HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

JaInFra

Java Inspection Framework

Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2011

Autor(en): Dominik Mengelt, Frederick Egli
Betreuer: Thomas Letsch

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.

Rapperswil, 12.12.2011

Frederick Egli

Dominik Mengelt

JaInFra
Abstract

December 22, 2011

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	04.12.2011	f1egli	Dokument erstellt
1.0rc02	09.12.2011	f1egli	AspectJ besser erklärt
1.0	12.12.2011	f1egli	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Abstract **3**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Abstract

Das Java Inspection Framework, kurz JaInFra, ist eine API, die es ermöglicht, kompilierten Code zu analysieren und zu verändern, ohne den originalen Sourcecode zu besitzen. Die API muss lediglich als JAR File in den eigenen Code eingebunden werden. Danach sind keine weiteren Zusatzpakete erforderlich.
Intern arbeitet JaInFra mit AspectJ und Reflection. Letzteres wird benötigt, um statische Analysen über das unbekannte Programm zu erstellen. Somit hat man Zugriff auf die Methoden und Felder einer kompilierten Klasse.
AspectJ erweitert Java, damit aspekt-orientierte Programmierung möglich wird. Dadurch lassen sich generische Funktionalitäten über mehrere Klassen verwenden. Das Verweben dieser Funktionalitäten mit dem eigentlichen Code, nennt sich weaving. In JaInFra wird AspectJ benötigt, um das Programm während der Laufzeit zu analysieren und gegebenenfalls zu verändern. Da AspectJ kein Runtimeweaving unterstützt, benutzt JaInFra das Loadtimeweaving. Bevor das unbekannte Programm mittels Reflection in einem eigenen Thread gestartet wird, "weavt" sich JaInFra in praktisch jeden erdenklichen Abschnitt des Programms ein (Beispiel: vor einem Methodenaufruf und danach).
Der Kunde kann JaInFra über sogenannte Filter mitteilen, was er überhaupt Inspizieren möchte. Die Filter wurden mittels dem Interpreter Pattern erstellt. Somit lassen sich beliebig viele Konstrukte bilden, um ein Maximum an Flexibilität zu gewährleisten. Ist ein Filter während der Programmausführung befriedigt, wird ein Callback zum Kunden ausgeführt. Der Kunde kann dabei selber entscheiden, was bei einem Callback passiert, in dem er das entsprechende Interface ausprogrammiert. So ist es zum Beispiel möglich, die gewünschten Trace Inhalte, wie Signaturnamen und Argumentinhalte, in einem File abzuspeichern.
Die Injection geschieht ebenfalls mit Hilfe von Filtern und Callback Interfaces. Dabei können zwei unterschiedliche Injektionsverfahren angewendet werden. Einerseits lassen sich return Werte einer Methode abändern und andererseits können Felder beim Schreiben oder Lesen mit einem eigenen Wert überschrieben werden.
Das Inspizieren und Verändern von Code kann während der gesamten Laufzeit des unbekannten Programms gestartet, gestoppt oder verändert werden.

JaInFra
Management Summary

December 22, 2011

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	18.12.2011	f1egli	Dokument erstellt
1.0rc02	19.12.2011	f1egli	Überarbeitung gemäss Feedback
1.0	20.12.2011	f1egli, dmengelt	Finale Version

http://www.jainfra.com

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Management Summary **3**
3.1 Ausgangslage 3
3.2 Vorgehen 3
3.3 Technologien 3
3.4 Ergebnisse 4
3.5 Ausblick 4

Literaturverzeichnis **5**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Management Summary

3.1 Ausgangslage

Ein Programm durchläuft von der Erstellung bis zur Ausführung verschiedene Schritte:

- Erstellung des Programms
  Der Programmierer schreibt sein Programm in einer menschenlesbaren Syntax. Dieser Quelltext kann von einem Menschen gelesen, interpretiert und analysiert werden.
- Kompilierung
  Die Kompilierung ist ein Zwischenschritt. Der Kompiler wandelt den Quelltext in eine ausführbare Datei um.
- Ausführung
  Die ausführbare Datei kann von einem Betriebssystem geladen und ausgeführt werden. Besitzt man allerdings nur diese Datei, ist die Analyse des Programms (beispielsweise wann wird welche Methode aufgerufen), mit erheblichem Aufwand verbunden.

Das Java Inspection Framework, kurz JaInFra, soll es einem erlauben, ausführbare Java Dateien einfach zu Analysieren und allenfalls das Verhalten des Programms abzuändern, obwohl man den Quelltext nicht besitzt. Es soll sowohl für Anfänger, wie auch für Fortgeschrittene Java-Entwickler bedienbar sein.

3.2 Vorgehen

Während des ganzen Projekts wurde RUP [rup] angewandt. 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 Prototypen begonnen, da im Vorhinein nicht bekannt war, ob das Projekt überhaupt im gewünschten Stil realisierbar sein kann. Dank der Dokumentation von Architektur und Design hatte man eine gute Diskussionsgrundlage und war immer im Bild, wie JaInFra aufgebaut wurde.

3.3 Technologien

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

- Java [jav]
  Wurde von der Aufgabenstellung verlangt.
- AspectJ [asp]
  Wurde in der Aufgabenstellung angedacht. Konnte durch den Prototypen als richtige Technologiewahl bestätigt werden.

