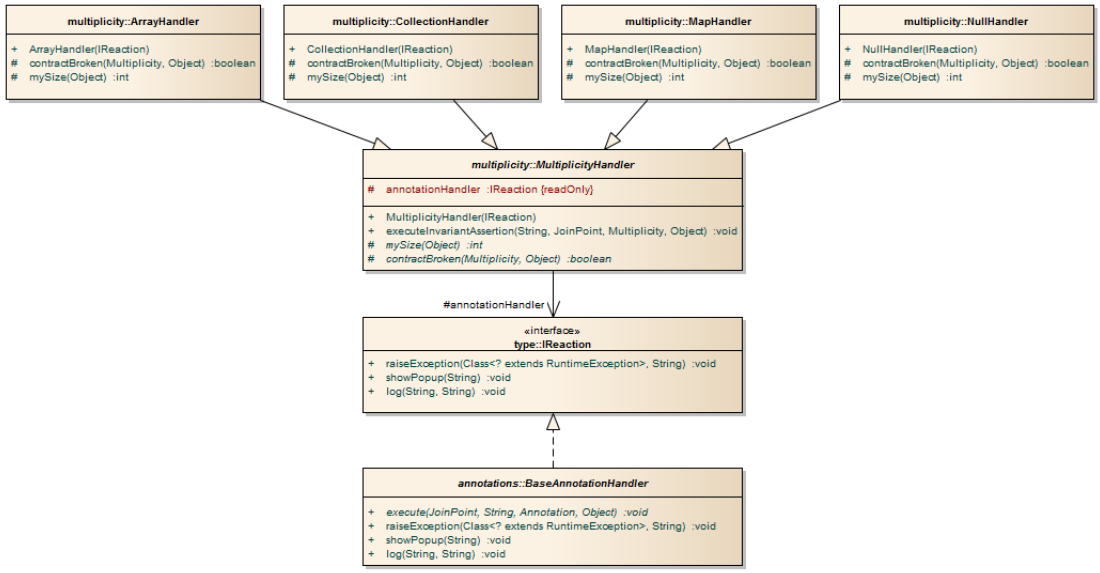


Abbildung 18: Alle Handlers für unterstützte Typen mit @Multiplicity

Als Beipiel sieht die *contractBroken(...)*-Methode im *ArrayHandler* folgendermassen aus:

Listing 21: contractBroken() für Multiplicity auf einem Array

```
1 @Override
2 protected boolean contractBroken(Multiplicity multiplicity, Object value) {
3     int lower = multiplicity.lower();
4     int upper = multiplicity.upper();
5     int currentSize = ((Object[]) value).length;
6     return (currentSize < lower) || (currentSize > upper);
7 }
```

4.6.3 Handlers für unterstützte Typen mit @Subsets

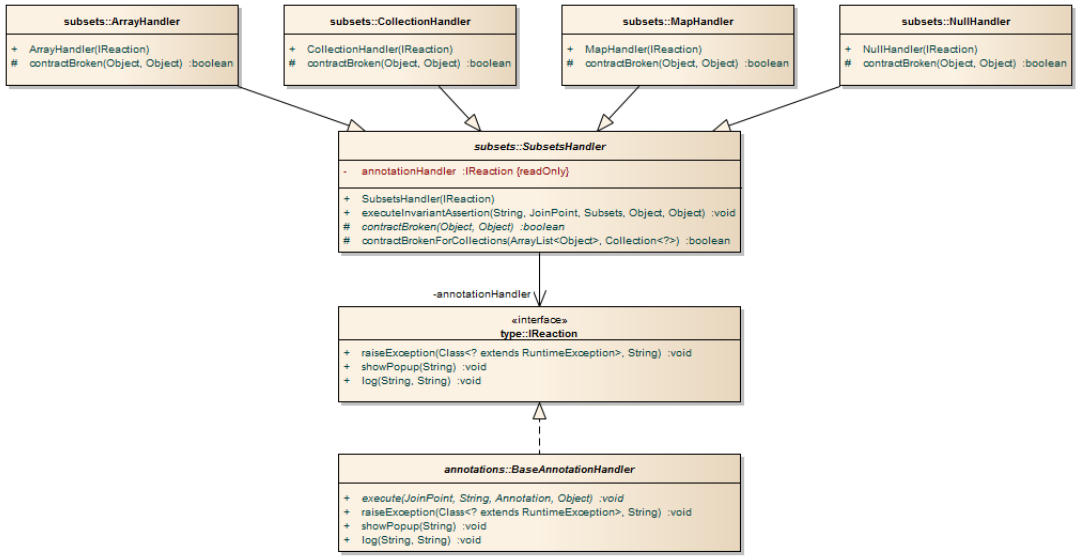


Abbildung 19: Alle Handlers für unterstützte Typen mit @Subsets

Die TypHandler der *@Subsets*-Annotation beziehen sich auf den "grösseren" Container. Daher müssen die *contractBroken(. . .)*-Methoden zusätzlich überprüfen, welchen Typ das annotierte Feld (die Teilmenge) hat. Dabei gibt es folgende Möglichkeiten:

- Das annotierte Feld ist kein Container.
  Es wird durch den grösseren Container iteriert. Findet IAGen das annotierte Feld im Container, ist die Invariante nicht verletzt. Wenn das Feld nicht gefunden wird, ist die Invariante verletzt.

- Das annotierte Feld ist eine Collection.
  Wenn der grössere Container keine Collection ist, wird sie auf eine Collection abgeändert. Danach wird die Methode *contractBrokenForCollections(. . .)* aufgerufen.

- Das annotierte Feld ist eine Map oder ein Array [ ].
  Das annotierte Feld und der grössere Container (falls noch keine Collection) wird auf eine Collection abgeändert. Danach wird die Methode *contractBrokenForCollections(. . .)* aufgerufen.

Die Methode *contractBrokenForCollections(...)* überprüft, ob sich die kleinere Collection in der grösseren Collection finden lässt. Bei Vorhandensein retourniert die Methode *false*, also keinen Invariantenbruch. Sind nicht alle Elemente von der kleineren Collection in der grösseren vorhanden, retourniert die Methode *true* und die Invariante gilt als verletzt.

Listing 22: contractBrokenForCollections()

```
1  protected final boolean contractBrokenForCollections(final ArrayList<Object
       > valueCollection,
2          final Collection<?> containerCollection) {
3
4     if (valueCollection.size() == 0) {
5        return false;
6     }
7
8     for (Object valueObject : valueCollection) {
9        boolean valueObjectMatch = false;
10       for (Object containerObject : containerCollection) {
11          if (valueObject == containerObject) {
12             valueObjectMatch = true;
13             break;
14          }
15       }
16
17       if (!valueObjectMatch) {
18          return true;
19       }
20    }
21    return false;
22 }
```

4.7 Annotation Processing

Um den Java-Entwickler bestmöglich zu unterstützen, bietet IAGen Annotation Processing [apt] an. Dadurch lassen sich potentielle Fehler bereits zur Kompilierzeit erkennen. Für jede unterstützte Annotation (*@Min*, *@Max*, *@Range*, *@Multiplicity* und *@Subsets*) gibt es einen verantwortlichen Processor:

- InvariantProcessor
  Gibt eine Warnung aus, wenn *@Invariant* nicht auf der Klasse gesetzt ist, obwohl eine der unterstützten Annotationen gesetzt ist.

- RangeProcessor
  Überprüft die Annotationen *@Min*, *@Max* und *@Range*. Gibt einen Fehler aus, wenn einer der gesetzten Werte nicht zum Typ passt (Beispiel 1.5 auf int) oder wenn das Minimum grösser gesetzt ist als das Maximum (Beispiel *@Min(“10“) @Max(“1“) oder @Range(min=“10“, max=“1“)*). Gibt eine Warnung aus, wenn *@Min*, *@Max* und *@Range* gesetzt sind.

- MultiplicityProcessor
  Gibt einen Fehler aus, wenn *@Multiplicity* auf einem falschen type gesetzt wird oder wenn *lower* bzw. *upper* kleiner als 0 ist. Gibt eine Warnung aus, wenn *lower* grösser *upper* ist.

- SubsetsProcessor
  Gibt eine Warnung aus, wenn das referenzierte Feld (value) von *@Subsets* nicht gefunden werden konnte, auf sich selber referenziert oder wenn das referenzierte Feld vom falschen type ist.

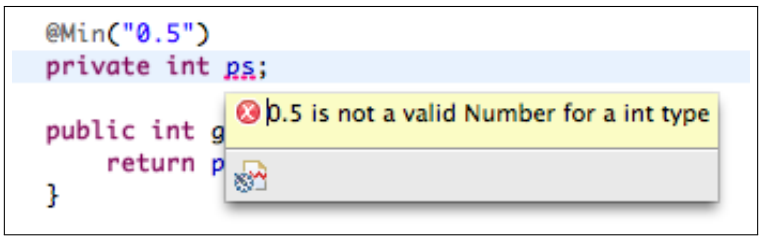


Abbildung 20: Fehler - 0.5 ist für einen int nicht zulässig

4.8 Enterprise Architect Add-In

4.8.1 Übersicht

Das IAGen Add-In greift über das Enterprise Architect Repository auf die Funktionen des API zu. Damit dies möglich ist, muss Interop.EA.dll [comb] als Assembly referenziert werden. Anschliessend stehen die Funktionen des Enterprise Architect APIs zur Verfügung.

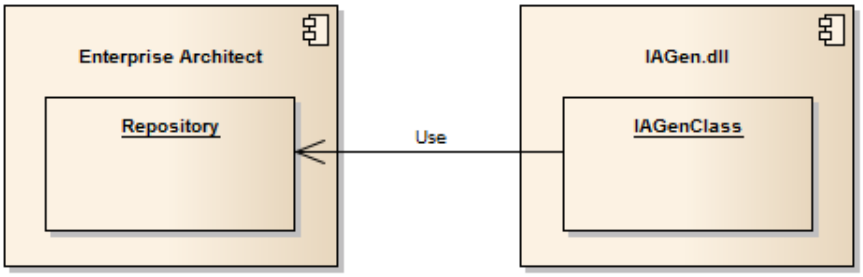


Abbildung 21: Zugriff von IAGen auf das Enterprise Architect Repository

4.8.2 Enterprise Architect API [add]

Das Enterprise Architect API bietet folgende Events an, welche verwendet werden können:

- EA_Connect
- EA_Disconnect
- EA_GetMenuItems
- EA_MenuClick
- EA_GetMenuState
- EA_ShowHelp
- EA_OnOutputItemClicked
- EA_OnOutputItemDoubleClicked

Die IAGen Add-In Klasse *IAGenClass.cs* verwendet folgende wichtige API Events:

- **public void EA_GetMenuState(...)**
  Wird aufgerufen wenn ein Add-In-Menü geöffnet wurde. Bestimmt, ob ein Menüpunkt aktiviert oder deaktiviert (ausgegraut) ist:

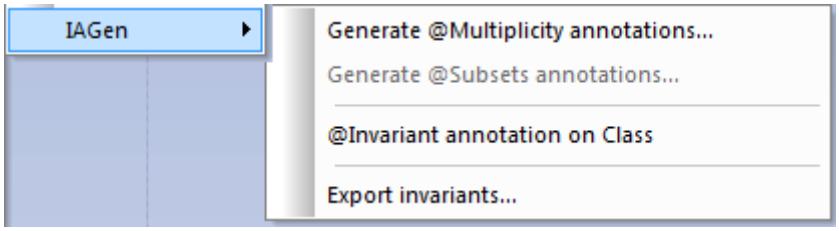


Abbildung 22: Das IAGen Add-In

- **public void EA_MenuClick(...)**
  Wird aufgerufen, sobald der Entwickler ein Menüpunkt des Add-Ins auswählt.

4.8.3 Enterprise Architect Tagged Value 'annotations'

Entscheid: Modellierte Invarianten (Assoziationen, Subsets, Wertebereiche) werden als Annotationen in die Tagged Value "annotations" der jeweiligen Elemente gespeichert. Diese Tagged Value wird bei der Generierung des Sourcecodes berücksichtigt:

Listing 23: @Invariant Annotation auf eine Klasse (clickedElement) hinzufügen.

```
1 clickedElement.TaggedValues.AddNew("annotations", "@Invariant");
```

Wir haben uns für diese Lösung entschieden, weil der Enterprise Architect bei der Generierung des Codes lediglich auf die Tagged Value "annotations" Rücksicht nimmt. Definierte Annotationen an anderen Orten werden ignoriert.

4.8.4 Multiplicity und Subsets Finder

Die im Enterprise Architect modellierten Multiplizitäten und Subsets bei Assoziationen werden von den Klassen *MultiplicityGenerator* und *SubsetsGenerator* gefunden.

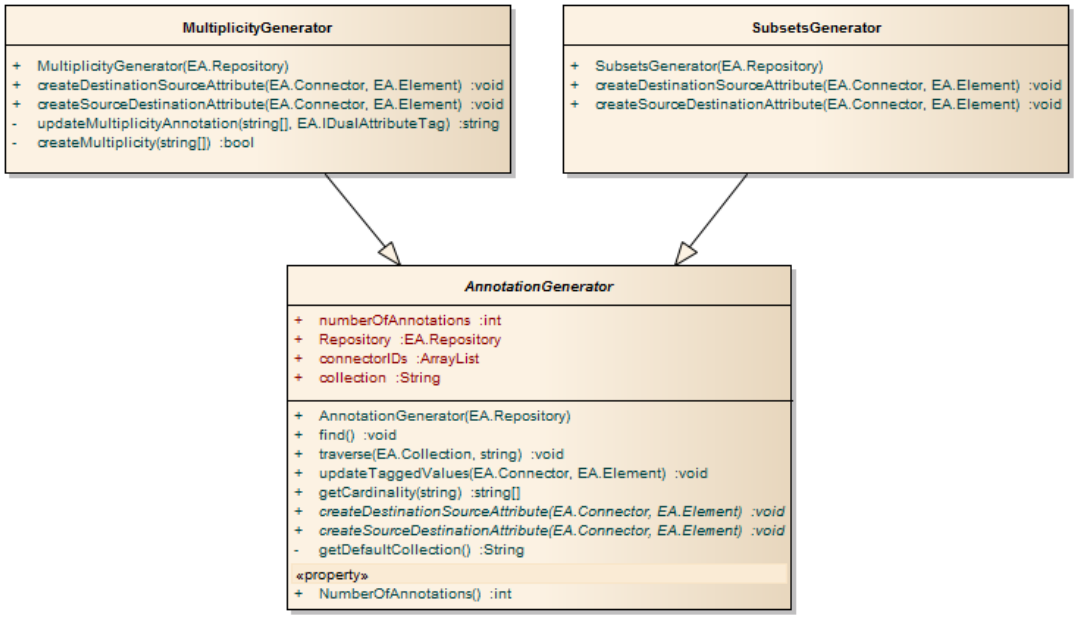


Abbildung 23: Multiplicity und Subsets Finder

Die wichtigsten Methoden der Generator-Klassen:

- **private void traverse(...)**
  Diese Methode der Oberklasse *AnnotationGenerator* traversiert durch das Repository und ruft für jeden Connector (Assoziation) die *updateTaggedValues(...)*-Methode auf.

- **public void updateTaggedValues(...)**
  Ruft je nach Connector-Typ (Source -> Destination, Destination -> Source, Bi-Directional) entweder die Methode *createDestinationSourceAttribute(...)* und/oder die Methode *createSourceDestinationAttribute(...)* auf.

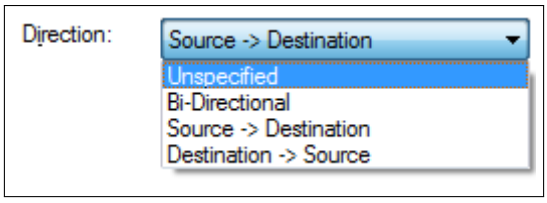


Abbildung 24: Connector-Typen im Enterprise Architect

- **public abstract void createSourceDestinationAttribute(...)**
  Setzt auf dem betroffenen Attribut eine *@Multiplicity*- oder *@Subsets*-Annotation. Falls noch kein passendes Attribut existiert, wird ein neues Attribut mit dem Namen der Rolle der Assoziation erstellt.

- **public abstract void createDestinationSourceAttribute(...)**
  Setzt auf dem betroffenen Attribut eine *@Multiplicity*- oder *@Subsets*-Annotation. Falls noch kein passendes Attribut existiert, wird ein neues Attribut mit dem Namen der Rolle der Assoziation erstellt.

- **private String getDefaultCollection(...)**
  Überprüft, ob im Enterprise Architect unter Tools -> Options... -> Source Code Engineering -> Java -> Collection Classes eine Default Collection gesetzt wurde:

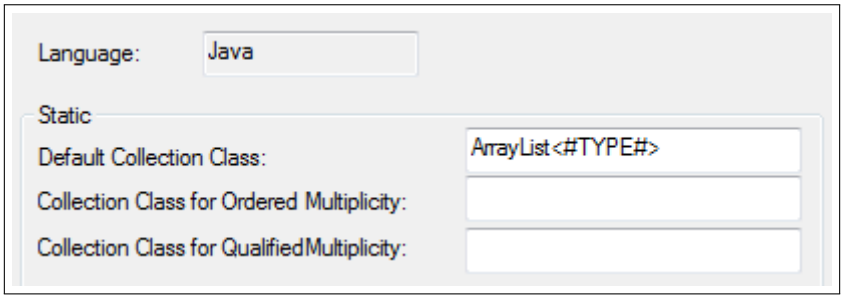


Abbildung 25: Java Collection Classes im Enterprise Architect

4.8.5 @Invariant Annotation

Entscheid: IAGen macht die Invariantenüberprüfung nur, wenn eine Klasse mit der *@Invariant*-Annotation annotiert ist. Im Enterprise Architect gilt dieselbe Regel. Assoziationen und Subsets werden nur generiert, wenn auf einer Klasse die *@Invariant*-Annotation vorhanden ist.

Listing 24: Überprüfung, ob @Invariant-Annotation auf Klasse vorhanden ist.

```
1 if (taggedValue.Value.Contains("@Invariant"))
2 {
3     // Code weggelassen
4 }
```

4.8.6 aop.xml Builder

Das IAGen Add-In kann die IAGen-Invariantendefinitionen per XML-Datei exportieren. Diese Aufgabe übernimmt die Datei *AopXmlBuilder.cs*. Nach dem Traversieren durch das Enterprise Architect Repository entsteht eine aop.xml Datei [ltw] mit folgendem Format:

Listing 25: XML Beispieldatei nach Export

```
1 <aspectj>
2   <aspects>
3     <concrete-aspect name="ch.iagen.aspects.reverse.Annotater">
4       <declare-annotation
5       type="package.Person"
6       annotation="@ch.iagen.annotations.Invariant"
7       />
8       <declare-annotation
9       field="* package.Person.cars"
10      annotation="@ch.iagen.annotations.Multiplicity(lower=1,upper=10)"
11      />
12    </concrete-aspect>
13  </aspects>
14  <weaver options="-Xreweavable -XlazyTjp -Xset:completeBinaryTypes=true"
        />
15 </aspectj>
```

Die Deklaration von Invarianten im aop.xml (Beispiel oben) wird von AspectJ beim aktuellen Stable Release (1.6.12) nicht unterstützt. Diese Funktionalitäten sind erst zu einem späteren Zeitpunkt geplant. Es existieren jedoch bereits Developer Builds, welche diese Annotationendeklarierung im aop.xml zulassen.

Entscheid: IAGen verwendet die aktuellste Developer-Version von AspectJ, damit aus dem Enterprise Architect direkt ein aop.xml generiert werden kann. Somit muss nicht ein eigenes XML-Schema/Format definiert werden, welches anschliessend von der IAGen Library (IAGen.jar) geparsed werden müsste. Weitere Informationen dazu sind im Kapitel 4.10.3 zu finden.

4.9 Testing

Um die Funktionalitäten von IAGen sicherzustellen, wurden über 940 Tests geschrieben. Die zwei Hauptpackages sind:

- ch.iagen.test.cases
  Enthalten die Tests.
- ch.iagen.test.resources
  Enthalten sogenannte "Dummy Classes". Diese werden von den Test Files aufgerufen, um das Verhalten von IAGen zu überprüfen.

Da *@Min*, *@Max* und *@Range* auf allen primitiven Datentypen (ausser boolean) und deren Wrapperklassen sowie String angewendet werden kann, lag die Möglickeit auf der Hand, die Dummy Classes generisch zu erstellen.
Entscheid: Für jeden unterstützten Typ eine eigene Dummy Class erstellen mit dem Nachteil, dass daraus viele Klassen resultieren. Das Problem mit generischen Klassen ist, das primitive Datentypen so nicht getestet werden können und dass direkte Zuweisungen auf ein generisches Feld nicht funktionieren, was aber zwingend getestet werden muss:

```
T test = 3.3; // Führt zu einem Fehler, da T auch ein Integer sein könnte.
```

Für alle Tests wurden jeweils richtige wie auch falsche Werte für alle unterstützten Typen getestet.

4.9.1 Package nach Annotationen

Für jede IAGen-Annotation wurde ein Package in ch.iagen.test.resources erstellt:

- ch.iagen.test.resources.range.* - für *@Min*, *@Max* und *@Range*
- ch.iagen.test.resources.multiplicity.* - für *@Multiplicity*
- ch.iagen.test.resources.subsets.* - für *@Subsets*
- ch.iagen.test.resources.notnull.* - für *@NotNull*
- ch.iagen.test.resources.consts.* - für *@Const*

4.9.2 Dummy Classes für range, multiplicity und notnull

Um die Feldannotationen nach den Konstruktoraufrufen testen zu können, wurden folgende sub packages erstellt:

- ch.iagen.test.resources.range.ctor.simple
  ch.iagen.test.resources.multiplicity.ctor.simple
  ch.iagen.test.resources.notnull

  Testen einen einfachen Konstruktor. Innerhalb des Konstruktors werden die Invarianten temporär verletzt (internes Setzen des Feldes und Aufruf von *setter*-Methoden).

- ch.iagen.test.resources.range.ctor.pre
  ch.iagen.test.resources.multiplicity.ctor.pre

  Testen die direkte Feld Initialisierung.

- ch.iagen.test.resources.range.ctor.thisinctor
  ch.iagen.test.resources.multiplicity.ctor.thisinctor

  Testen verschachtelte Konstruktorenaufrufe.

Um die Feldannotationen nach einem Methodenaufruf testen zu können, wurden folgende sub packages erstellt:

- ch.iagen.test.resources.range.setter
  ch.iagen.test.resources.multiplicity.setter

  Testen eine *setter* Methode. Innerhalb dieser Methode werden die Invarianten temporär verletzt (internes Setzen des Feldes).

Um die Feldannotationen nach einem direkten Setzen des Feldes testen zu können, wurden folgende sub packages erstellt:

- ch.iagen.test.resources.range.directaccess
  ch.iagen.test.resources.multiplicity.directaccess
  ch.iagen.test.resources.notnull

Um die Feldannotationen mit null-Werten testen zu können, wurden folgende sub packages erstellt:

- ch.iagen.test.resources.range.nullcheck
  Wenn *@Min("0")* oder *@Range(min="0", max="xx")* ist, darf das zugewiesene Feld null sein.

- ch.iagen.test.resources.multiplicity.nullcheck
  Wenn *Multiplicity(lower=0, upper="xx")* ist, darf das zugewiesene Feld null sein.

Um die Feldannotationen auf Vererbung testen zu können, wurden folgende sub packages erstellt:

- ch.iagen.test.resources.range.inheritance.ctor
  Testet einen *super()*-Konstruktoraufruf. Innerhalb des Konstruktors werden die Invarianten temporär verletzt (internes Setzen des Feldes und Aufruf von *setter*-Methoden).

- ch.iagen.test.resources.range.inheritance.inheritedField
  Testet vererbte und annotierte Felder.

- ch.iagen.test.resources.range.inheritance.inheritedMethod
  Testet vererbte Methoden, welche mit *super()* aufgerufen werden.

- ch.iagen.test.resources.range.inheritance.overwrittenField
  Testet annotierte Felder, die überschrieben werden. Eine Klasse überschreibt das Feld ohne Annotationen, eine andere Klasse überschreibt das Feld mit einer neuen Invariantendefinition.

- ch.iagen.test.resources.range.inheritance.superCall
  Testet überschriebene Methoden, welche in der Subklasse mit *super.method()* aufgerufen werden.

Um die Parameterannotationen testen zu können, wurden folgende sub packages erstellt:

- ch.iagen.test.resources.range.parameter.ctor
  ch.iagen.test.resources.notnull.parameter

  Testen Parameterannotationen in Konstruktoren.

- ch.iagen.test.resources.range.parameter.method
  ch.iagen.test.resources.notnull.parameter

  Testen Parameterannotationen in Methoden.

4.9.3 Dummy Classes für subsets

- ch.iagen.test.resources.subsets.simplefield
  Testet *@Subsets* auf einfachen Datenfeldern. Es gibt verschiedene Testklassen für die unterschiedlich grossen Container: Collection, MapKey, MapValue und Array.

- ch.iagen.test.resources.subsets.array
  Grosser Container ist ein Array. Die *@Subsets*-Annotation wird auf Feldertypen von Collection, MapKey, MapValue und Array getestet.

- ch.iagen.test.resources.subsets.collection
  Grosser Container ist eine Collection. Die *@Subsets*-Annotation wird auf Feldertypen von Collection, MapKey, MapValue und Array getestet.

- ch.iagen.test.resources.subsets.mapkey
  Grosser Container ist ein Map und die relevanten Daten sind im Key. Die *@Subsets*-Annotation wird auf Feldertypen von Collection, MapKey, MapValue und Array getestet.

- ch.iagen.test.resources.subsets.mapvalue
  Grosser Container ist ein Map und die relevanten Daten sind im Value. Die *@Subsets*-Annotation wird auf Feldertypen von Collection, MapKey, MapValue und Array getestet.

- ch.iagen.test.resources.subsets.inheritance
  Testet, wenn grosser Container in der Oberklasse ist und sich das annotierte Feld in der Subklasse befindet.

4.9.4 Dummy Classes für consts

- ch.iagen.test.resources.consts.field
  Testet *@Const* für Felder.

- ch.iagen.test.resources.consts.method
  Testet *@Const*-Methoden.

- ch.iagen.test.resources.consts.parameter
  Testet *@Const*-Parameter in Methoden und Konstruktoren.

4.9.5 QS und Testauswertung

Die Massnahmen zur Qualitätssicherung sowie die Testauswertungen findet man unter:

02_QM/QS_und_Testauswertung_v2.0.pdf

4.10 Verworfene Ideen

4.10.1 IAGen mit APT [apt]

Mit AspectJ wird die Logik für die IAGen-Invariantenüberprüfung zur Ladezeit (Load Time Weaving) in den bestehenden Byte Code verwoben. Eine andere Variante wäre der Einsatz von APT, um neue Java-Source-Dateien beim Kompilieren zu erzeugen. Diese neuen Java-Source-Dateien würden anschliessend die Logik für die IAGen-Invariantenüberprüfung beinhalten.
Die APT-Variante wurde nicht eingesetzt, weil der Entwickler dann seinen ursprünglichen Code mit den neu generierte Java-Source-Dateien ersetzen müsste. Somit gehen Flexibilität und Einfachheit verloren.

4.10.2 IAGen mit AST-Transformierung [ast]

Mit der AST-Transformierung könnten Invariantenüberprüfungen beim Kompilieren hinzugefügt werden. Allerdings bietet Java keine direkte Unterstützung für AST-Transformationen an. Es existieren jedoch unveröffentlichte APIs von javac, welche dies ermöglichen würden. Bei Änderungen am JDK könnten diese API-Definitionen jedoch ändern und das Transformieren würde nicht mehr funktionieren.

4.10.3 Eigenes XML-Schema für Annotationen auf Bytecode

AspectJ bietet die Möglichkeit, beim Load Time Weaving [ltw] mit Bytecode ein aop.xml anzugeben. In dieser XML-Datei können Aspects definiert werden. Mit dem *<declare-annotation>*-Attribut werden dann neue Annotationen auf Felder, Methoden oder Klassen von Bytecode gesetzt.

Als zweite Möglichkeit wurde getestet, ob dies auch mit einem eigenen XML-Schema möglich wäre:

Listing 26: Eigenes XML-Schema für die Definition von Invarianten

```
<iagen>
    <min>
        <field>
            <name>numberOfChildren</name>
            <class>Person</class>
            <package>ch.iagen.domain</package>
            <value>3</value>
        </field>
    </min>
</iagen>
```

Die Idee wurde verworfen, da man bei AspcetJ (Developement Release vom 11.04.2012) im aop.xml bereits sehr flexibel neue Annotationen deklarieren kann. Zudem müssten beim Programmstart sowohl die eigene XML-Datei mit den Invariantendefinitionen als auch ein aop.xml mit den IAGen Aspects angegeben werden. Dies wollten wir vermeiden, damit die Konfiguration für den Entwickler einfach bleibt.

4.10.4 Annotation-Vererbung mit *xajavac*

Wie in Kapitel 4.3 erwähnt, lassen sich Annotationen nicht vererben. Dies verunmöglicht es bei Annotationen gegen eine gemeinsamen Oberklasse zu programmieren. Eine Alternative wäre der Einsatz des Extended Annotation Enabled javac (*xajavac*) von Mathias Ricken:

`http://www.cs.rice.edu/~mgricken/research/xajavac/`

Bei *xajavac* wurden minimale Änderungen am Java Compiler vorgenommen, um Vererbung bei Annotationen zu unterstützen:

Listing 27: Vererbung bei Annotationen

```
1 @interface SubAnnotation extends BaseAnnotation {
2   String value();
3 }
```

Auf den Einsatz von *xajavac* wurde verzichtet, weil diese Funktionalität von Java nicht unterstützt wird und zusätzlich auch das Reflection API angepasst wurde. Zukünftige Releases von Java würden die weitere Funktionalität von *xajavac* nicht garantieren.

4.11 Einschränkungen

4.11.1 Info Meldungen beim Annotation Processing

Wegen eines offenen Bugs können im Eclipse keine Info-Meldungen beim Annotation Processing angezeigt werden. Zurzeit werden lediglich Warnungen (*Kind.WARNING*) und Fehler (*Kind.ERROR*) unterstützt.

`https://bugs.eclipse.org/bugs/show_bug.cgi?id=342600`

Setzt der Java-Entwickler im Programmcode IAGen-Annotationen und vergisst dabei, die Annotation @Invariant zu setzen, wird deshalb eine Warnung und keine Info-Meldung angezeigt.

4.11.2 Keine Methodenparameter-Annotationen im aop.xml

Werden die im Enterprise Architect modellierten Invarianten gemäss Kapitel 4.8.6 exportiert, können der XML-Datei nachträglich lediglich Klassen- (type="pkg.Class"), Feld- (field="* pkg.Class.field") und Methodenannotationen (method="* pkg.Class.method(..)") manuell hinzugefügt werden. Konstruktor- und Methodenparameter-Annotationen werden von AspectJ nicht unterstützt.

`https://bugs.eclipse.org/bugs/show_bug.cgi?id=375881`

4.11.3 Keine import Statements beim Forward Engineering

Werden im Enterprise Architect die modellierten IAGen Invarianten generiert und anschliessend mittels Forward Engineering mit dem bestehenden Sourcecode synchronisiert, werden die Java *import* Statements nicht automatisch generiert. Der Java-Entwickler muss die imports manuell hinzufügen (Ctrl + Shift + o bei Eclipse).

4.11.4 super() calls bei vererbten Methoden

Wird in einer Methode einer vererbten Klasse ein Methodenaufruf auf eine obere Klasse ausgeführt (*super.method()*), interpretiert AspectJ diesen nicht als *call()*. Die Invariantenüberprüfung findet erst nach Ablauf der Methode, welche die vererbte Klasse ausgeführt hat, statt.

http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html

4.11.5 Überprüfung von *null* Werten bei nicht unterstützten Typen

Wird der IAGen Annotationen Processor nicht verwendet und wird eine der Annotationen *@Min*, *@Range* oder *@Multiplicity* mit Untergrenze grösser 0 auf einen nicht unterstützen Datentyp mit dem Wert *null* gesetzt, gilt die Invariante als verletzt.

4.11.6 Invariantenüberprüfung auf Referenzen zu annotierten Feldern

Werden Felder, welche mit einer IAGen Annotation versehen sind, anderen Feldern zugewiesen, findet keine Invariantenüberprüfung statt. Dies kommt vor, wenn das annotierte Feld einer Klasse *public* ist oder man über eine *public*-Methode darauf zugreifen kann. Das Problem entsteht dadurch, weil AspectJ keinen *set()* nach dem *get()* auslöst:

Listing 28: Führt zu keiner Invariantenverletzung

```
1  @Invariant
2  public class Person {
3
4      @Multiplicity(lower=0, upper=2)
5      public ArrayList<Car> cars = new ArrayList<Car>();
6
7      public static void main(String[] args) {
8          Person p = new Person();
9
10         ArrayList<Car> tmpCar = p.cars; // ApsectJ get() auf cars Feld
11         tmpCar.add(new Car()); //kein AspectJ set() auf cars Feld
12         tmpCar.add(new Car());
13         tmpCar.add(new Car());
14     }
15 }
```

Obige Problematik ergibt sich auch bei *@Const*-Annotationen auf Feldern, da IAGen keine entsprechende Annotierung auf Rückgabewerte einer Methode anbietet.