3.4 Ergebnisse

Mit JaInFra kann man die unbekannten Programme analysieren und gegebenenfalls das Verhalten des Programms abändern. Die Benutzung ist dabei intuitiv und kann schnell erlernt werden. Unter anderem gibt es folgende Teilergebnisse:

- JaInFra Modi
  Durch die zwei Modi, Basic und Advanced, kann sowohl ein Anfänger, wie auch ein fortgeschrittener Java-Entwickler JaInFra ideal benutzen.
- Inspizierung der Klassen
  JaInFra bietet die Möglichkeit, Klassen vorab, das heisst nicht zur Laufzeit, zu inspizieren. Dadurch lassen sich Informationen zu Klassen, beispielsweise welche Methoden eine Klasse besitzt, vorgängig in Erfahrung bringen.
- Filter
  Der Kunde kann durch sogenannte Filter entscheiden welche Typen, wie beispielsweise ein Methodenaufruf, zum unbekannten Programm analysiert werden sollen. Diese Filter lassen sich beliebig kombinieren und können während der Programmlaufzeit ausgetauscht werden.
- Callbacks [cal]
  JaInFra schickt die Daten, welcher der Kunde will, zu ihm zurück. Das heisst, der Kunde kann entscheiden, was er mit den Daten machen will. Zum Beispiel können die Daten in ein Textfile gespeichert werden.
- Logging
  JaInFra kann sich auch selber analysieren. Dadurch können Logfiles generiert werden, welche bei allfälligen Problemen helfen, JaInFra zu verbessern.
- Injection
  JaInFra bietet einen limitierten Satz von Injectionmöglichkeiten. Es können beispielsweise Ergebnisse von Methoden mit eigenen Werten überschrieben werden. Somit würde sich ein defektes Programm mit Leichtigkeit reparieren lassen.

3.5 Ausblick

Durch die JaInFra API könnte ein GUI erstellt werden, welche es einem Endanwender erlauben würde, ein unbekanntes Programm noch einfacher analysieren zu können. Die Filterauswahl von JaInFra ist gross, jedoch wäre es denkbar, noch weitere Filter einzubauen.

Literaturverzeichnis

[asp] Offizelle Aspectj Projekt Website. URL. http://www.eclipse.org/aspectj; zuletzt besucht im Dezember 2011.

[cal] Funktionsweise Callback. URL. http://en.wikipedia.org/wiki/Callback_(computer_programming); zuletzt besucht im Dezember 2011.

[jav] Java Wiki Seite. URL. http://de.wikipedia.org/wiki/Java_(Programmiersprache); zuletzt besucht im September 2011.

[rup] Rational Unified Process. URL. http://de.wikipedia.org/wiki/Rational_Unified_Process; zuletzt besucht im Dezember 2011.

JaInFra
Technischer Bericht

December 22, 2011

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	11.12.2011	f1egli	Dokument erstellt
1.0rc02	12.12.2011	f1egli, dmengelt	Gliederung angepasst
1.0rc03	13.12.2011	f1egli	Filterergebnis hinzugefügt
1.0rc04	13.12.2011	dmengelt	Einleitung und Übersicht
1.0	15.12.2011	f1egli, dmengelt	Version 1.0
2.0rc01	16.12.2011	f1egli	Ergebnisse weitergeführt
2.0rc02	16.12.2011	dmengelt	CatchedObject Ergebnis
2.0rc03	16.12.2011	dmengelt	Schlussfolgerung LTW
2.0rc04	16.12.2011	f1egli	Schlussfolgerung CatchedObject
2.0rc05	18.12.2011	dmengelt	Überarbeitung Einleitung und Übersicht
2.0rc06	19.12.2011	f1egli, dmengelt	Überarbeitung gemäss Feedback
2.0	20.12.2011	f1egli, dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Einleitung und Übersicht **3**
3.1 Problemstellung 3
3.2 Aufgabenstellung 3

4 Ergebnisse **5**
4.1 Load Time Weaving: Wann wird "geweaved"? 5
4.2 CatchedObject: Was soll der Callback dem Kunden retournieren? 5
4.3 Die Filterklassen: Eingrenzung der Kriterien durch Kunde 5

5 Schlussfolgerungen **8**
5.1 Bewertung der Ergebnisse 9
5.1.1 Load Time Weaving (LTW) 9
5.1.2 CatchedObject 9
5.1.3 Filter 9
5.1.4 Injection 10
5.1.5 Logger 10
5.2 Weiteres Vorgehen 10

Literaturverzeichnis **11**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Einleitung und Übersicht

3.1 Problemstellung

Möchte man ein Programm erstellen, so schreibt man zuerst einen Quelltext und kompiliert [kom] diesen anschliessend. Wenn alles richtig gemacht wurde, bekommt man eine ausführbare Datei. Der Quelltext ist menschenlesbar, wohingegen die ausführbare Datei für die Maschine gedacht ist und somit nicht für einen Menschen lesbar ist. Besitzt man nur die ausführbare Datei, kann man sie also ausführen, jedoch ohne speziellen Aufwand weder analysieren, noch verändern.

3.2 Aufgabenstellung

Bei der aspektorientierten Programmierung ist es möglich, eigenen Java-Code in bestehenden Java-Bytecode einzubinden. Dabei muss der Quelltext dieses Bytecodes nicht vorhanden sein. Mittels AspectJ, einer aspektorientierten Erweiterung von Java, soll nun eine Programmierschnittstelle entwickelt werden, welche die Analyse mittels Tracing [tra] einer bestehenden Java Applikation auf Bytecode-Ebene zulässt. Folgende Grafik zeigt die spätere Verwendung der erstellten Programmbibliothek des Java Inspection Framework, kurz JaInFra:

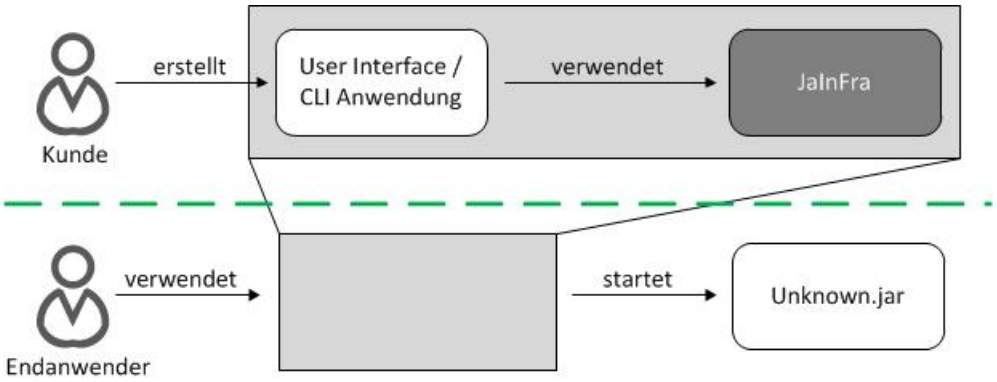

Figure 1: JaInfra Kontext

Im Kontext von JaInFra existieren zwei Benutzergruppen:

- **Kunde**

  Erstellt unter Verwendung der JaInFra Programmierschnittstelle ein eigenes Programm. Dieses Programm wird dann dem Endanwender zur Verfügung gestellt. Ob dieses Programm ein Command Line Interface [cli] oder eine graphische Benutzer Oberfläche ist, spielt keine Rolle.

- **Endanwender**

  Verwendet das vom Kunden erstellte Programm um eine ausführbare, unbekannte Applikation (Unknown.jar) zu analysieren.

Die JaInFra Programmbibliothek soll zwei unterschiedliche Modi anbieten. Einer, welcher das Analysieren der elementaren Operationen eines Programms ermöglicht und ein erweiterter, bei welchem die Filterungsmöglichkeiten sehr detailliert eingestellt werden können:

- **Einfacher Modus**

  Im einfachen Modus soll die Analyse von Methodenaufrufen bzw. Klassenfeldern möglichst einfach gehalten werden. Das primäre Ziel ist es, dass der Kunde schnell mit der Analyse beginnen kann. Nur wenige Konfigurationsschritte sollten notwendig sein um die Programmbibliothek in Betrieb zu nehmen.

- **Erweiterter Modus**

  Der erweiterte Modus richtet sich an den erfahrenen Programmierer. AspectJ Begriffe wie Advice, Before, After, Around sind ihm bereits bekannt. JaInFra soll dem Programmierer im erweiterten Modus viele Konfigurationsmöglichkeiten für die Analyse von Methoden und Felder bieten. Das Setzen, Kombinieren und Negieren von unterschiedlichen Filtertypen muss möglich sein.

- **Optionale Funktionalitäten**

  Zu den optionalen Funktionalitäten zählen das Injizieren von eigenem Java-Code in den bestehenden Bytecode und das Setzen von Breakpoints. Beides sollte jeweils zur Laufzeit möglich sein. Mit der Injekton von eigenem Java-Code sollten dann beispielsweise der Rückgabewert einer Methode, oder die Zuweisung eines Wertes an ein Klassenfeld, verändert werden können.

4 Ergebnisse

JaInFra lässt sich durch die zwei Modi sowohl von Anfängern, wie auch von Profis bedienen. Dank der flexiblen Filterung, kann der Kunde entscheiden, was überhaupt getraced werden soll. Callbacks werden dann ausgelöst, wenn ein Filter zufrieden gestellt ist. Dadurch kann der Kunde entscheiden, was mit den Daten gemacht werden soll. Mit den Injectionmöglichkeiten von JaInFra lässt sich sogar das Verhalten des Programms abändern. So hat man beispielsweise die Möglichkeit, einen Rückgabewert einer Methode mit einem eigenen Wert zu überschreiben.

4.1 Load Time Weaving: Wann wird "geweaved"?

Das Einbinden der generischen Methoden in die unbekannten Klassen nennt sich weaving. Bei AspectJ gibt es zwei Zeiten, an dem das Weaving [Lad10a] stattfinden kann:

- **Build-Time weaving**
  Es wird verwoben, wenn kompiliert wird.
- **Load-Time weaving**
  Das Verweben findet statt, wenn die Klassen geladen werden.

Laut Ausgangslage ist das unbekannte Programm bereits kompiliert und man besitzt keinen Quelltext. Zur Problemlösung des Weavingzeitpunktes wurde also das load-time weaving Konzept angewandt.
Da der Kunde zur Laufzeit der Applikation die Kriterien der Filterung ändern möchte, entstand nicht nur das Problem "wann wird geweaved", sondern auch "wo wird geweaved". Bei AspectJ kann man über sogenannte PointCuts [Lad10b] definieren, wo im Code verwebt werden soll. Die if() [ifa] Anweisung in einem PointCut wird zur Laufzeit ausgewertet. Da wir im Vornherein nicht wissen, was der Kunde tracen will, haben wir uns überall mit einer if() Anweisung eingewoben. Diese wird dann zur Laufzeit ausgewertet und je nach Filter gibt es einen Callback oder nicht.

4.2 CatchedObject: Was soll der Callback dem Kunden retournieren?

Je nach definiertem Filter wird ein Callback zum Kunden ausgelöst. Nun entstand die Frage, welche Informationen über den aktuellen JoinPoint dem Kunden zurückgeliefert werden. Die AspectJ JoinPoints [joi] enthalten sehr viele Informationen wie beispielsweise den Methodennamen, die Anzahl Argumente einer Methode, den Returntype usw. Da wir das Ziel verfolgten die kundenseitige Konfiguration möglichst einfach zu halten und man eher selten am gesamten Kontext des JoinPoints interessiert ist, haben wir das CatchtedObject eingeführt. Dieses Objekt kapselt den eigentlichen JoinPoint und stellt zudem die wichtigsten Funktionen wie getSignatureName() (für einen Felder- oder Methodennamen) bereit. Natürlich kann der erfahrene Programmierer weiterhin mittels getJoinPoint() auf den JoinPoint zugreifen.

4.3 Die Filterklassen: Eingrenzung der Kriterien durch Kunde

Da wir nicht wissen, was der Endanwender überhaupt tracen möchte, entstand das Problem, wie JaInFra nur die gewünschten Daten zurückliefert.
Der Kunde muss die Möglichkeit haben, verschiedene Kriterien zusammenzufügen und

kombinieren zu können. Wir wollten allerdings JaInFra seitig, den Code kurz halten und dabei keine unnötigen if-else Konstrukte bilden.
Die Problemlösung sind die Filterklassen, welche das Interpreterpattern[int] umsetzen.