4.11.7 Enterprise Architect Forward Engineering

Synchronisiert man im Enterprise Architect das Modell mit dem bestehenden Sourcecode, werden bei mehreren Annotationen hintereinander diese auf einer Linie angezeigt:

Listing 29: Annotationen vorher (zwei Linien)

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target({ ElementType.FIELD, ElementType.PARAMETER })
3 public @interface Max {
4     // ..
5 }
```

Listing 30: Annotationen nachher (eine Linie)

```
1 @Retention(RetentionPolicy.RUNTIME)@Target({ ElementType.FIELD })
2 public @interface Max {
3     // ..
4 }
```

Das führt zu Problemen bei der Synchronität mit dem Sourcecode. Deshalb werden Annotationen im IAGen-Sourcecode auf einer Linie geführt.

Literaturverzeichnis

[add] Enterprise Architect Add-in Events. http://www.sparxsystems.com/uml_tool_guide/sdk_for_enterprise_architect/eaaddintemplateobject.htm.

[ann] Extending Java Annotations. http://stackoverflow.com/questions/1624084/why-is-not-possible-to-extend-annotations-in-java.

[apt] Annotation processor. http://docs.oracle.com/javase/1.5.0/docs/guide/apt/GettingStarted.html.

[ast] Abstract syntax tree. http://en.wikipedia.org/wiki/Abstract_syntax_tree.

[coma] ActiveX COM objects. http://www.sparxsystems.com/uml_tool_guide/sdk_for_enterprise_architect/addins_2.htm.

[comb] COM Interop. http://en.wikipedia.org/wiki/COM_Interop.

[eaa] Enterprise Architect Add-in Architektur. http://geertbellekens.files.wordpress.com/2011/02/ea-addin.png.

[ltw] AspectJ Load Time Weaving mit aop.xml. http://www.eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html#configuring-load-time-weaving-with-aopxml-files.

IAGen
Invariants-Assertion-Generation

Developer Guide

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	03.04.2012	f1egli	Erste Version
1.0rc02	03.04.2012	dmengelt	EA Add-In
1.0rc03	04.05.2012	dmengelt	Erweiterungen EA Add-In
1.0rc04	15.05.2012	f1egli	Java Teil erweitert
1.0rc05	15.05.2012	dmengelt	Änderungen Enterprise Architect Add-In
1.0	17.05.2012	dmengelt	Version 1.0
2.0rc01	18.05.2012	dmengelt	Anpassungen gemäss Wochenbesprechung
2.0rc02	29.05.2012	f1egli	Subsets Annotationen
2.0rc03	29.05.2012	f1egli	Gemeinsame Einstellungsmöglichkeiten der Annotationen
2.0rc04	29.05.2012	f1egli	NotNull
2.0rc05	29.05.2012	f1egli	Const
2.0rc06	30.05.2012	dmenget	Subsets im Add-In
2.0rc07	30.05.2012	f1egli	Parameterannotationen
2.0rc08	01.06.2012	f1egli	Internes Logging
2.0	04.06.2012	dmengelt	Finale Version 2.0

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Java-Sourcecode-Annotierung

3.1 Eclipse und IAGen

Damit die Annotierungen erkannt und ausgewertet werden können, braucht es in Eclipse einige Anpassungen. Am einfachsten kopiert man IAGen.jar in einen Ordner, welcher im Package Explorer des jeweiligen Projektes ersichtlich ist.

3.1.1 Build Path

IAGen.jar muss in den Build-Path eingebunden werden.

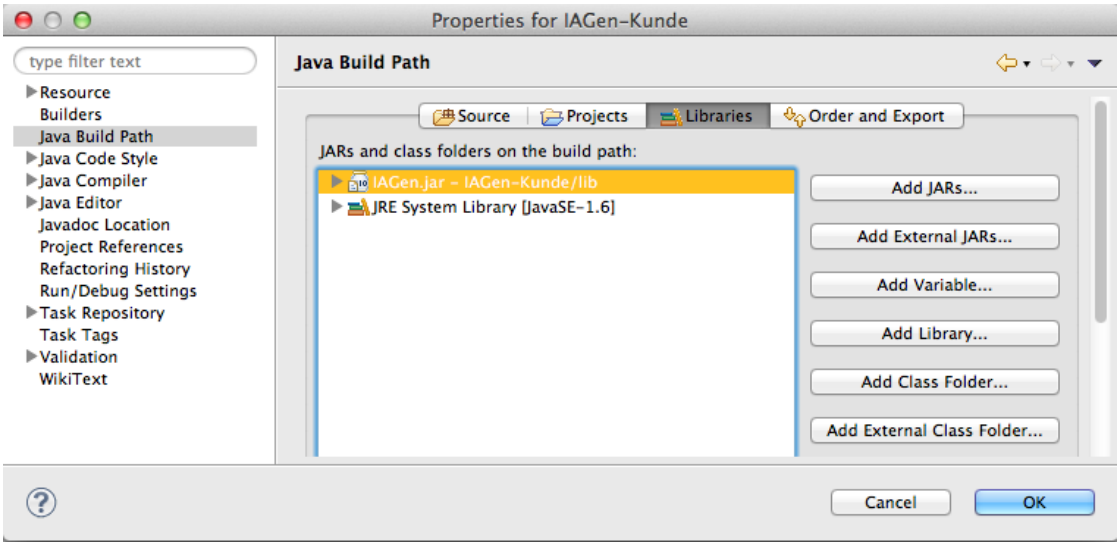


Abbildung 1: Properties des Projektes -> Java Build Path -> Libraries

Dadurch werden die Annotierungen wie *@Max* etc. gefunden.

3.1.2 Annotation Processor

Den Annotation Processor muss man ebenfalls in den Properties des Projektes einstellen. Das Menü befindet sich unter: Java Compiler -> Annotation Processing bzw unter: Java Compiler -> Annotation Processing -> Factory Path

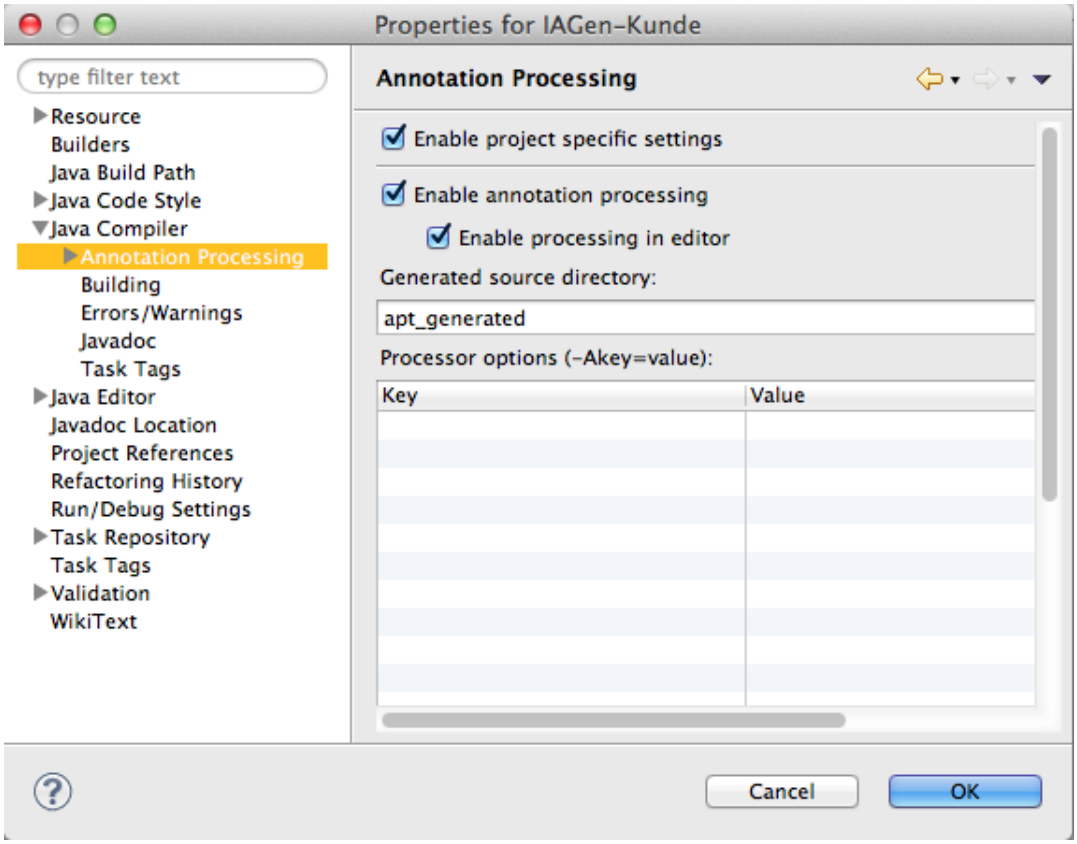


Abbildung 2: Annotation Processing muss aktiviert werden

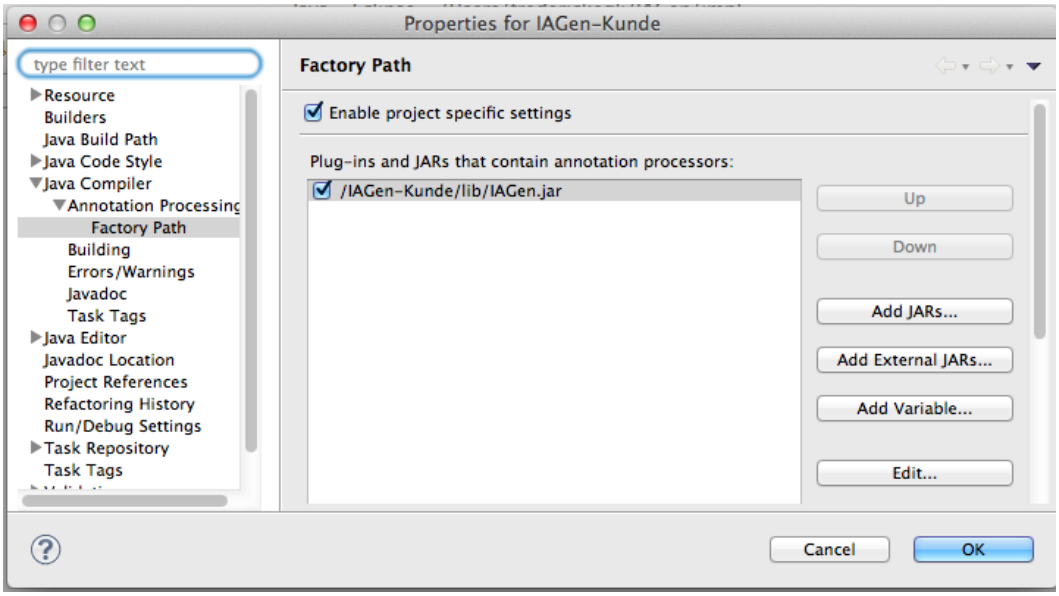


Abbildung 3: IAGen.jar als Factory Path hinzufügen

Durch diese Konfiguration werden die Annotation Processors von IAGen beim Kompilieren des Projektes gestartet. Dadurch lassen sich Fehler erkennen. Beispielsweise wird erkannt, falls der Werttyp nicht zum Feldtyp passt:

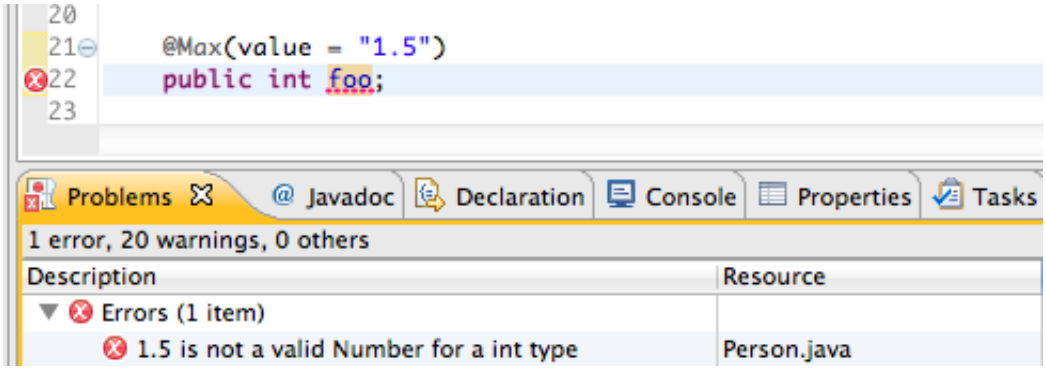


Abbildung 4: Fehler, da 1.5 nicht zu Integer geparst werden kann

Man muss die Java Datei allerdings speichern, damit die sie kompiliert wird und so den Annotation Processor anstösst.

3.1.3 Run Configuration

Damit die Invarianten zur Laufzeit überprüft werden, muss die IAGen Logik in den Code verwoben werden. Damit dies geschieht, muss in der Run Configuration das Argument: -javaagent:lib/IAGen.jar gesetzt werden (Pfad zum Jar-File kann natürlich variieren).

Beim gewünschten Java File: Rechter Mausklick -> Run As -> Run Configurations...

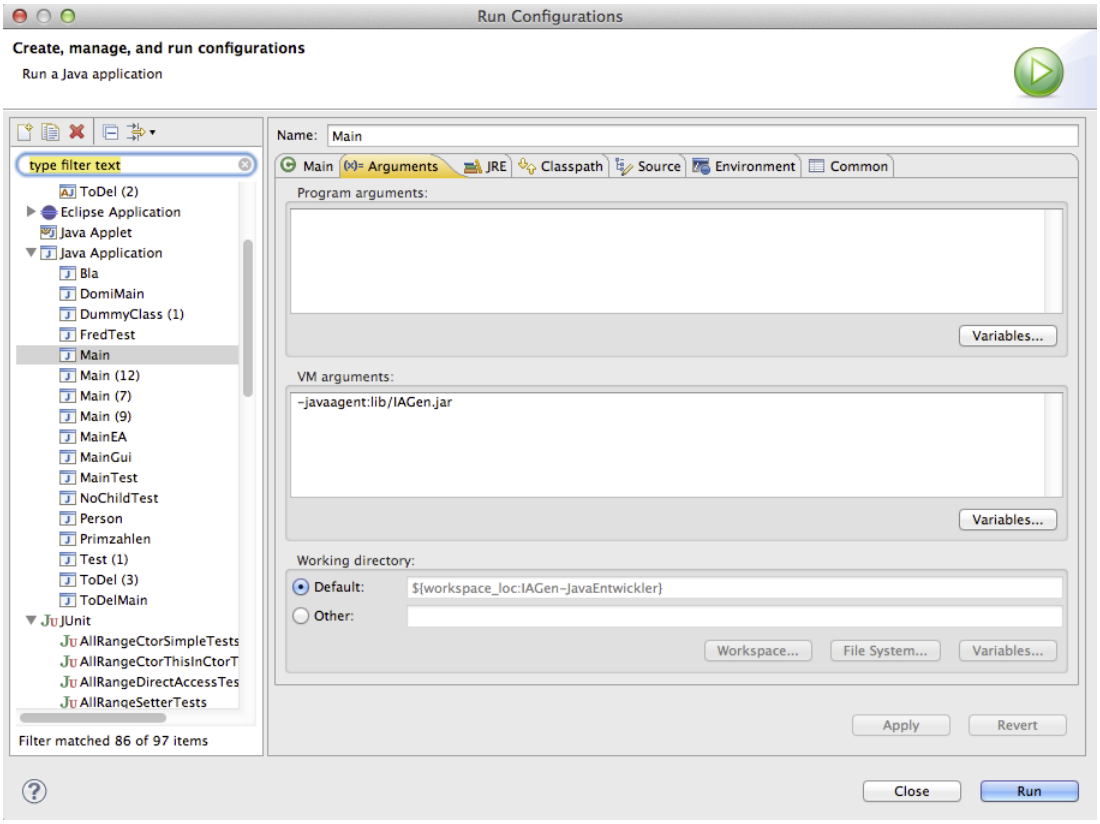


Abbildung 5: Unter Arguments -> VM arguments -> -javaagent:PATH-TO-IAGen.jar

Nach einem Klick auf Run werden die Invarianten zur Laufzeit ausgewertet.

3.2 Markierungsannotation

Alle Annotierungen, die IAGen zu Verfügung stellt, werden erst zur Laufzeit ausgewertet, wenn die Klasse mittels *@Invariant* annotiert ist:

Listing 1: Range wird überprüft, da Klasse markiert ist

```
1 @Invariant
2 public class Car {
3     @Range(min="0", max="1000")
4     public int ps;
5 }
```

Ein Java-Entwickler kann dadurch die Invariantenüberprüfung bequem "ein- bzw. ausschalten". Er muss lediglich die Markierannotation *@Invariant* kommentieren bzw. auskommentieren. Da die Annotation allerdings leicht vergessen werden kann, generiert der Annotation Processor eine Warnung, falls eine IAGen-Annotation ohne die *@Invariant*-Annotation verwendet wurde.

3.3 Gemeinsame Einstellungsmöglichkeiten der Annotationen

Es gibt Einstellungen, die bei allen Annotationen (ausser *@Const*) gleich sind. Die Einstellungen sind mit Standardwerten vordefiniert. Man muss diese Werte also nur abändern, wenn man vordefinierten Einstellungen anpassen möchte.

- *boolean showPopup() default false;*
  Per Standard ausgeschaltet. Wenn eingeschaltet, kommt bei einer Invariantenverletzung ein Swing Popup zum Vorschein.
- *boolean raiseException() default true;*
  Per Standard eingeschaltet. Wirft bei einer Invariantenverletzung eine Exception.
- *Class<? extends RuntimeException> exceptionClass() default InvariantBrokenException.class;*
  Per Standard wird eine *InvariantBrokenException* geworfen, wenn raiseException() eingeschaltet ist. Die Exceptionklasse kann hier angepasst werden. Sie muss jedoch von *RuntimeException* abgeleitet sein.
- *String exceptionMessage() default "@xxx contract broken!";*
  Per Standard wird für jede Annotation eine Exceptionmessage vordefiniert. Der Text der Exception kann über *exceptionMessage()* angepasst werden.
- *boolean debug() default false;*
  Per Standard ausgeschaltet. Wenn eingeschaltet, wird eine Debugmeldung auf die Konsole geschrieben. Diese Meldung kommt immer (unabhängig ob Invariante verletzt wird oder nicht), wenn das Feld neu gesetzt wird.
- *boolean log() default false;*
  Per Standard ausgeschaltet. Wenn eingeschaltet, werden die Invariantenverletzungen in ein Logfile geschrieben.
- *String logFile() default "log.txt";*
  Wenn log() eingeschaltet ist, werden die Invariantenverletzungen per Standard in log.txt gespeichert. Der Name des Logfiles kann hier gesetzt werden.

3.4 Wertebereich

Es werden folgende Annotation unterstützt:

- *@Max*
- *@Min*
- *@Range*

Diese Annotationen werden auf Felder gesetzt. Es sind alle primitiven Datentypen (ausser Boolean) sowie deren Wrapperklassen und String unterstützt.

3.4.1 @Min und @Max

Möchte man die Standardeinstellungen der Invarianten beibehalten, so reicht es, den Wert als String () anzugeben:

Listing 2: Beispiel einer Min-Annotation