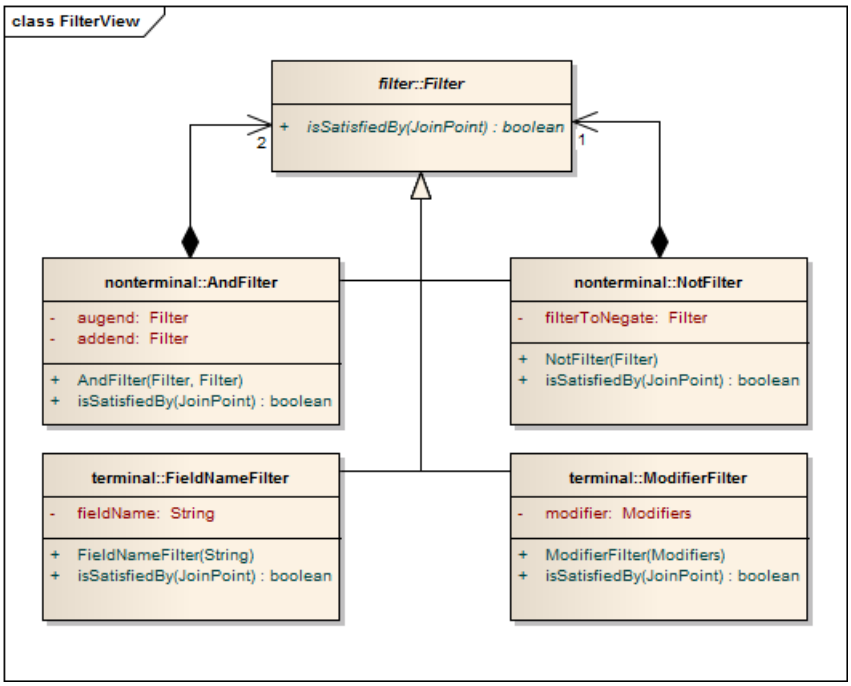

Figure 2: Auszug der Filterklassen

Es gibt folgende nonterminal Filter:

- AndFilter
  Verbindet zwei Filter(terminal oder nonTerminal) mit einem AND. Das heisst beide Argumente müssen erfüllt sein.
- OrFilter
  Verbindet zwei Filter(terminal oder nonTerminal) mit einem OR. Das heisst mindestens einer der beiden Argumente muss erfüllt sein.
- NotFilter
  Negiert den Filter.

Und folgende terminal Filter:

- AlwaysSatisfiedFilter
  Dieser Filter retourniert immer true, ist also immer zufriedengestellt. Wenn man nur diesen Filter setzt, wird entsprechend nichts ignoriert bzw. alle Daten werden dem Kunden zurückgeschickt.
- NeverSatisfiedFilter
  Dieser Filter retourniert immer false. Der Filter kann gebraucht werden um temporär nichts mehr zu schicken bzw. alles zu ignorieren.

- FieldNameFilter
  Bei diesem Filter kann via einer Regex angegeben werden, wie ein Feld heissen soll. Der Filter retourniert immer dann true, wenn etwas mit diesem Feld gemacht wird (zum Beispiel das Lesen dieses Feldes).
- MethodNameFilter
  Bei diesem Filter kann via einer Regex angegeben werden, wie eine Methode heissen soll. Er greift immer dann, wenn diese Methode aufgerufen oder ausgeführt wird.
- ClassNameFilter
  Bei diesem Filter kann via einer Regex angegeben werden, wie die Klasse heisst, bei der etwas gemacht wird (zum Beispiel das ausführen einer Methode in der angegebenen Klasse).
- PackageNameFilter
  Bei diesem Filter kann via einer Regex angegeben werden, wie der Packagename lauten soll. So kann man beispielsweise alle Felder und Methoden in einem Package tracen.
- JoinPointCategoryFilter
  Unter AspectJ gibt es verschiedene JoinPointKategorien. Ein JoinPoint ist der Teil, der gerade im Kontrollfluss des Programms ist. Die Kategorie dazu ist, was dieser Teil gerade zu tun versucht (zum Beispiel Ausführen einer Methode, lesen eines Feldes etc.). Mit diesem Filter können diese Kategorien über das entsprechende Enum (JoinPointCategory) gesetzt werden.
- ModifierFilter
  Über diesen Filter kann man die Modifier einer Methode oder eines Feldes setzen. Dies geschieht über das entsprechende Enum (Modifiers). Beispielsweise kann man so alle static Methoden und Felder tracen.
- ReturnTypeFilter
  Mit diesem Filter kann man den Return Typ einer Methode setzen. Dies geschieht via einem String. Möchte man alle Methoden tracen die void retournieren, wäre der ReturnTypeFilter die richtige Wahl.
- MethodParameterCountFilter
  Bei diesem Filter kann via einem Integer angegeben werden, wie viele Parameter in einer Methode vorhanden sein soll.
- MethodParameterTypeFilter
  Bei diesem Filter kann mit einem String angegeben werden, welcher Parametertyp mindestens einmal vorkommen soll. So kann man beispielsweise alle Methoden tracen, welche mindestens einen String als Argument entgegennehmen.

Als Beispiel: Trace aller Methoden get,set,getter,setter etc. in der Klasse Wuerfel.