```
1 @Invariant
2 public class Car {
3
4     @Min("2")
5     public int numberOfDoors;
6 }
```

Setzt man eine Wertebereichannotation auf ein Stringfeld, so wird die Länge des Strings als Wert interpretiert. Beispielsweise kann man so einen String auf eine maximale Länge beschränken:

Listing 3: String marke darf nur 10 Zeichen lang sein

```
1 @Invariant
2 public class Car {
3
4     @Max("10")
5     public String marke;
6 }
```

Natürlich lassen sich *@Min* und *@Max* kombinieren:

Listing 4: Beispiel einer Min- und Max-Annotierung

```
1 @Invariant
2 public class Car {
3
4     @Min("2")
5     @Max("5")
6     public int numberOfDoors;
7 }
```

Für *@Min* und *@Max* gibt es folgende Einstellungsmöglichkeiten:

- *String value();*
  Muss immer gesetzt werden, um den minimalen/maximalen Wert des Feldes zu definieren. Wenn nur value gesetzt wird (und alle anderen Einstellungen auf Default bleiben), reicht es aus, die Annotierung ohne das Wort Value zu setzen (*Beispiel:*

@Max("10")). Sobald Default-Werte geändert werden, muss das Wort value gesetzt werden (Beispiel: *@Max(value="10", debug=true)*)

  - Wenn der value auf 0 gesetzt ist, darf das Feld *null* (bei Wrapperklassen und Strings) sowie 0 sein.

- Alle anderen Einstellungsmöglichkeiten sind unter 3.3 zu finden.

Als Beispiel soll das Feld *public int numberOfDoors* Mindestens zwei und Maximal fünf sein dürfen. Es wird keine Exception geworfen, jedoch sollen Debug Informationen angezeigt werden und es soll ein Popup bei einer Invariantenverletzung erscheinen:

Listing 5: Kombination von Einstellungen

```
1 @Invariant
2 public class Car {
3
4     @Min(value="2", raiseException=false, showPopup=true, debug=true)
5     @Max(value="5", raiseException=false, showPopup=true, debug=true)
6     public int numberOfDoors;
7 }
```

3.4.2 @Range

@Range ist die Kombination von *@Min* und *@Max*. Mittels dieser Annotation lässt sich also einen Bereich mit Unter- und Obergrenze definieren. Möchte man die Standardeinstellungen der Annotation beibehalten, müssen lediglich *min* und *max* als Einstellung für die Grenzen gesetzt werden:

Listing 6: Beispiel einer Range-Annotierung

```
1 @Invariant
2 public class Car {
3
4     @Range(min="2", max="5")
5     public int numberOfDoors;
6 }
```

Für *@Range* gibt es folgende Einstellungsmöglichkeiten:

- *String min();*
  Muss immer gesetzt werden, um den minimalen Wert des Feldes zu definieren.

  - Wenn der value auf 0 gesetzt ist, darf das Feld *null* (bei Wrapperklassen und Strings) sowie 0 sein.

- *String max();*
  Muss immer gesetzt werden, um den maximalen Wert des Feldes zu definieren.

- Alle anderen Einstellungsmöglichkeiten sind unter 3.3 zu finden.

3.5 @Multiplicity

Um Multiplizitäten auf einem Datenfeld zu definieren, wird die Annotation *@Multiplicity* verwendet. Diese Annotation lässt sich auf den folgenden Typen definieren:

- *Map*
- *Collection*
- *Array [ ]*

Werden die Standardeinstellungen verwendet, reicht die Angabe einer Unter- und einer Obergrenze:

Listing 7: Eine Person besitzt zwischen 0 und 2 Autos

```
1 @Invariant
2 public class Person {
3
4     @Multiplicity(lower = 0, upper = 2)
5     public ArrayList<Car> cars;
6 }
```

Das gleiche Beispiel unter Verwendung eines Arrays:

Listing 8: Eine Person besitzt zwischen 0 und 2 Autos

```
1 @Invariant
2 public class Person {
3
4     @Multiplicity(lower = 0, upper = 2)
5     public Car[] cars;
6 }
```

Für *@Multiplicity* gibt es folgende Einstellungsmöglichkeiten:

- *int lower();*
  Muss immer gesetzt werden, um die untere Grenze der Multiplizität zu definieren.
  - Wenn lower als 0 gesetzt ist, so darf das Feld *null* oder leer sein.
  - Wenn lower grösser als 0 ist, muss beachtet werden, dass das annotierte Feld nie *null* oder leer sein darf, da sonst eine Invariantenverletzung vorliegt:

    Listing 9: Invariantenverletzung nach Konstruktoraufruf

    ```
    1  @Invariant
    2  public class Person {
    3
    4      @Multiplicity(lower = 1, upper = 2)
    5      public ArrayList<Car> cars;
    6
    7      public Person() {
    8          cars = new ArrayList<Car>();
    9      }
    10 }
    ```

- *int upper();*
  Muss immer gesetzt werden, um die obere Grenze der Multiplizität zu definieren.
- Alle anderen Einstellungsmöglichkeiten sind unter 3.3 zu finden.

3.6 @Subsets

Mit der Annotation *@Subsets* lassen sich Teilmengen auf einen grösseren Container definieren. Als Beispiel: Es gibt eine Turnierklasse. Diese besitzt eine Liste (grosser Container) mit Spielern, welche *players* heisst. Ein Turnier kann einen Gewinner haben. Mit *@Subsets* kann man auf dem Gewinnerfeld die Invariante definieren, dass das Feld winner eine Teilmenge von *players* sein muss. Man kann sagen: "Ein Gewinner eines Turnieres muss auch daran teilgenommen haben".

Werden die Standardeinstellungen verwendet, reicht die Angabe des Feldnamens des Containers:

Listing 10: @Subsets auf ein Feld

```
1  @Invariant
2  public class Tournament {
3
4      List<Player> players;
5
6      @Subsets("players") // winner muss in der players-Liste vorhanden sein.
7      Player winner;
8
9      public Tournament() {
10         players = new ArrayList<Player>();
11     }
12
13     public void setWiner(Player p){
14         winner = p;
15     }
16 }
```

Das Feld winner darf mit *@Subsets* annotiert sein, wenn der Container (in unserem Beispiel *players*) vom Typ *Map*, *Collection* oder *Array [ ]* ist.

Das oben genannte Beispiel kann erweitert werden. Wenn es nicht nur einen Spieler gibt, sondern mehrere, dann wird das Feld *winner* ebenfalls zu einem Container. Die Annotation *@Subsets* darf nicht nur auf einzelnen Feldern annotiert werden, sondern ebenfalls auf einer *Map*, einer *Collection* oder einem *Array [ ]*.

Listing 11: @Subsets auf einer Liste

```
1  @Invariant
2  public class Tournament {
3
4      List<Player> players;
5
6      @Subsets("players")
7      List<Player> winners; //Alle Gewinner müssen beim Turnier mitgespielt
           haben.
8
9      public Tournament() {
10         players = new ArrayList<Player>();
11         winners = new ArrayList<Player>();
12     }
13
14     public void addWiner(Player p) {
15         winners.add(p);
16     }
17 }
```

Der kleine Container (*winners*) und der grosse Container (*players*) müssen nicht zwingend vom gleichen Typ sein. Als Beispiel könnte players auch eine Map sein und winners ein Array [ ].

Ist das annotierte Feld *null* oder besitzt keine Einträge in dem Container, wird dies nicht als Invariantenverletzung interpretiert.

Für *@Subsets* gibt es folgende Einstellungsmöglichkeiten:

- *String value();*
  Muss immer gesetzt werden, um den Namen des "grösseren" Containers anzugeben. Wenn nur value gesetzt wird (und alle anderen Einstellungen auf Default bleiben), reicht es aus, die Annotierung ohne das Wort Value zu setzen (Beispiel: *@Subsets("players")*. Sobald Default-Werte geändert werden, muss das Wort Value gesetzt werden (Beispiel: *@Subsets(value="players", debug=true)*)
- Alle anderen Einstellungsmöglichkeiten sind unter 3.3 zu finden.

3.7 @NotNull

Wie der Name der Annotation *@NotNull* sagt, dürfen Felder, welche diese Annotation tragen, nicht den Wert *null* annehmen.

Listing 12: String name darf nie null sein

```
1  @Invariant
2  public class Player {
3
4      @NotNull
5      String name;
6
7      Player(String name) {
8          this.name = name;//Invariantenbruch, wenn name auf null gesetzt wird.
9      }
10
11     public void setName(String name){
12         this.name = name;//Invariantenbruch, wenn name auf null gesetzt wird.
13     }
14
15     public void setNameToNull() {
16         this.name = null;//Invariantenbruch, da name auf null gesetzt wird.
17     }
18 }
```

Diese Annotation darf auf allen Typen gesetzt werden (auch auf primitiven Datentypen, wobei diese nie *null* annehmen und daher die Annotation ignoriert wird).

Alle Einstellungsmöglichkeiten von *@NotNull* sind unter 3.3 zu finden.

3.8 @Const

Um die *@Const*-Annotierungen zur Laufzeit eines Programmes auswerten zu lassen, benötigt man zusätzliche Einstellungen in den Run Configurations von Eclipse.

Möcht man nur die *@Const* Features aktivieren (*@Min, @Max* etc. werden dann ignoriert), muss man eine System Property in den VM arguments entsprechend setzen:

```
-Dorg.aspectj.weaver.loadtime.configuration=META-INF/const.xml
```

Beim gewünschten Java File: Rechter Mausklick -> Run As -> Run Configurations...

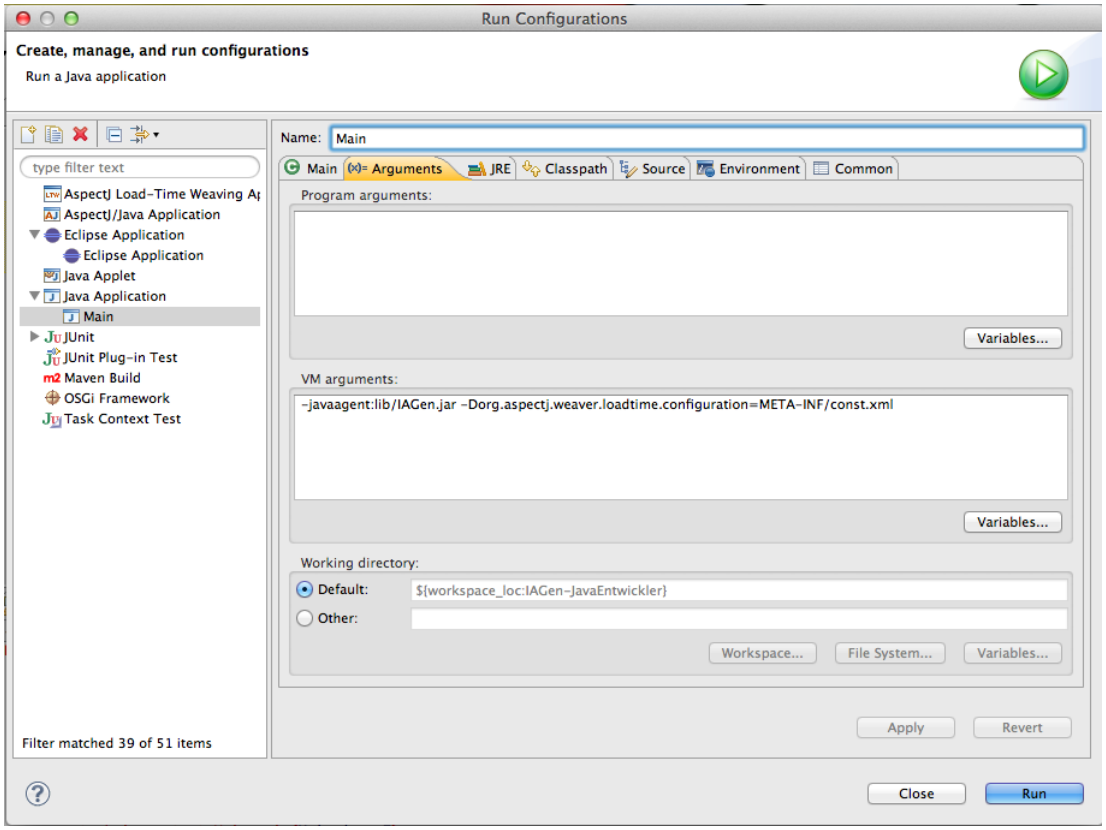


Abbildung 6: IAGen überprüft nur Const-Annotationen

Um alle IAGen Features zu aktivieren, muss man eine System Property in den VM arguments entsprechend setzen:

```
-Dorg.aspectj.weaver.loadtime.configuration=META-INF/const.xml;META-INF/aop.xml
```

Beim gewünschten Java File: Rechter Mausklick -> Run As -> Run Configurations...

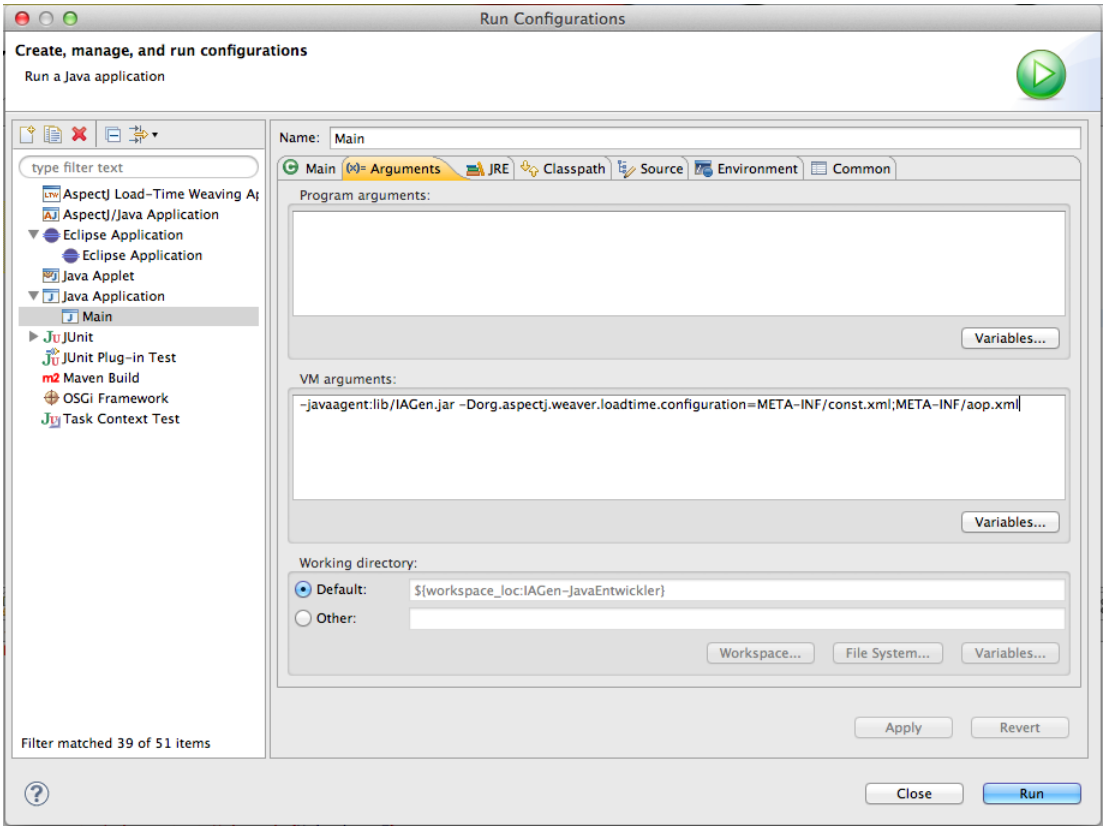


Abbildung 7: IAGen überprüft alle Annotationen

@Const kann an drei verschiedenen Orten gesetzt werden:

- auf Datenfeldern,
- auf Methoden,
- auf Methoden- und Konstruktorenargumenten.

Für *@Const* gibt es keine Einstellungsmöglichkeiten.

3.8.1 @Const auf Datenfeldern

Wenn ein Datenfeld mit *@Const* annotiert ist, dann dürfen in diesem Objekt keine Werte geändert werden. Es dürfen ausserdem nur Methoden auf diesem Feld aufgerufen werden, die mit *@Const* gekennzeichnet sind. Als Beispiel dient eine Personklasse. Eine Person hat ein *Car*-Datenfeld, welches mit *@Const* definiert ist.

Listing 13: Die Klasse Car ohne Const-Methoden