Listing 1: BeispielFilter

```
1 Filter f =
2    new AndFilter(
3       new ClassNameFilter("Wuerfel"),
4       new MethodNameFilter("[gGsS]et.*"));
```

5 Schlussfolgerungen

Mit JaInFra ist es möglich, innert kürzester Einarbeitungszeit unbekannte Applikationen zu tracen und das Verhalten via Injections abzuändern. Mittels folgenden Klassen kann man beispielsweise alle set, get, setter, getter etc. Methoden inspizieren:

Listing 2: Callback Receiver

```
1  public class MyJaInFraReceiver implements IReceiver{
2
3      @Override
4      public void callbackBefore(CatchedObject catchedObject) {
5          System.out.println("Before " + catchedObject);
6      }
7
8      @Override
9      public void callbackAfter(CatchedObject catchedObject) {
10         System.out.println("After " + catchedObject);
11     }
12 }
```

Listing 3: Main Klasse

```
1  public class Main {
2      public static void main(String[] args) {
3          JaInFraAdvanced jif = JaInFraAdvanced.getInstance();
4          try {
5              jif.configureApplication(null, new URL("file:///tmp/x.jar"));
6              jif.enableAdvice(true, true);
7              jif.configureReceiver(new MyJaInFraReceiver());
8              jif.setFilter(new MethodNameFilter("[gGsS]et.*"));
9              jif.start();
10         } catch (MalformedURLException e) {
11             e.printStackTrace();
12         } catch (Exception e) {
13             e.printStackTrace();
14         }
15     }
16 }
```

Die Ausgabe könnte dann so aussehen:

Listing 4: Output

```
1  Before call(int Wuerfel.getSumme())
2  Before execution(int Wuerfel.getSumme())
3  After execution(int Wuerfel.getSumme())
4  After call(int Wuerfel.getSumme())
5  Before call(void javax.swing.JTextField.setText(String))
6  After call(void javax.swing.JTextField.setText(String))
```

5.1 Bewertung der Ergebnisse

5.1.1 Load Time Weaving (LTW)

Die Anforderungen verlangten den Einsatz von AspectJ und Java. Da jedoch der Kunde die Tracing Kriterien zur Laufzeit (über die Filter) anpassen können muss und AspectJ kein Runtime Weaving unterstützt, galt es folgende zwei Variante gegeneinander abzuwägen:

1. **Analyse mit Reflection und anschliessender Generierung der Aspects**

   Hierbei hätten wir vorgängig mittels Reflection die gesamte Struktur der zu tracenden Applikation analysiert und dann die Aspects dynamisch in eine XML Datei geschrieben.

2. **Weaving zur Ladezeit (LTW) mittels WeavingURLClassLoader [url]**

   Der AspectJ WeavingURLClassLoader verwebt die bereits vorgängig definierten Aspects mit der zu tracenden Applikation:

Listing 5: Weaving mit dem AsepctJ WeavingURLClassloader

```
1  weavingClassLoader = new WeavingURLClassLoader(sourceList, aspectsList,
2  classloader)
```

Die erste Variante verunmöglicht es, zur Laufzeit gewisse Anweisungen zu überprüfen. Möchte der Kunde also das Tracing eingrenzen oder verändern, müsste die Applikation gestoppt werden. Die angepassten Aspects müssten dann neu mit dem Programm verwoben werden. Aus diesem Grund haben wir uns für die zweite Variante entschieden. Das Weaving wird gestartet, sobald der Kunde die start() Methode aufruft. Der WeavingURLClassloader verwebt hierbei den Bytecode (also die zu tracende Applikation) mit den definierten Aspects. Dies bietet uns die Flexibilität, die Aspects und das darin enthaltene Filtering bereits vorgänging zu definieren.

5.1.2 CatchedObject

JaInFra muss auch ohne Kenntnisse von AspectJ bedienbar sein können. Anfänglich haben wir im Callback zum Kunden einfach den entsprechenden JoinPoint[joi] mitgeschickt, denn in diesem Objekt sind die Daten, die der Kunde für das Tracing braucht. Da JoinPoints aber ein Bestandteil von AspectJ ist, hätten wir die die Kunden eventuell verunsichert. Mit der Einführung von CatchedObject, welches den aktuellen JoinPoint kapselt und zusätzliche Methoden bereitstellt, konnten wir diesem Problem entgegentreten.

5.1.3 Filter

Durch die Verwendung des Interpreterpattern [int] konnte der Code kurz und dadurch besonders einfach wartbar gemacht werden. Kundenseitig können beliebige Konstrukte gebildet werden und die Filter sind ausserdem, mit anonymen inneren Klassen, erweiterbar.
Unsere vorgängigen Ideen konnten die Filterlösung nicht überbieten:

- Alle JoinPoints[joi] werden dem Kunden geschickt.
  Diese Lösung hat mehrere Nachteile. Der Kunde muss alle Daten selber Filtern, was unkomfortabel ist. Die Performance leidet, denn es muss immer alles geschickt werden.

- Statische Filter werden vordefiniert.
  Nachteilig ist hierbei, dass es unendlich viele Filterkombinationen gibt. Alle Kriterien hätte man also sowieso nie definieren können und der Kunde hätte sich durch eine riesige Filterliste durcharbeiten müssen, um den gewünschten Filter finden zu können.

5.1.4 Injection

Wir haben bewusst nur zwei Injectionmöglichkeiten in JaInFra implementiert:

- Beim Lesen oder Schreiben eines Feldes kann der Wert via Callback verändert werden.

- Der Returnwert einer Methode kann verändert werden. Die Methode selber wird jedoch abgearbeitet.

Bei Injections ist die Gefahr gross, das Exceptions auftreten können. Die in JaInFra angebotenen Injectionmöglichkeiten empfinden wir als die nützlichsten und gleichzeitig gefahrlosesten Arten für Injection. Natürlich muss dennoch auf die Konsistenz der Datentypen geachtet werden. Gibt eine Methode einen String zurück, darf nicht ein Integer zurückgegeben werden. Mit den entsprechenden Filtern, die bei der Injection ebenfalls verwendet werden, sollte dies allerdings kein Problem darstellen.

5.1.5 Logger

Wenn man sich in die aspektorientierte Programmierung einliest, wird immer als Beispiel das Logging der eigenen Applikation erwähnt. Normalerweise verursacht die Einführung eines Loggers höhere Kopplung und redundanten Code. Natürlich lässt sich diese mit AspectJ vermeiden und uns war klar, dass wenn wir schon ein Projekt mit aspektorientierter Programmierung erstellen, dass wir die dann auch für das eigene Logging gebrauchen. Wir konnten die Filterklassen für das Logging verwenden. So ist es nun möglich, das Logging gezielt anwenden zu können.

5.2 Weiteres Vorgehen

Das weitere Vorgehen bestünde jetzt darin, ein GUI zu entwickeln, dass die JaInFra API verwendet. Dies war jedoch nicht Teil der Arbeit.

Literaturverzeichnis

[cli] Erklärung CLI. Website. http://en.wikipedia.org/wiki/Command-line_interface; besucht im November 2011.

[ifa] if() Pointcut Expression. Website. http://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj-pcadvice.html#d0e3729; besucht im Oktober 2011.

[int] Interpreter Pattern. Website. http://en.wikipedia.org/wiki/Interpreter_pattern; besucht im Dezember 2011.

[joi] AspectJ JoinPoints. Website. http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html; besucht im Oktober 2011.

[kom] Genauer Beschreib des Kompiliervorgangs. Website. http://de.wikipedia.org/wiki/Kompilierung; besucht im Dezember 2011.

[Lad10a] Ramnivas Laddad. *AsepectJ in Action*. Manning, 2010. Seite 199.

[Lad10b] Ramnivas Laddad. *AsepectJ in Action*. Manning, 2010. Seite 64.

[tra] Funktionsweise Tracing. Website. http://en.wikipedia.org/wiki/Tracing_(software); besucht im Dezember 2011.

[url] AspectJ WeavingURLClassloader. Website. http://www.eclipse.org/aspectj/doc/next/weaver-api/org/aspectj/weaver/loadtime/WeavingURLClassLoader.html; besucht im Oktober 2011.

JaInFra
Persönliche Berichte

December 22, 2011

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	18.12.2011	dmengelt	Template erstellt
1.0rc02	19.12.2011	f1egli	Draft von Frederick Egli
1.0rc03	19.12.2011	f1egli	Draft von Dominik Mengelt
1.0rc04	21.12.2011	f1egli	Fertigstellung f1egli
1.0rc05	21.12.2011	dmengelt	Fertigstellung dmengelt
1.0	21.12.2011	f1egli, dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Persönliche Berichte **3**
3.1 Frederick Egli 3
3.2 Dominik Mengelt 4

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Persönliche Berichte

3.1 Frederick Egli

Dank der Studienarbeit konnte ich mich in die aspektorientierte Programmierung bzw. AspectJ einarbeiten. Den Ansatz, logische Aspekte von der Geschäftslogik zu trennen, finde ich extrem spannend. Ich werde in weiteren Projekten die aspektorientierte Programmierung sicher als mögliche Lösung für Probleme wie Logging- oder Auditverhalten in Betracht ziehen. Für die Dokumentation brauchten wir das erste mal LaTeX. Ich werde wohl nie wieder darauf verzichten wollen, denn das Mergen von Texten ist viel einfacher als bei entsprechenden WYSIWYG Editoren.

Da wir durch das gesamte Projekt RUP angewandt haben, konnte ich mein Verständnis dazu noch weiter ausbauen. Insbesondere habe ich gelernt, dass es wichtig ist, sich zu hinterfragen, ob es überhaupt Sinn macht gewisse Dinge zu dokumentieren. Beispielsweise haben wir auf Sequenzdiagramme verzichtet, da sie dem Leser des Architektur und Design Dokumentes nicht dienlich wären. Extrem schade finde ich, dass die HSR noch zusätzliche den Technischen Bericht als Abgabe verlangt. So muss man gegen das "dry" (don't repeat yourself) Prinzip verstossen.

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 äusserst konstruktiv. So konnte man, während den Sitzungen, Thematiken besprechen und gegebenenfalls ausdiskutieren. Es wurde nie einfach nur befohlen, sondern immer sachlich argumentiert, sodass wir alle zusammen auf gute Lösungsansätze kamen.

Da unsere Arbeit ein API war, kamen viele neue Probleme auf uns zu. Dies spürte man schon bei den Begriffsgrundlagen. Ist unser Kunde derjenige, der das API braucht oder der Endanwender? Durch genaue Begriffsdefinition konnten wir dieses Problem lösen. Ein weiteres Problem waren die Exceptions. Wann soll eine Exception weitergereicht und wann soll sie durch JaInFra selbst gecatched werden? Das Testen einer API bringt ebenfalls neue Erfahrungen mit sich. Natürlich kann man die interne Logik mit JUnit testen. Wie kann man jedoch den Kunden simulieren? Es blieb uns nichts anderes übrig, als eigene Kunden Applikationen zu schreiben.

Ich würde JaInFra jederzeit wieder als Studienarbeit wählen. Die 14 Wochen waren äusserst lehrreich und sehr spannend.

3.2 Dominik Mengelt

Zu Beginn der Studienarbeit hatten wir eine ziemlich lange Evaluationsphase. Es war unklar, ob die gestellten Anforderungen überhaupt erfüllt werden können. Wir mussten unterschiedliche Frameworks zur aspektorientieren Programmierung testen. Als wir uns schlussendlich für AspectJ entschieden, stellten sich schnell neue Probleme heraus. Das Weaving zur Laufzeit wird von AspectJ grundsätzlich nicht unterstützt. Glücklicherweise wertet AspectJ mit den sogenannten "if() Pointcut Expressions" eine Bedingung zur Laufzeit aus. Unsere Idee war es dann, die Aspect Klassen so zu definieren, dass sich diese in praktisch jeden Code Abschnitt der zu tracenden Applikation "einweben". Der Kunde kann anschliessend mit den JaInFra Filtern selbst bestimmen, an welchem Teil des Programmablaufs er interessiert ist. Innerhalb der "if()-Checks" prüfen wir dann die vom Kunden angegebenen Filter. Das Design des Filtermechanismus konnten wir mit dem Interpreter Pattern sehr sauber gestalten. Uns war bewusst, das dieses Vorgehen die Performance der zu tracenden Applikation einschränkt. Allerdings spielt die Performance eher eine unwichtige Rolle, wenn es darum geht eine Applikation zu tracen.