```
1 public class Car {
2    public int ps;
3    public void setPs(int ps) {
4       this.ps = ps;
5    }
6 }
```

Listing 14: Das Auto Objekt der Person darf inhaltlich nicht verändert werden

```
1  @Invariant
2  public class Person {
3
4     @Const
5     private Car car;
6
7     public Person(Car car) {
8        this.car = car;
9     }
10
11    public void changeCarsPs(int ps){
12       car.setPs(ps); //Exception, da setPs(int ps) nicht const ist.
13       car.ps = ps; //Exception, da Datenfeld geändert wird.
14    }
15 }
```

Werden Methoden auf dem Konstanten Feld aufgerufen, die selber Const sind, wird keine Exception geworfen:

Listing 15: Die Klasse Car mit einer Const-Methode

```
1 @Invariant
2 public class Car {
3    private int ps;
4
5    @Const public int getPs() {
6       return ps;
7    }
8 }
```

Listing 16: getCarsPs() darf aufgerufen werden, da diese Methode Const ist

```
1 @Invariant
2 public class Person {
3
4    @Const
5    private Car car;
6
7    public Person(Car car) {
8       this.car = car;
9    }
10
11   public void getCarsPs() {
12      car.getPs();
13   }
14 }
```

Mehr zum Thema Const-Methoden findet man im folgenden Unterkapitel.

3.8.2 @Const-Methoden

Eine Methode, die mit *@Const* annotiert ist, darf *this* nicht verändern. Wenn eine Const-Methode eine andere Methode innerhalb der gleichen Klasse aufruft, so muss diese ebenfalls Const sein. Als Beispiel dient eine Personklasse.

Listing 17: setName(String name) führt zu einer Exception

```
1 @Invariant
2 public class Person {
3
4    private String name;
5
6    public Person(String name) {
7       this.name = name;
8    }
9
10   @Const public void setName(String name) {
11      this.name = name; //This wird verändert, obwohl Const gesetzt ist.
12   }
13 }
```

Innerhalb der Klasse dürfen aus einer Const-Methode nur Const-Methoden aufgerufen werden:

Listing 18: setShortName() führt zu einer Exception

```
1 @Invariant
2 public class Person {
3
4    @Const public void setShortName() {
5       calculate(); //Eine nonConst-Methode wird von einer Const-Methode
            aufgerufen.
6
7    private void calculate() {
8    }
9 }
```

Ein Spezialfall ergibt sich, wenn innerhalb einer Const-Methode eine Methode auf einem Datenfeld aufgerufen wird, die *this* übergibt. Als Beispiel *Person* und *Car*:

Listing 19: setOwner hat ein Cont-Parameter

```
1 @Invariant
2 public class Car {
3
4    public void setOwner(@Const Person person) {
5    }
6 }
```

Listing 20: setCarOwner übergibt this

```
1 @Invariant
2 public class Person {
3
4    private Car c;
5
6    public Person(Car c) {
7       this.c = c;
8    }
9
10   @Const public void setCarOwner() {
11      c.setOwner(this); //keine Exception, da Argument in setOwner() Const
            ist.
12   }
13 }
```

Wenn der Parameter der Methode setOwner(*@Const Person person*) in der Autoklasse nicht mit *@Const* annotiert wäre, würde *setCarOwner()* in der Personklasse eine Exception werfen. Mehr zum Thema Const-Parameter findet man im folgenden Unterkapitel.

3.8.3 @Const-Parameter

Wenn ein Argument eines Konstruktors oder einer Methode mit *@Const* annotiert ist, dürfen auf diesem Argument keine Änderungen vollzogen werden. Es dürfen auch nur *@Const*-Methoden auf diesem Argument aufgerufen werden. Als Beispiel dienen erneut die Klassen *Car* und *Person*:

Listing 21: Car ohne Const-Methoden

```
1 public class Car {
2    public int ps;
3
4    public void setPs(int ps) {
5       this.ps = ps;
6    }
7 }
```

Listing 22: Person mit Const-Parameter im Konstruktor

```
1 @Invariant
2 public class Person {
3    public Person(@Const Car c) {
4       c.setPs(330); //Exception, da Aufruf auf eine nonConst-Methode.
5       c.ps = 330; //Exception, da c verändert wird.
6    }
7 }
```

Folgendes Beispiel führt jedoch zu keiner Exception:

Listing 23: Car mit Const-Methode

```
1 @Invariant
2 public class Car {
3    public int ps;
4
5    @Const public int getPs() {
6       return ps;
7    }
8 }
```

Listing 24: Person mit Const-Parameter im Konstruktor

```
1 @Invariant
2 public class Person {
3    public Person(@Const Car c) {
4       c.getPs(); //Keine Exception, da getPs() const ist.
5    }
6 }
```

3.9 Parameterannotationen

Die Annotationen *@Min*, *@Max*, *@Range*, *@NotNull* dürfen auch auf Methoden- sowie Konstruktorparametern gesetzt werden. Es gelten dabei die gleichen Einstellungsmöglichkeiten, wie sie in den einzelnen Kapitel der Feldannotationen beschrieben sind.

Listing 25: Min-Parameterannotation

```
1  @Invariant
2  public class Person {
3      private String name;
4
5      public Person(@Min("1") String name) { //name muss mindestens ein
           Zeichen lang sein.
6          this.name = name;
7      }
8
9      public void setName(@Min("1") String name) { //name muss mindestens ein
           Zeichen lang sein.
10         this.name = name;
11     }
12 }
```

Die Invariantenüberprüfung findet bei Parameterannotationen beim Eintritt in eine Methode oder beim Aufruf eines Konstruktors statt (Pre-Condition).

3.10 Internes Logging

IAGen ist in der Lage, sich selbst zu loggen. Wenn man das IAGen-Projekt im Eclipse importiert hat, findet man unter *src.ch.iagen.aspects* den Aspekt: *Logger.aj*. Will man das interne Logging einstellen, so muss man den Aspect mit einschliessen:

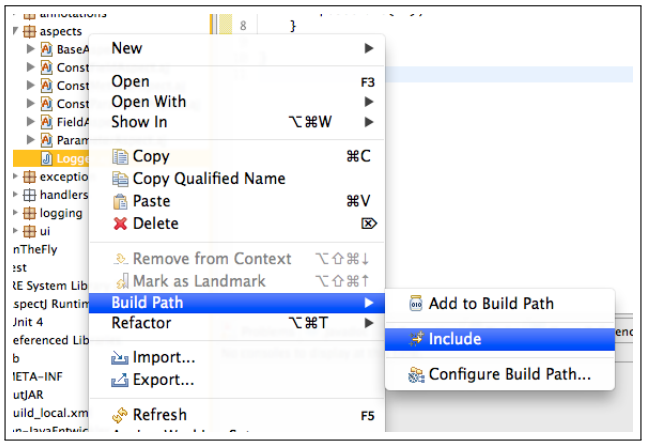


Abbildung 8: Rechter Mausklick auf Logger.aj -> Build Path -> Include

Startet man dann ein Programm innerhalb des Projektes, erstellt IAGen die Logdatei IAGen-Log.txt.
Beispielauszug des internen Loggings:

```
Sat Jun 02 13:55:00 CEST 2012 - staticinitialization of ch.iagen.aspects.BaseAspect.<clinit>
Sat Jun 02 13:55:00 CEST 2012 - staticinitialization of ch.iagen.aspects.ConstFieldAspect.<clinit>
```

Wenn man das interne Logging wieder ausschalten möchte, kann man den Aspect ausschliessen: Rechter Mausklick auf Logger.aj -> Build Path -> Exclude.

3.11 Kombinierte Beispiele

Natürlich lassen sich IAGen-Annotationen auch miteinander kombinieren. Dies soll anhand von zwei Beispielen verdeutlicht werden:

3.11.1 @Multiplicity mit @NotNull

Wenn eine *@Multiplicity*-Annotation mit der Untergrenze 0 gesetzt ist, so darf das annotierte Feld auch *null* sein. Möchte man dies verhindern, kann man das Feld mit *@Multiplicity* und mit *@NotNull* annotieren:

Listing 26: cars darf leer sein, nicht aber null

```
1  @Invariant
2  public class Person {
3
4      @NotNull
5      @Multiplicity(lower=0, upper=200)
6      List<Car> cars;
7
8      public Person() {
9          cars = new ArrayList<Car>();
10     }
11
12     public void addCarToPerson(Car c){
13         cars.add(c);
14     }
15 }
```

3.11.2 @Subsets mit @Multiplicity

Wenn das mit *@Subsets* annotierte Feld eine *Collection, Map oder Array [ ]* ist, kann man dieses auch zusätzlich mit *@Multiplicity* annotieren. Im folgenden Beispiel besteht ein Tournament zwischen 0 und 100 Teilnehmern. Es können zwischen 0 und 5 Spieler gewinnen, diese müssen jedoch in der *players*-Liste sein, also beim Tournament mitgespielt haben.

Listing 27: Subsets mit Multiplicity

```
1  @Invariant public class Tournament {
2
3      @Multiplicity(lower=0, upper=100)
4      List<Player> players;
5
6      @Multiplicity(lower=0, upper=5)
7      @Subsets("players")
8      List<Player> winners;
9
10     public Tournament() {
11         players = new ArrayList<Player>();
12         winners = new ArrayList<Player>();
13     }
14 }
```

4 Enterprise Architect Add-In

Das IAGen Enterprise Architect Add-In ist notwendig, um die modellierten Multiplizitäten und Subsets in die Tagged Values [tag] der bestehenden Attribute zu generieren. Diese werden anschliessend beim "Code Engineering" [cod] in den bestehenden Java Sourcecode generiert. Zusätzlich bietet es für die "Reverse Bytecode Engineering" Variante die Möglichkeit an, die modellierten Invarianten in eine XML-Datei zu exportieren.

4.1 Installation

Das setup.exe und der dazugehörige Microsoft Installer findet man unter folgendem Link:

05_Implementation/RX.X/setup.exe

05_Implementation/RX.X/IAGen-EA-Plugin.msi

Das setup.exe wird nur bei der erstmaligen Installation benötigt. Es prüft, ob das Microsoft .NET Framework installiert ist. Anschliessend kann immer der Microsoft Installer (.msi Datei) ausgeführt werden.

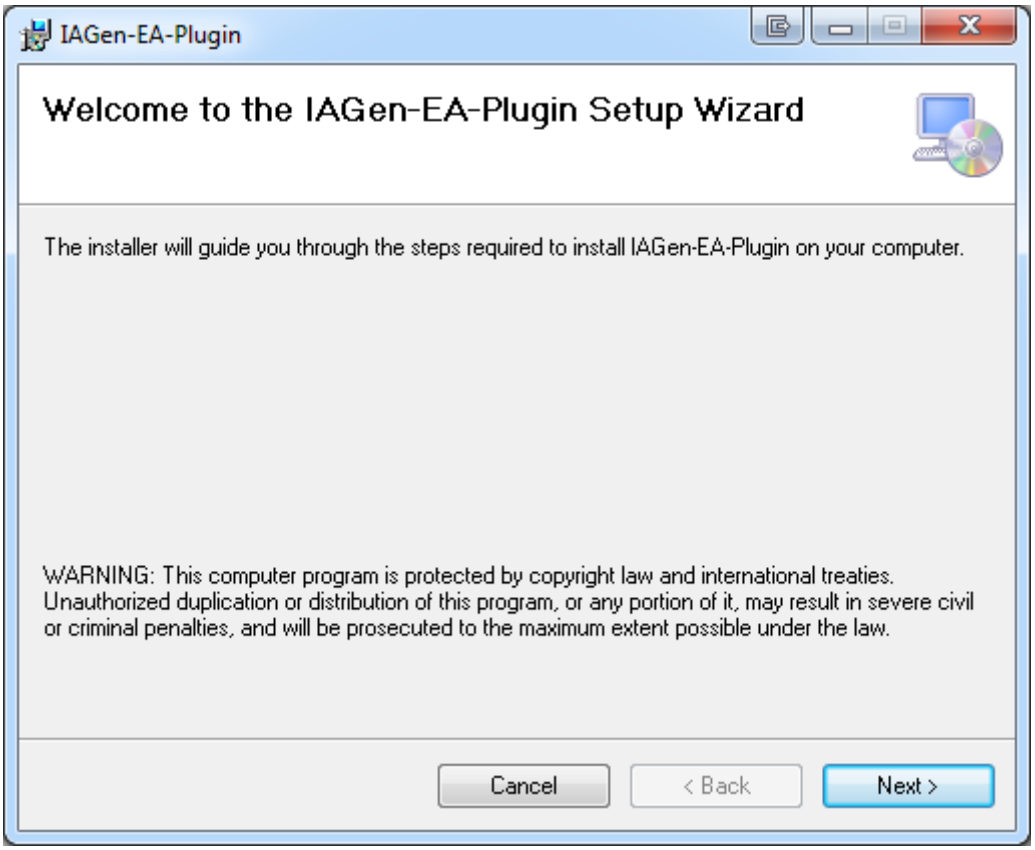


Abbildung 9: Installer für das IAGen Enterprise Architect Plugin

4.2 Wertebereiche im Enterprise Architect

Wertebereichannotationen können im Enterprise Architect über die Tagged Values eines Attributes gesetzt werden. Sie werden anschliessend bei der Generierung des Codes berücksichtigt.

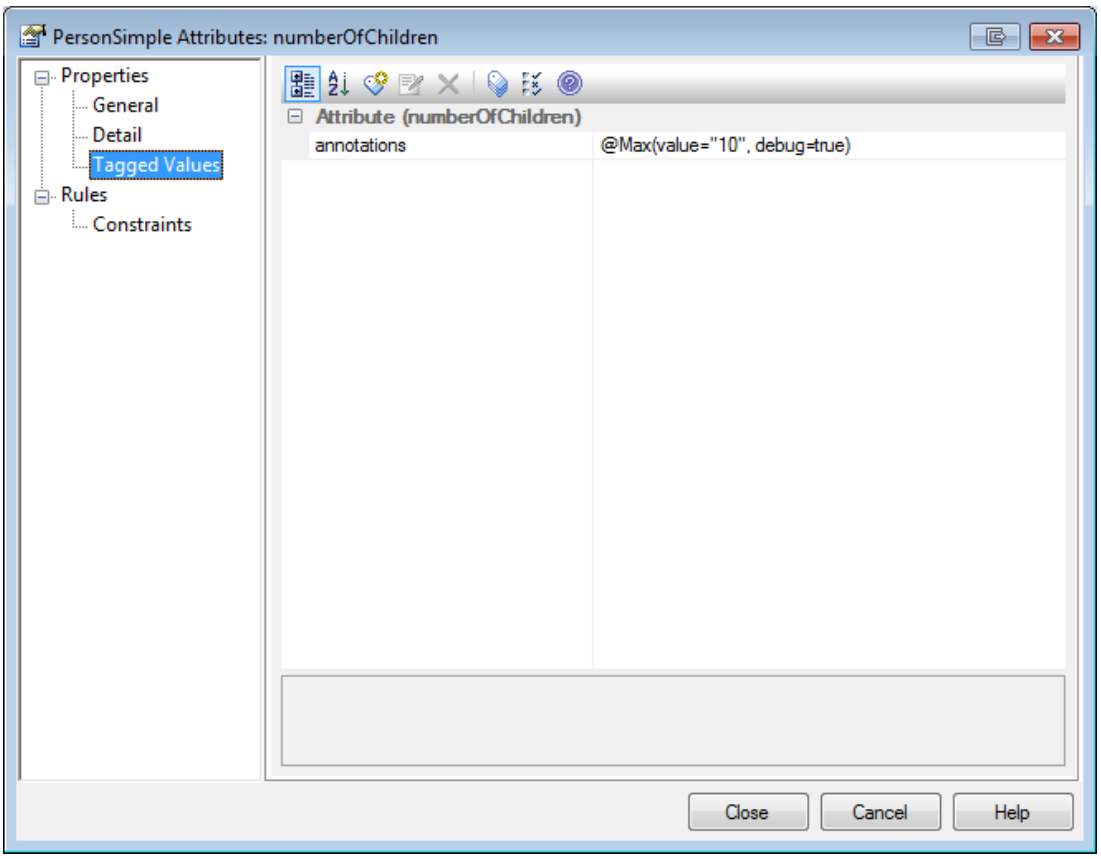


Abbildung 10: Das Setzen von Annotationen im Enterprise Architect

4.3 Anwendung des Add-Ins

Im Project Explorer des Enterprise Architect erscheint nach der Installation bei einem Rechtsklick auf ein Element das Menü Add-Ins. Wählt man dann IAGen, zeigt der EA folgendes Menü:

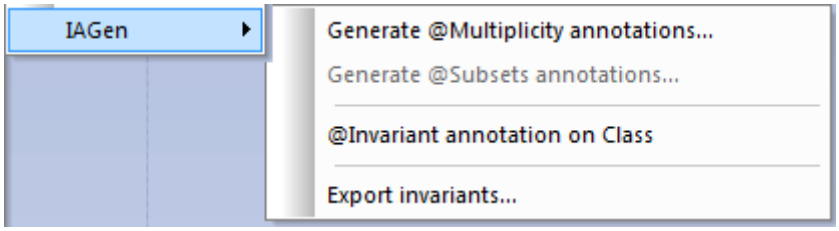


Abbildung 11: IAGen Add-In-Menü im Enterprise Architect

4.3.1 Generate @Mulitplicity annotations

Generiert die modellierten Multiplizitäten auf einer Assoziation. Dabei muss Folgendes beachtet werden:

- Die Multiplizität muss gesetzt sein (Beispiel: 1..10) und dabei eine der folgenden Formen haben:
  1..10
  1..1
  1
- Ein Rollenname muss gesetzt sein.

  Wenn der Rollenname mit dem betreffenden Attribut übereinstimmt, werden die Annotationen des Attributs aktualisiert. Ansonsten wird ein neues Attribut mit dem Rollennamen in der Klasse eingefügt.

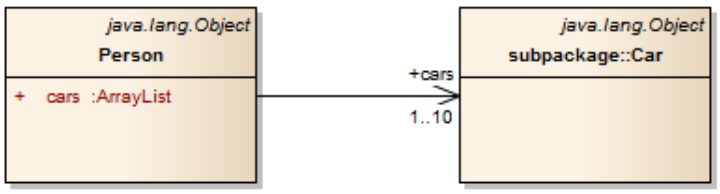


Abbildung 12: Der Rollenname stimmt mit dem Attribut überein.

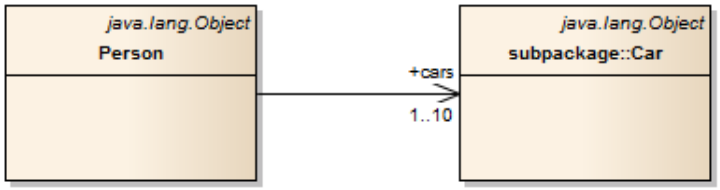


Abbildung 13: Ein neues Attribut *cars* wird kreiert.

- Die Verbindung zwischen den zwei Klassen muss vom Typ “Assoziation“ sein und eine der folgenden Directions haben:
  - Source -> Destination
  - Destination -> Source
  - Bi-Directional
- Das neu generierte Feld hat immer den Typ ArrayList<#TYPE#>. Dies kann jedoch über das Setzen der Java “Default Collection Class“ über Tools - Options - Sourcecode Engineering - Java - Collection Classes im Enterprise Architect geändert werden.

4.3.2 Generate @Subsets annotation

Generiert die modellierten Subsets auf einer Assoziation. Dabei muss Folgendes beachtet werden:

- Die Multiplizität muss gesetzt sein.
  Bei den Multiplizitäten 0..1, 1 und 1..1 generiert IAGen ein zusätzliches Feld mit dem referenzierten (subsetted) Typ. Bei Multiplizitäten wie 1..*, 1..10, 0..5 generiert IAGen eine neue Collection mit dem referenzierten (subsetted) Typ. Wird als Untergrenze nicht 0 angegeben, generiert IAGen zudem die *@NotNull* Annotation auf das Feld bzw. die Collection.
- Ein Rollenname muss gesetzt sein.
- Die Verbindung zwischen den zwei Klassen muss vom Typ Assoziation sein und eine der folgenden Directions haben:
  - Source -> Destination
  - Destination -> Source
  - Bi-Directional
- Das Constraint-Feld muss folgendes Format haben:
  subsets <*field-name*>

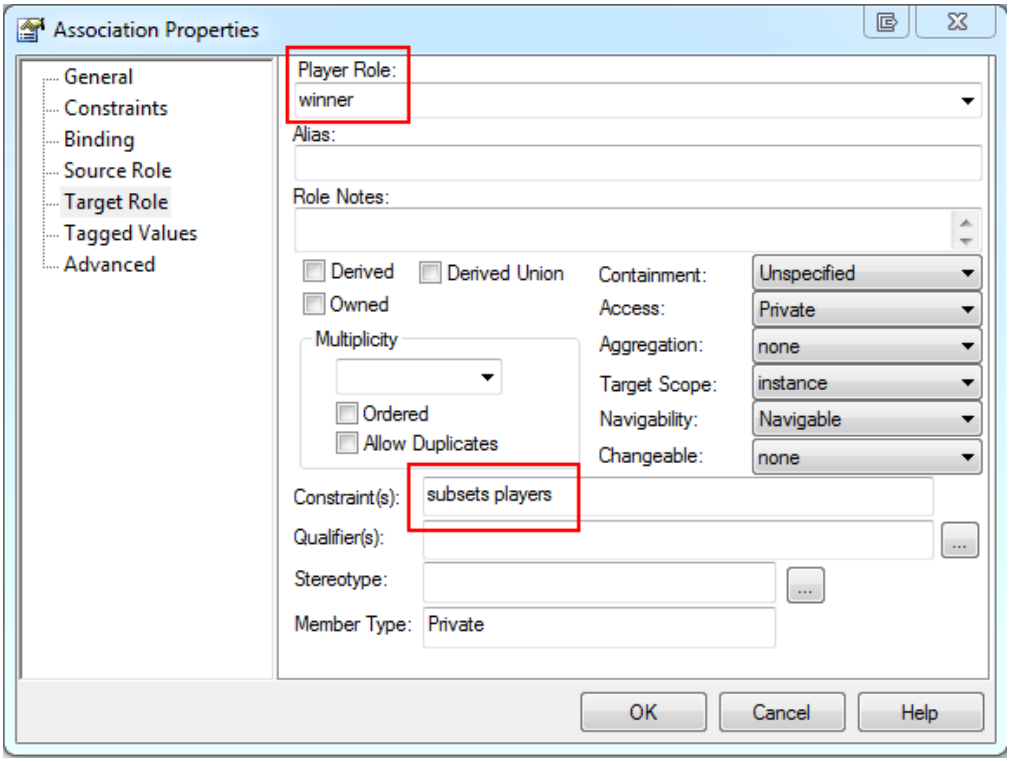


Abbildung 14: Erfassen eines subsets im Enterprise Architect

4.3.3 @Invariant annotation on Class

Damit die auf die Felder gesetzten Annotationen von IAGen (beispielsweise *@Max*) überprüft werden, muss bei der jeweiligen Klasse die *@Invariant*-Annotation gesetzt sein. Im IAGen Enterprise Architect Add-In kann dies mit dem Menüpunkt "*@Invariant* annotation on Class" erreicht werden. Ein Checkbox links vom Menüpunkt zeigt, ob die *@Invariant*-Annotation gesetzt ist oder nicht:

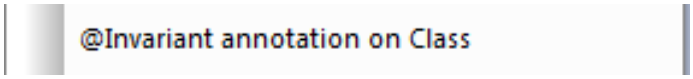


Abbildung 15: *@Invariant*-Annotation auf Klasse nicht vorhanden.

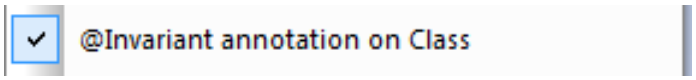


Abbildung 16: *@Invariant*-Annotation auf Klasse wurde gesetzt.

4.3.4 Export invariants

Mit dem Menüpunkt "Export invariants..." können die modellierten Änderungen in eine XML-Datei exportiert werden. Dies ist dann nützlich, wenn man bestehenden Java Bytecode reverse engineered hat und ein Forward Engineering somit unmöglich ist.

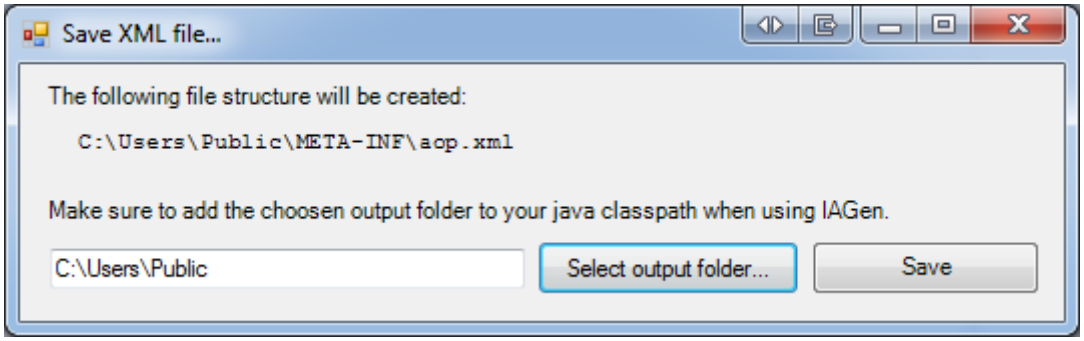


Abbildung 17: Exportieren der Invarianten

Im ausgewählten output folder wird folgende Dateistruktur angelegt:

<output-folder>\META-INF\aop.xml

Dieser <*output-folder*> muss dann zum Java Classpath des Programms (Bytecode), welches die IAGen Library verwenden, hinzugefügt werden:

- Windows

```
$ java -cp lib/IAGEn.jar;<output-folder>;. -javaagent:lib/IAGen.jar package.Main
```

- Mac OS X

```
$ java -cp lib/IAGEn.jar:<output-folder>:. -javaagent:lib/IAGen.jar package.Main
```

Die exportierte XML-Datei hat folgendes Format:

Listing 28: XML Beispieldatei nach Export