Neu war für uns auch die Entwicklung eines API. Bis anhin entwickelten wir immer das Endprodukt und es war teilweise schwierig, sich in die Lage des "API-Anwenders" zu versetzen. Ziel unserer Arbeit war es auch, das man beliebige ausführbare Java Programme tracen kann. Normalerweise verwendet man AspectJ gezielt, um ein Programm beispielsweise mit einer Logging Funktionalität zu erweitern. Das war bei JaInFra natürlich anders. Wir entwickelten, ohne Kenntnisse über das zu tracende Programm zu besitzen. Somit entwickelte sich auch das Testing zu einer heiklen Angelegenheit. Wir versuchten die JaInFra Library mit möglichst vielen "unbekannten" JAR Files zu testen.

Für die Dokumentation während des Projekts haben wir RUP eingesetzt. Dies war natürlich bei einer Arbeit, wo die Anforderungen zu Beginn nicht ganz klar waren teils schwierig. Trotzdem half uns diese Vorgehensweise vor allem am Ende des Projekts. Wir mussten nicht noch in den letzten zwei Wochen die gesamte Dokumentation schreiben. Zudem haben wir früh im Projekt eine Artefaktenliste mit den abzugebenden Dokumenten erstellt. Häufig konnten wir uns an dieser Liste orientieren. Dies ermöglichte es uns, die anstehenden Arbeiten optimal zu planen.

Organisationsprobleme im Team gab es eigentlich nie. Wir arbeiteten bereits im UserInterface 1 und im Software Engineering 2 Projekt zusammen. Vielmals setzten wir auf Pair Programming und konnten so Designprobleme gleich während des Programmierens miteinander diskutieren.

Persönlich habe ich während der Studienarbeit einiges gelernt. Ich war einige Male beeindruckt von der Mächtigkeit der aspektorientierten Programmiersprache. Natürlich nehme ich auch einiges aus dem Projektmanagement für die Bachelor Arbeit mit. Es ist wichtig, dass man früh die Anforderungen definiert, damit man anschliessend eine gute Basis für das weitere Vorgehen im Projekt hat.

JaInFra
Projektplan

December 22, 2011

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	23.09.2011	dmengelt	Erstellung des Dokuments
1.0rc02	27.09.2011	f1egli	Risikomanagement, Releases, Termine
1.0rc03	07.10.2011	dmengelt	Artefaktenliste Verweis, Releasebeschreibung, Projektorganisation
1.0rc04	11.10.2011	dmengelt	Art der Zeiterfassung ergänzt
1.0rc05	18.10.2011	dmengelt	Kleinere Anpassungen
1.0	21.10.2011	dmengelt	Anpassungen und Korrekturen
2.0rc01	31.10.2011	f1egli, dmengelt	Erwähnung Releasepapers
2.0rc02	08.11.2011	dmengelt	Domainmodel Entscheid
2.0	15.11.2011	dmengelt	Genauere Beschreibung Releases
3.0rc01	18.11.2011	dmengelt	Korrekturen QM
3.0rc02	29.11.2011	f1egli	Ergänzungen QM
3.0	01.12.2011	dmengelt	Version 3.0
4.0rc01	12.12.2011	dmengelt	Literaturverzeichnis
4.0	20.12.2011	dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Projekt Übersicht **3**
3.1 Projektidee . . . 3
3.2 Annahmen und Einschränkungen . . . 3
3.3 Entscheid Domainmodel . . . 3

4 Projektorganisation **4**
4.1 Organisationsstruktur . . . 4
4.2 Externe Partner . . . 4

5 Management Abläufe **4**
5.1 Zeiterfassung . . . 4
5.2 Wochenbesprechung . . . 5
5.3 Releases . . . 5
5.3.1 Releasepapers . . . 5
5.3.2 Release 1.0 Prototype . . . 5
5.3.3 Release 2.0 . . . 5
5.3.4 Release 3.0 . . . 5
5.3.5 Release 4.0 . . . 6
5.4 Artefaktenliste . . . 6

6 Risiko Management **6**

7 Issues **7**
7.1 Issue Kategorien . . . 7

8 Infrastruktur **7**

9 Qualitätsmanagement **8**
9.1 Besprechung & Pair Programming . . . 8
9.2 Versionsmanagement [vmg] . . . 8
9.3 Automatischer Build . . . 8
9.4 Project Style Guide . . . 8
9.4.1 Code Style Guide . . . 8
9.4.2 Code Formatter Eclipse [ecl] . . . 8
9.5 Unit Testing . . . 8

Literaturverzeichnis **9**

2 Ziel und Zweck

Siehe Kapitel "Artefaktenliste"

3 Projekt Übersicht

3.1 Projektidee

Oft kommt es vor, dass man nur die *.class Files bzw. ein JAR File besitzt, wenn man zusätzliche Libraries in seinem Java Code verwenden möchte. Ohne den Source-Code ist es denkbar schwierig, die Methoden und Felder dieser Klassen zu Testen und zu Analysieren. Das Ziel von JaInFra ist es, genau dieser Problematik entgegenzutreten. Mit Hilfe von JaInFra werden folgende Punkte möglich:

- Alle definierten Methoden und Felder einer kompilierten Klasse können gefunden werden.
- Informationen zur Klasse: ist es eine "normale" oder Abstrakte Klasse, welches Interface wird implementiert, von welcher Klasse erbt diese Klasse etc.
- Zu den gefundenen Methoden können Tracing Informationen eingeblendet werden: wann wird diese Methode aufgerufen und mit welchen Übergabeparameter
- Ebenfalls können Informationen zu den gefundenen Felder angezeigt werden: wann wird ein Feld verändert etc.