```
<aspectj>
  <aspects>
    <concrete-aspect name="ch.iagen.aspects.reverse.Annotater">
      <declare-annotation
      type="package.Person"
      annotation="@ch.iagen.annotations.Invariant"
      />
      <declare-annotation
      field="* package.Person.cars"
      annotation="@ch.iagen.annotations.Multiplicity(lower=1,upper=10)"
      />
    </concrete-aspect>
  </aspects>
  <weaver options="-Xreweavable -XlazyTjp -Xset:completeBinaryTypes=true"
      />
</aspectj>
```

Folgende XML-Elemente sind definiert:

- <*concrete-aspect*>
  Definiert den neuen Aspect, welcher die *declare* Statements enthält.
- <*declare-annotation*>
  Die modellierten Invarianten werden mit *declare* Statements im *concrete-aspect* festgelegt.
- <*weaver*>
  Weaver Optionen. [wea]

Diese XML-Datei kann nun mit einem Editor beliebig erweitert werden. Es ist nicht zwingend nötig, die Invarianten jedesmal im Enterprise Architect zu modellieren.

Literaturverzeichnis

[cod] Enterprise Architect Code Engineering. http://www.sparxsystems.com/enterprise_architect_user_guide/software_development/codeengineering.html.

[tag] Enterprise Architect Tagged Values. http://www.sparxsystems.com/uml_tool_guide/modeling_tool_features/thetaggedvaluestab.htm.

[wea] AsepctJ Weaver Options. http://www.eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html#weaver-options.

IAGen
Invariants-Assertion-Generation

Qualitätssicherung & Testauswertung

12. Juni 2012

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	08.06.2012	dmengelt	Erste Version & stan4j
1.0rc02	11.06.2012	f1egli	Unit Testing und Coverage
1.0rc03	11.06.2012	dmengelt	Jenkins und Zyklen
1.0	12.06.2012	dmengelt	Finale Version 1.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Review des Codes

Im Team haben wir oft Pair Programming eingesetzt. Diese Technik ermöglichte es uns, Fehler im Programmcode direkt bei der Entstehung zu erkennen. Dabei wurde ebenfalls gleich ein Code Review durchgeführt. Während des Pair Programmings wurden auch Tools wie Checkstyle, FindBugs etc. laufen gelassen. Die angezeigten Fehler wurden dann gemeinsam korrigiert.

4 Testing

4.1 Usability Test

Mit den Usability Tests wollten wir die Benutzung von IAGen testen. Hierbei stellten wir die Developer Guide, die IAGen Library und das Enterprise Architect Add-In externen Benutzern zur Verfügung. Selbständig und ohne fremde Hilfe mussten die Benutzer bestehenden Java Code in den Enterprise Architect importieren und IAGen-Invarianten modellieren. Anschliessend sollten sie das Add-In verwenden um die Invarianten zu speichern, um danach mit der Forward-Engineering-Funktionalität des Enterprise Architects die Annotationen in den Java-Sourcecode zu Generierieren. Zum Schluss musste die Library als Java Agent eingesetzt werden, um die Invarianten zur Laufzeit zu überprüfen. Wir konnten so wertvolle Informationen für die weitere Entwicklung von IAGen gewinnen. Beispielsweise ist so das Feature: “Generate @Multiplicity annotations. . . “ entstanden.

4.2 JUnit Testing für IAGen Library

Für das Testen unserer Hauptfunkionalitäten haben wir JUnit verwendet. Der Fokus wurde dabei auf die Packages *ch.iagen.handlers* und *ch.iagen.aspects* gelegt:

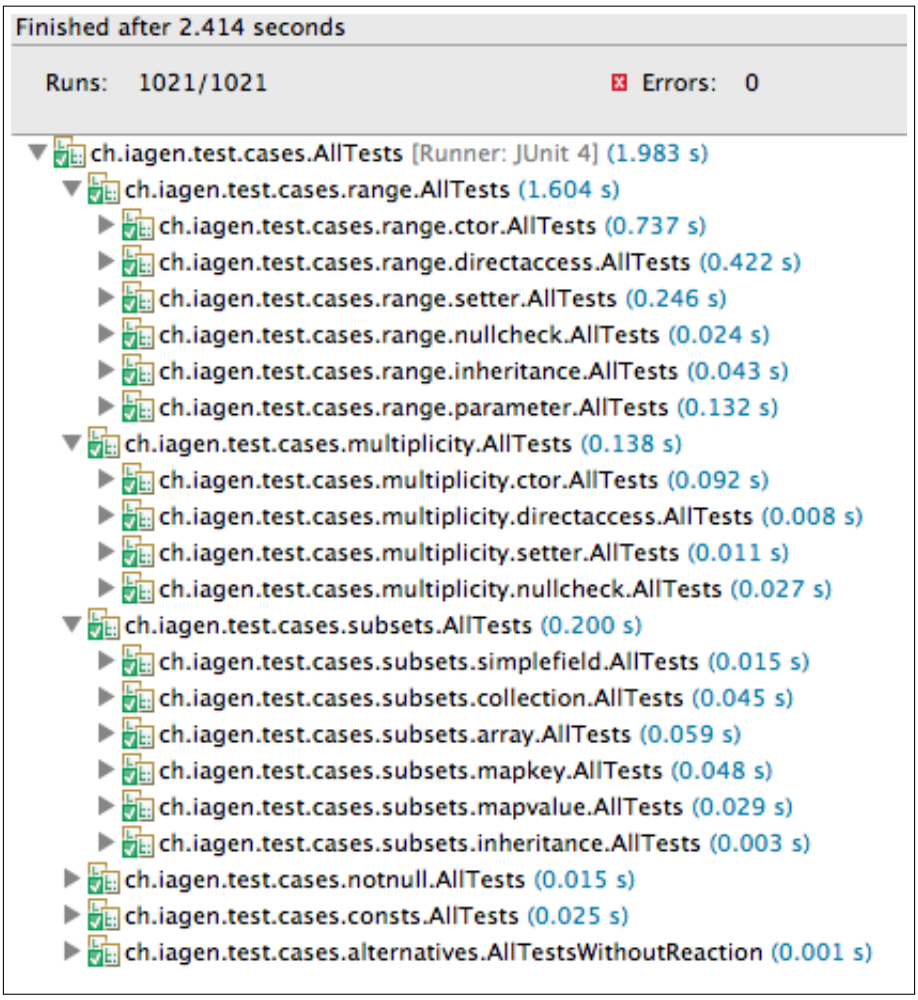


Abbildung 1: IAGen JUnit Tests

Durch die grosse Anzahl an Tests (über 940), konnten Refactoring Schritte und Design Änderungen im Sourcecode jederzeit überprüft werden.
Im späteren Teil des Projekts (nach dem Prototyp) wurden Tests geschrieben, bevor die Features überhaupt implementiert wurden. Durch diese Vorgehensweise wurde getestet, was verlangt wurde und nicht, was man vorgängig implementiert hat.

Dank der grossen Anzahl an Tests, konnte man auch Refactoring-Massnahmen für die Optimierungen der Performanz von IAGen überprüfen:

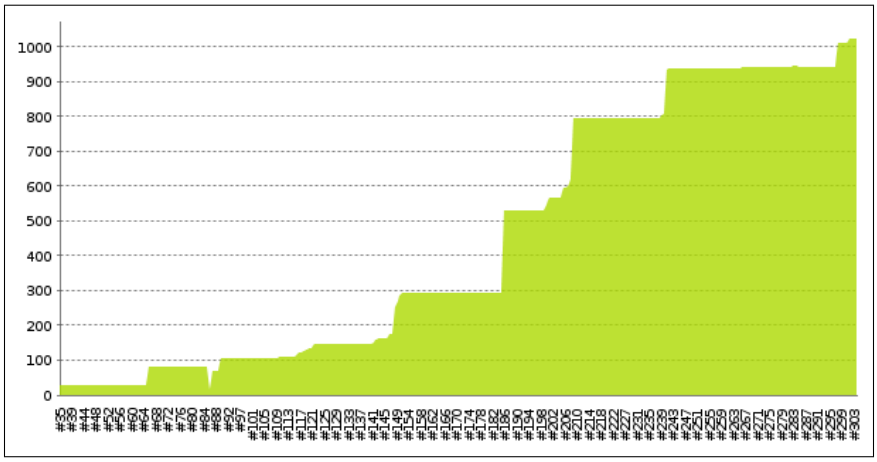


Abbildung 2: IAGen JUnit Tests - Anzahl pro Build

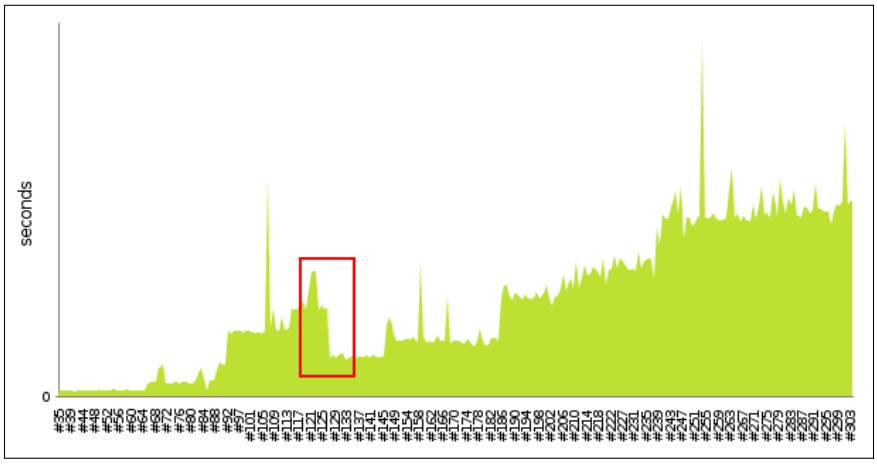


Abbildung 3: IAGen JUnit Tests - Dauer pro Build

Wie man in der Grafik sieht, wurde die Dauer pro Build massiv verkürzt (roter Kasten), obwohl die Anzahl Tests in dieser Zeit konstant blieben.

4.2.1 Testabdeckung - eCobertura

Schreibt man Tests, muss auch sichergestellt werden, dass alle Abdeckungen der wesentlichen Programmlogik getestet wird. Aus diesem Grund setzten wir das Testabdeckungsplugin eCobertura ein.

Name	Lines	Total	%	Branches	Total	%
All Packages (2012-06-12 17:53:36)	590	626	94.25 %	423	448	94.42 %
ch.iagen.aspects	157	177	88.70 %	91	108	84.26 %
ch.iagen.handlers.annotations	36	38	94.74 %	21	22	95.45 %
ch.iagen.handlers.annotations.container	57	65	87.69 %	21	24	87.50 %
ch.iagen.handlers.annotations.range	32	32	100.00 %	8	8	100.00 %
ch.iagen.handlers.type.multiplicity	46	46	100.00 %	24	24	100.00 %
ch.iagen.handlers.type.range	111	117	94.87 %	103	106	97.17 %
ch.iagen.handlers.type.subsets	151	151	100.00 %	155	156	99.36 %

Abbildung 4: IAGen Testabdeckung mit

Mit über 94% Abdeckung haben wir das gesetzte Ziel von 90% übertroffen. Die geringe Abdeckung im Package *ch.iagen.aspects* lässt sich dadurch erklären, das die Pointcuts nicht korrekt miteinberechnet werden. Die restlichen Abdeckungslücken sind meist fehlende *catch* Durchläufe (Reflection-Zugriffe).

5 Sonstige Tools und Hilfsmittel

5.1 FindBugs

FindBugs wurde jeweils ohne Konfigurationsänderungen durchlaufen, da die Grundeinstellungen effizient Fehler im Code aufzeigt. Durch die Anwendung dieses Plugins konnten die Bugs im Projekt stark reduziert werden. Zwei Bugs konnten nicht behoben werden:

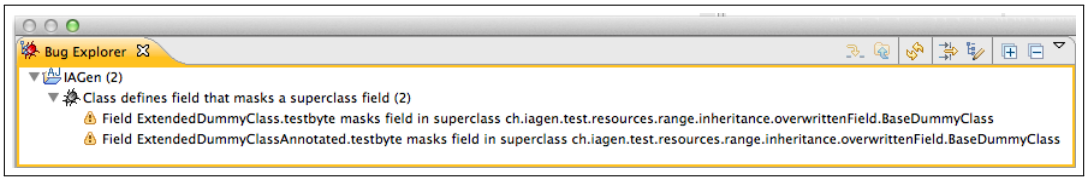


Abbildung 5: Bug Explorer Ansicht

Wir haben diese beiden Bugs jedoch bewusst offen gelassen, weil die Tests im Package *ch.iagen.test.resources.range.inheritance.overwrittenField.** genau diese Problematik beim Überschreiben von vererbten *public*-Feldern testen soll.

5.2 CheckStyle

Um den Style des Sourcecodes zu verbessern haben wir Checkstyle eingesetzt. Unnötige Leerschläge oder falsche Modifiers konnten so verhindert werden. Folgende Packages wurden nicht mit Checkstyle geprüft:

- *ch.iagen.ui*
  ch.iagen.test

  Wurden nicht mit Checkstlye geprüft, weil diese Packges viele berechtigte Magic Numbers enthalten, welche als "Violations" gelten.

5.3 stan4j

Mit dem Structure Analysis Tool stan4j konnten wir zwei zyklischen Abhängigkeiten erkennen. Diese wurden mit der Einführung des *IReaction* Interfaces und leichten Umstellungen bei der Package-Struktur eliminiert. IAGen hat somit weder vertikale noch horizontale zyklische Abhängigkeiten. Weitere Informationen zur Package-Struktur können dem Dokument "Archtitektur und Design" entnommen werden:

04_Architektur_und_Design/Architektur_und_Design_vX.X.pdf

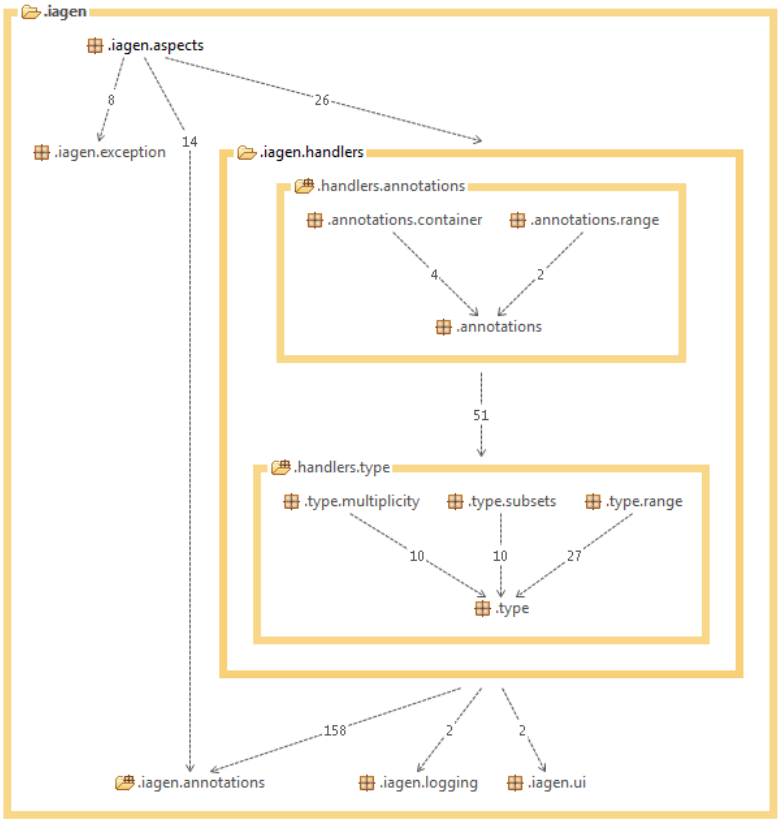


Abbildung 6: IAGen ohne zyklische Abhängigkeiten

5.4 Logging

Das Logging der IAGen Library kann über einen separaten Aspect eingeschaltet werden. Beispielauszug des Loggings:

```
Sat Jun 02 13:55:00 CEST 2012 - staticinitialization of ch.iagen.aspects.BaseAspect.<clinit>
Sat Jun 02 13:55:00 CEST 2012 - staticinitialization of ch.iagen.aspects.ConstFieldAspect.<clinit>
```

Weiter Informationen zum internen Logging können dem Dokument "Developer Guide" entnommen werden:

04_Architektur_und_Design/00_Developer_Guide/Developer_Guide_vX.X.pdf

6 Code-Statistiken

Werte bei "Anzahl Linien im Code" sind sinnvoll gerundet.

IAGen Library	Werte
Anzahl Linien im Code insgesamt (inkl. Tests)	15000
Anzahl Linien im Code insgesamt (ohne Tests)	2000
Methoden	134
Klassen	42
Packages	13

IAGen Add-In	Werte
Anzahl Linien im Code insgesamt	800

6.1 Anzahl automatische Builds

Nach jedem Commit werden sowohl die IAGen Library als auch das IAGen Add-In neu gebildet. Während des gesamten Projekts wurde dies über 300 mal für die Library und 210 mal für das Add-In gemacht:

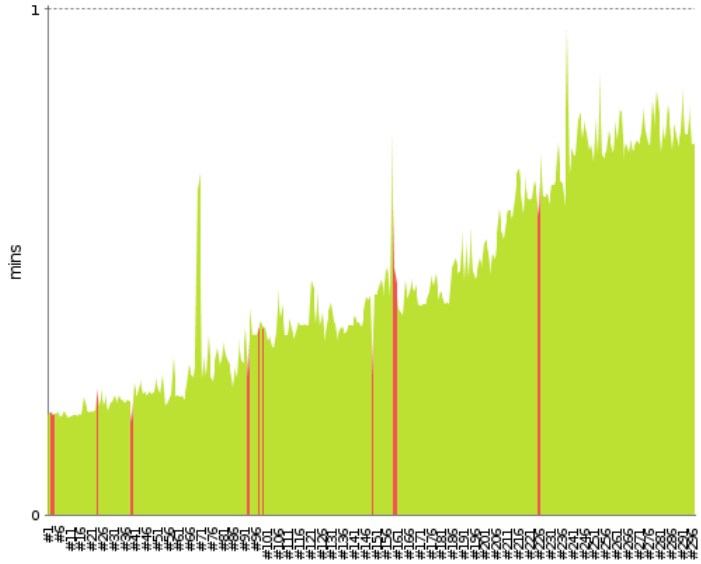


Abbildung 7: Anzahl Builds der IAGen Library

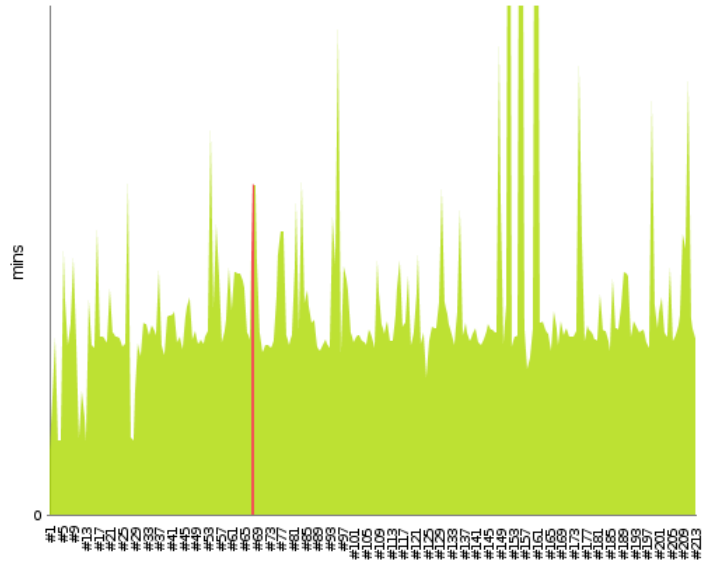


Abbildung 8: Anzahl Builds des IAGen Add-Ins

IAGen
Invariants-Assertion-Generation

Tools und Infrastruktur

12. Juni 2012

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	23.05.2012	dmengelt, f1egli	GIT Repositories und ANT
1.0	24.05.2012	f1egli	GIT Befehle
2.0rc01	02.06.2012	f1egli	Anpassungen GIT Repositories
2.0rc02	02.06.2012	dmengelt	Anpassungen ANT, msbuild
2.0rc03	07.06.2012	dmengelt	stan4j
2.0	08.06.2012	dmengelt	Finale Version 2.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Infrastruktur und Werkzeuge

3.1 GIT

3.1.1 GIT Installation

Auf dem Server kann mittels:

`sudo aptitude install git`

die neuste git Version installiert werden.

Clientseitig variiert die Installation je nach Betriebssystem:

- Mac OS X: `http://code.google.com/p/git-osx-installer/`
- Windows: `http://code.google.com/p/msysgit/downloads/list`

3.1.2 Neues GIT-Repository erstellen

Auf dem Server iagen.ch gibt es das Verzeichnis *"/var/gitrepos/"*. Darin können neue GIT-Repositories mit den folgenden Befehlen erstellt werden:

Listing 1: Neue GIT-Repo

```
1  [root@IAGen] ~
2   # cd /var/gitrepo/
3  [root@IAGen] /var/gitrepo
4   # mkdir Test.git
5  [root@IAGen] /var/gitrepo
6   # cd Test.git/
7  [root@IAGen] /var/gitrepo/Test.git
8   # git init  --bare --shared=group
9  Initialized empty shared Git repository in /var/gitrepo/Test.git/
10 [root@IAGen] /var/gitrepo/Test.git
11  #
```

3.1.3 GIT Repositories Clonen

Auf dem Server befinden sich drei GIT-Repositories:

- impl.git
- doc.git
- scratch.git

Das *impl* Repository enthält die gesamte Implementation von IAGen. Diese besteht aus der Java Library (IAGen.jar) und dem Enteprise Architect Add-In (IAGen.dll). Das *doc* Repository enhält die Dokumentation von IAGen. Für experimente und temporäre Tests oder Testprojekte dient das *scratch* Repository.

Um die Repositories lokal zu erstellen verwendet man den Befehl git clone.

```
git clone ssh://<user>@iagen.ch/var/gitrepo/impl.git
git clone ssh://<user>@iagen.ch/var/gitrepo/doc.git
git clone ssh://<user>@iagen.ch/var/gitrepo/scratch.git
```

3.1.4 Pull und Push

Hat man die Repositories mittels clone erstellt, kann man mit git pull jederzeit die aktuellsten Files von dem Server beziehen.

```
cd /path/to/repository
git pull
```

Möchte man lokale Änderungen auf den Server speichern benötigt es folgende Befehle:

```
git add .
git commit -am "updated DeveloperGuide"
git push
Counting objects: 18, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (13/13), 210.80 KiB, done.
Total 13 (delta 3), reused 0 (delta 0)
To ssh://<user>@iagen.ch/var/gitrepo/impl.git
   633cdee..bccc57b  master -> master
```

3.1.5 Nützliche Befehle

Lokale Änderungen verwerfen:
Um die lokalen Änderungen zu verwerfen verwendet man am einfachsten die folgenden Befehle:

```
git checkout -- <file>
git pull
```

Differenzen anzeigen:
Falls man die Differenzen zwischen einem lokalen File und dem Server File anzeigen will, kann man den Befehl git diff wie folgt verwenden.

```
git diff <file>
```

Remote Branch anlegen:

```
git branch
* Testing
master
git push origin Testing
Counting objects: 39, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (22/22), 3.85 KiB, done.
Total 22 (delta 5), reused 0 (delta 0)
To ssh://<user>@iagen.ch/var/gitrepo/impl.git *
[new branch] Testing -> Testing
```

Entwickler, welche diesen Branch dann wollen:

```
git pull
//TEXT OMITTED
git checkout -b Testing origin/Testing
Branch Testing set up to track remote branch refs/remotes/origin/Testing.
Switched to a new branch "Testing"
```

Lokaler Branch löschen:

```
git branch
  Testing
* Master
git branch -d Testing
Deleted branch Testing (was 4460739).
```

Remote Branch löschen:

```
git push origin :Testing
To ssh://<user>@iagen.ch/var/gitrepo/impl.git
 - [deleted]         Testing
```

Tag anlegen:

```
git tag -a v1.1 -m "prototype release v1.1"
git push origin v1.1
```

3.2 Jenkins

Jenkins installiert man über apt:

```
sudo aptitude install jenkins
```

Danach kann der Server gestartet werden:

```
sudo /etc/init.d/jenkins start
```

Die weiteren Konfigurationen können über einen Webbrowser getätigt werden:

`http://iagen.ch:8080`

3.3 Redmine

Die Projektmanagement Software Redmine kann wie folgt auf einem System mit Debian installiert werden:

```
sudo aptitude install redmine redmine-mysql apache2 mysql-server
libapache2-mod-passenger
```

Ein Apache2 VirtualHost wird benötigt:

```
<VirtualHost *:80>
    DocumentRoot /usr/local/lib/redmine-1.2/public
    <Directory /usr/local/lib/redmine-1.2/public>
        AllowOverride None
    </Directory>
    <Directory "/usr/local/lib/redmine-1.2/public/downloads">
        Options +Indexes
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

Weitere Konfigurationen können direkt in Redmine vorgenommen werden.

3.4 ANT

Mittels ANT und Jenkins wird nach jedem Push auf das Git Repository ein automatischer Build der IAGen Library ausgeführt.

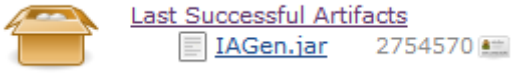


Abbildung 1: Artefakt IAGen.jar

Es existieren folgende Targets, welche nacheinander ausgeführt werden:

Listing 2: ANT build_local.xml Targets

```
1 <target name="compile">
2 <target name="tests">
3 <target name="publish">
```

3.4.1 Target compile

Kompiliert die Sourcefiles mittels ajc und erstellt eine jar Library (outjar). Zusätzlich werden mittels *zipgroupfileset* die Dateien des AspectJ Weavers in die Library kopiert und die Service Einträge für das Annotation Processing gemacht. Die im Projektordner *META-INF* erstellten aop.xml Dateien (aop.xml und const.xml) werden automatisch mit *zipfileset* in das erstellte jar kopiert.

Listing 3: ANT Target compile

```
1  <target name="compile">
2     <iajc outjar="./outJAR/IAGen.jar" source="1.6" target="1.6">
3        <src location="src" />
4        <src location="test" />
5        <exclude name="**/Logger.aj" />
6
7        <classpath>
8           <pathelement location="./lib/aspectjrt.jar" />
9           <pathelement location="./lib/junit.jar" />
10          <pathelement location="./lib/aspectjweaver-dev.jar" />
11       </classpath>
12    </iajc>
13
14    <echo message="created ./outJAR/IAGen.jar" />
15    <echo message="creating annotation service entry within META-INF/
          services" />
16
17    <jar destfile="./outJAR/IAGen.jar" update="true">
18
19       <service type="javax.annotation.processing.Processor">
20          <provider classname="ch.iagen.annotations.processors.
                RangeProcessor" />
21          <provider classname="ch.iagen.annotations.processors.
                InvariantProcessor" />
22       </service>
23
```

```
24        <zipgroupfileset dir="lib" includes="aspectjweaver*.jar" />
25        <zipfileset dir="META-INF" includes="*.xml" prefix="META-INF"/>
26
27        <manifest>
28           <attribute name="Premain-Class" value="org.aspectj.weaver.loadtime
                .Agent" />
29        </manifest>
30
31     </jar>
32
33     <antcall target="tests" />
34 </target>
```

3.4.2 Target tests

Führt die JUnit Tests direkt aus dem erstellten Jar File (IAGen.jar) aus.

Listing 4: Ausführung der Tests

```
1  <target name="tests">
2     <junit fork="yes" printsummary="withOutAndErr" haltonfailure="yes"
          showoutput="true">
3
4        <formatter type="xml" />
5        <classpath>
6           <pathelement location="./lib/aspectjrt.jar" />
7           <pathelement location="./lib/junit.jar" />
8           <pathelement location="./lib/hamcrest.jar" />
9           <pathelement location="./outJAR/IAGen.jar" />
10       </classpath>
11
12       <batchtest fork="yes" todir="./outJAR">
13          <zipfileset src="./outJAR/IAGen.jar">
14             <include name="**/iagen/test/cases/AllTests.class" />
15          </zipfileset>
16       </batchtest>
17
18    </junit>
19
20    <echo message="ran all tests within the iagen/test directory" />
21    <antcall target="publish" />
22 </target>
```

3.4.3 Target publish

Kopiert die IAGen Library in die Verzeichnisse der Java Projekte, welche die Library testen/verwenden:

Listing 5: Ausführung der Tests

```
1  <target name="publish">
2     <copy file="./outJAR/IAGen.jar" tofile="../IAGen-JavaEntwickler/lib/
          IAGen.jar" />
3     <copy file="./outJAR/IAGen.jar" tofile="../EAPluginTest/lib/IAGen.jar"
          />
4  </target>
```

3.5 msbuild

Für das Enterprise Architect Add-In wird msbuild [msb] und Jenkins eingesetzt. Werden Änderungen an der Implementation vorgenommen erstellt Jenkins mittels msbuild eine neue Version der IAGen.dll.

Abbildung 2: Artefakt IAGen.dll

3.6 Enterprise Architect

Während der Entwicklungszeit wurde die Version 9 des Enterprise Architect eingesetzt.

`http://www.sparxsystems.com/products/ea/`

3.7 Eclipse Plugins

3.7.1 eCobertura

Die neuste Version von eCobertura kann im Eclipse unter Help->Install New Software.. installiert werden. Die "Work with" URL lautet:

`http://ecobertura.johoop.de/update/`

Zur Entwicklungszeit wurde die Version 0.9.8 verwendet. Diese befindet sich unter:

`Software/EclipsePlugins/eCobertura/`

Bei offenen Fragen loht es sich, die Hersteller Webseite zu kontaktieren:

`http://ecobertura.johoop.de/`

3.7.2 FindBugs

Die neuste Version von FindBugs kann im Eclipse unter Help->Install New Software.. installiert werden. Die "Work with" URL lautet:

`http://findbugs.cs.umd.edu/eclipse/`

Zur Entwicklungszeit wurde die Version 2.0.0.20111221 verwendet. Diese befindet sich unter:

`Software/EclipsePlugins/FindBugs/`

Bei offenen Fragen lohnt es sich, die Hersteller Webseite zu kontaktieren:

`http://findbugs.sourceforge.net/`

3.7.3 CheckStyle

Die neuste Version von CheckStyle kann im Eclipse unter Help->Install New Software.. installiert werden. Die "Work with" URL lautet:

`http://eclipse-cs.sf.net/update/`

Das IAGen XML-Konfigurationsfile findet man unter:

`02_QM/01_Checkstyle/cs.xml`

Zur Entwicklungszeit wurde die Version 5.5.0.201111092104 verwendet. Diese befindet sich unter:

`Software/EclipsePlugins/Checkstyle/`

Bei offenen Fragen lohnt es sich, die Hersteller Webseite zu kontaktieren:

`http://checkstyle.sourceforge.net/`

3.7.4 stan4j

Die neuste Version von stan4j kann im Eclipse unter Help->Install New Software.. installiert werden. Die "Work with" URL lautet:

`http://update.stan4j.com/ide`

Zur Entwicklungszeit wurde die Version 2.0.3 verwendet. Diese befindet sich unter:

`Software/EclipsePlugins/stan4j/`

Bei offenen Fragen lohnt es sich, die Hersteller Webseite zu kontaktieren:

`http://http://stan4j.com/`

Literaturverzeichnis

[msb] MSBuild Reference. http://msdn.microsoft.com/en-us/library/0k6kkbsd.

IAGen Release Notes
Release: 1.0 [Prototype]

1. New Features
Feature #46 EA - Generation of attribute tagged values to xml files
Feature #45 Annotations @Min, @Max on primitive data types (and wrapper)

2. Defects Fixed
None

3. Known Issues
-

IAGen Release Notes
Release: 2.0

1. New Features
Feature #48: Annotation @Range on primitive attributes (and wrapper).
Feature #49: char and Character for Range (@Range, @Min, @Max).
Feature #50: Popup with additional stack trace information.
Feature #52: @Invariant marker annotation for better performance.
Feature #53: Create multiplicitiy annotations within Enterprise Architect.
Feature #54: Export modeled multiplicities and ranges within Enterprise Architect to aop.xml file.
Feature #55: set a @Min, @Max, @Range annotated field with a character unicode value.
Feature #57: Annotation processing for @Min, @Max and @Range.
Feature #58: Menu to create @Invariant annotation wihtin Enterprise Architect.

2. Defects Fixed
None

3. Known Issues
- Custom Interfaces (@interface) are not getting imported proberly. Enterprise Architect removes the "@".
- IAGen Annotations are not inherited to subclasses.

IAGen Release Notes
Release: 3.0 [final]

1. New Features
Feature #59: Annotation @Subsets on Fields, Maps, Collection and Arrays.
Feature #60: Annotation @NotNull on Fields and Parameters (Method, Constructor).
Feature #61: Annotation @Const on Fields, Methods and Parameters (Method, Constructor).
Feature #62: Annotations @Min, @Max and @Range on Parameters (Method, Constructor).
Feature #63: Annotation processing for @Subsets and @Multiplicity.
Feature #64: Exception can be specified on an annotation (except @Const).
Feature #65: Exception message can be specified on an annotation (except @Const).
Feature #66: Broken invariants can be logged into a specified log file (except @Const).
Feature #67: IAGen internal logging.
Feature #68: Create subsets annotations within Enterprise Architect.
Feature #69: Export modeled subsets within Enterprise Architect to aop.xml file.

2. Defects Fixed
- @Invariant annotation is now inherited to subclasses.

3. Known Issues
- Eclipse is not able to show APT note messages within code editor (only Warnings and Errors)
- No invarant assertion on super.method() calls within methods.

IAGen
Invariants-Assertion-Generation

Zeitauswertung

13. Juni 2012

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	09.06.2012	dmengelt	Erste Version der Zeiterfassung
1.0rc02	11.06.2012	dmengelt	Bar Chart hinzugefügt
1.0	12.06.2012	dmengelt	Finale Version 1.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

3 Auswertung Arbeitswochen

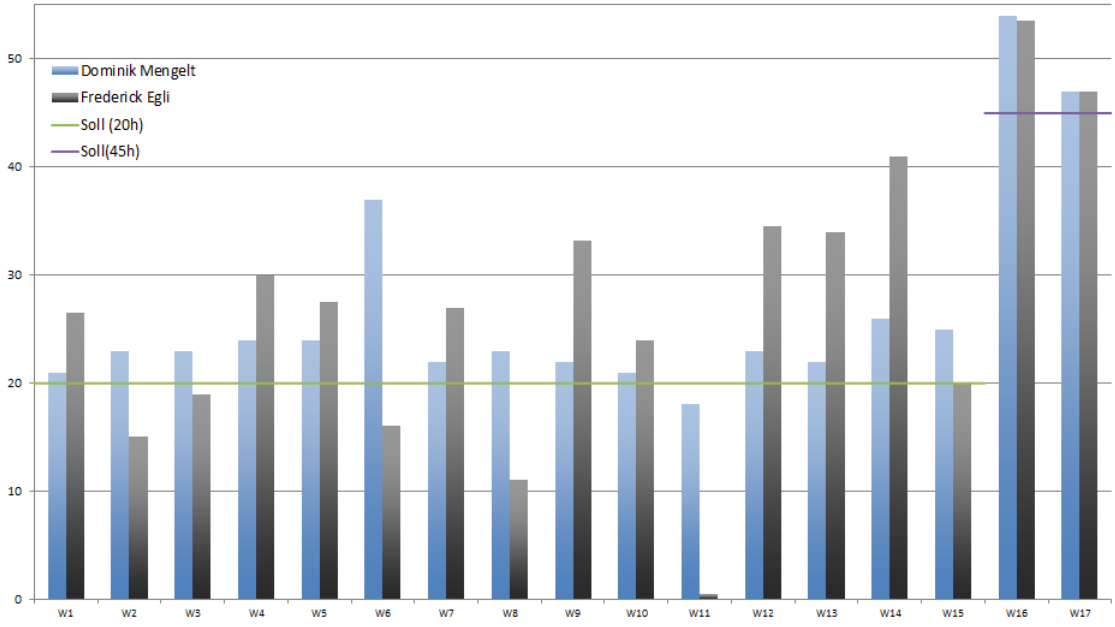


Abbildung 1: Anzahl Stunden pro Semesterwoche

Anmerkung: Aufwände wurden in der letzten Woche (W17) geschätzt.

3.1 Auswertung Woche 1-15

Soll: 20 Stunden

Mitglied	Aufgewendete Stunden (Ist-Wert) pro Woche im Durchschnitt
Frederick Egli	23.95
Dominik Mengelt	23.6

3.2 Auswertung Woche 16 & 17

Soll: 45 Stunden

Mitglied	Aufgewendete Stunden (Ist-Wert) pro Woche im Durchschnitt
Frederick Egli	50.25
Dominik Mengelt	50.5

4 Auswertung pro Kategorie

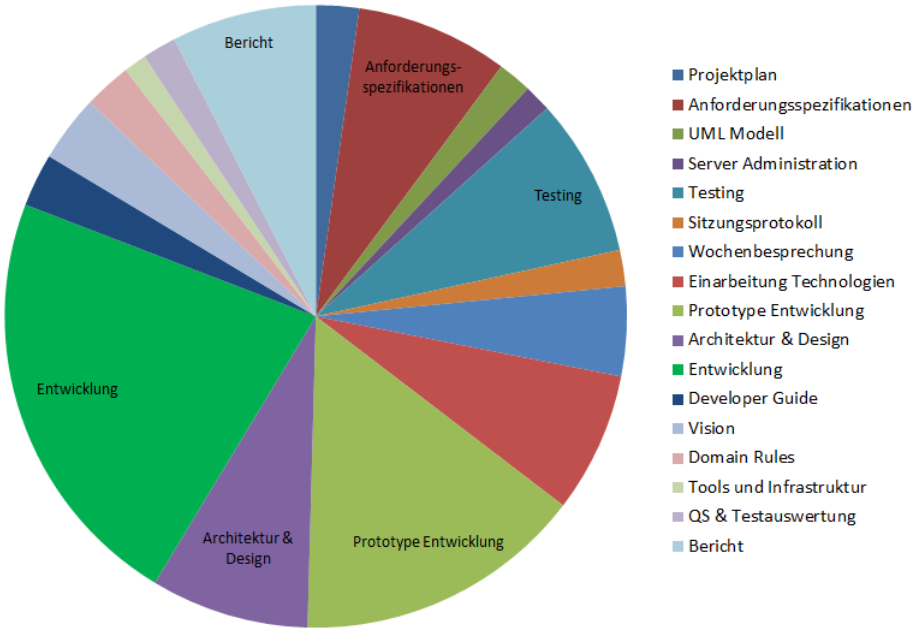


Abbildung 2: Zeitauswertung pro Kategorie

Total wurden 914.7 Stunden für das Projekt aufgewendet. Dabei haben folgende Arbeitspakete die meiste Zeit beansprucht:

Arbeitspaket	Aufwand in %	Aufwand in Stunden
Entwicklung	22	188.7
Prototype Entwicklung	15	128
Architektur und Design	8	70
Anforderungsspezifikationen	8	67.5
Bericht	8	64

IAGen
Invariants-Assertion-Generation

Glossar

12. Juni 2012

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	29.03.2012	dmengelt	Erste Begriffe definiert
1.0rc02	29.03.2012	dmengelt	Weitere Begriffe
1.0	03.05.2012	dmengelt	Anpassungen
2.0rc01	29.05.2012	dmengelt	Weitere Begriffe
2.0	11.06.2012	dmengelt	Finale Version 2.0

http://www.iagen.ch

Inhaltsverzeichnis

2 Ziel und Zweck

Siehe Kapitel Artefaktenliste im Projektplan.

Glossar

Add-In	Add-Ins (Extension, Plugin) ermöglichen es, den Enterprise Architect mit Funktionalitäten zu erweitern.
Advice	Ein Advice ist im Bezug auf einen Pointcut definiert. Der Programmcode eines Advices läuft immer dann ab, wenn ein definierter Pointcut zu einem JoinPoint passt. Es gibt drei Typen: Before, After, Around.
After	Ist ein Advice Typ. Es wird nach der eigentlichen Operation getraced.
Artefakt	Bezeichnet ein Dokument oder Codestück.
AspectJ	AspectJ erweitert Java, damit aspekt-orientierte Programmierung möglich wird. Dadurch lassen sich generische Funktionalitäten über mehrere Klassen verwenden.
Auftraggeber	Ist die Bezeichnung des direkten Kunden von IAGen. Zusammen mit dem Auftraggeber wurden die Anforderungen ausgearbeitet.
Before	Ist ein Advice Typ. Es wird vor der eigentlichen Operation getraced.
Container	Ein Container bezeichnet eine Java Collection, Map oder Array.
EA	Abkürzung für Enterprise Architect.
EA Entwickler	Modelliert Invarianten im Enterprise Architect und generiert anschliessend den Code.
Entwickler	Oberbegriff der Endanwender von IAGen. Spezialisierungen sind: EA-Entwickler und Java-Entwickler.
grosser Container	Wird bei Subsets als Wert angegeben. Dieser Container ist die grosse Menge.
Invariante	Die Definition der Invariante ist im Dokument Domain Rules zu finden.
Java-Entwickler	Annotiert direkt den Java-Sourcecode. Ohne die Verwendung des Enterprise Architect.
JoinPoint	Ein JoinPoint ist der Teil, welcher gerade im Kontrollfluss des Programms ist.
kleiner Container	Darauf wird die Subsets-Annotation gesetzt. Dieser Container muss vollständig in dem grossen Container vorhanden sein.
LTW	Load Time Weaving. Ermöglicht das Verweben von AspectJ Aspects mit bereits kompiliertem Java Code zur Ladezeit.
Pointcut	Ein Pointcut ist ein Programmelement, welches basierend auf definierten Bedingungen, Advices ausführt.
rc	Release Candidate.