In einem Satz: Durch JaInFra wird das Debuggen und Analysieren von kompiliertem Code erheblich vereinfacht.

3.2 Annahmen und Einschränkungen

Pro Projektmitglied sind 240 Arbeitsstunden für das Projekt vorgesehen.

3.3 Entscheid Domainmodel

Wegen den gegebenen Randbedingungen (Einsatz von AspectJ [asp]) und weil JaInFra als solches keine Datenschicht (Datenmodell) benötigt, wird auf die Erstellung eines Domainmodells verzichtet. Abläufe und Anforderungen an das Produkt können dem Architektur und Design bzw. den Anforderungsspezifikationen entnommen werden:

`03_Anforderungen/Anforderungsspezifikation_vX.X.pdf`

`04_Architektur_und_Design/Architektur_und_Design_vX.X.pdf`

4 Projektorganisation

4.1 Organisationsstruktur

Name	Frederick Egli	Dominik Mengelt
Kürzel	f1egli	dmengelt
Verantwortung	• Implementation • Testing • Qualitätsmassnahmen • Developer Guide • Architektur und Design	• Implementation • Riskikomanagement • Projektplanung • Anforderungsspezifikationen • Testing

4.2 Externe Partner

Das Projekt wird von Herrn Thomas Letsch betreut und bewertet.

5 Management Abläufe

5.1 Zeiterfassung

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

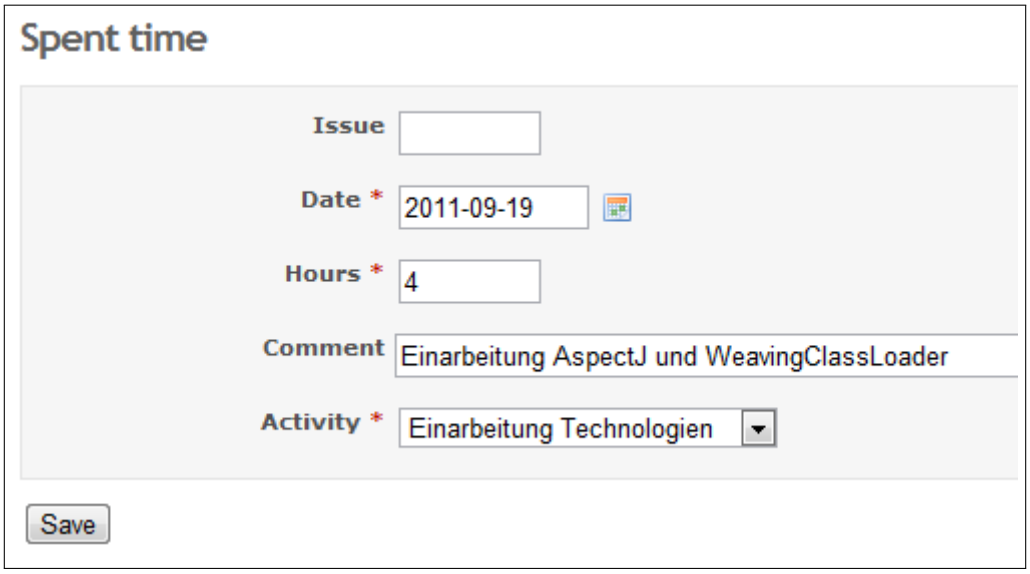

Figure 1: Zeiterfassung auf Aktivitäten in Redmine

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

5.2 Wochenbesprechung

Am Mittwoch findet jeweils 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/03_Protokolle/YYYYMMDD.txt`

5.3 Releases

Während den 14 Wochen des Semesters sind folgende 4 Releases geplant:

#	Datum	Name	Iteration/Phase
1.0	26.10.2011	R1.0 Prototype	Elaboration [Lar09]
2.0	23.11.2011	R2.0	Construction 2 [Lar09]
3.0	07.12.2011	R3.0	Construction 3 [Lar09]
4.0	23.12.2011	R4.0	Transition 1 [Lar09]

5.3.1 Releasepapers

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

- New Features
- Defects Fixed
- Known Issues

5.3.2 Release 1.0 Prototype

Erste Funktionalitäten vorhanden. Tracing bestimmter Methodenaufrufe möglich.

5.3.3 Release 2.0

Erste Version des JaInFra Advanced Modus. Mittels Filter können Klassenfelder und Methoden getraced werden. Filter können kombiniert werden, um dass Tracing einzugrenzen.

5.3.4 Release 3.0

Weitere Funktionalitäten des Advanced Modus implementiert. Optionale Features fliessen in diesen Release. Basic Version abgeschlossen. Sourcen und UML-Modell sind synchron.

5.3.5 Release 4.0

Sämtliche Hauptfunktionalitäten für den Basic- und Advancedmodus implementiert. Der bestehende Code wurde getestet und die einzelnen Codeteile wurden zusammengeführt.

5.4 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 sichtbar. Der Projektmitarbeiter kann sich an dieser Liste orientieren. Zusätzlich werden pro Artefakt die Versionen während des Projekt festgehalten. Beispiel:

Anforderungsspezifikationen
"Was muss das Produkt können". Enthält weder Termine noch Teamangaben.
v1.0 Use Case und Funktionalitäten beschrieben
v2.0 Sämtliche Anforderungen definiert

6 Risiko Management

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

7 Issues

Am Ende der Phasen Elaboration [Lar09], Construction 2 [Lar09], Construction 3 [Lar09] und Transition 1 [Lar09] folgt ein Release. Die geschlossenen Issues können der Projektmanagement Applikation Redmine entnommen werden. Eine Roadmap zeigt den aktuellen Stand eines Releases:

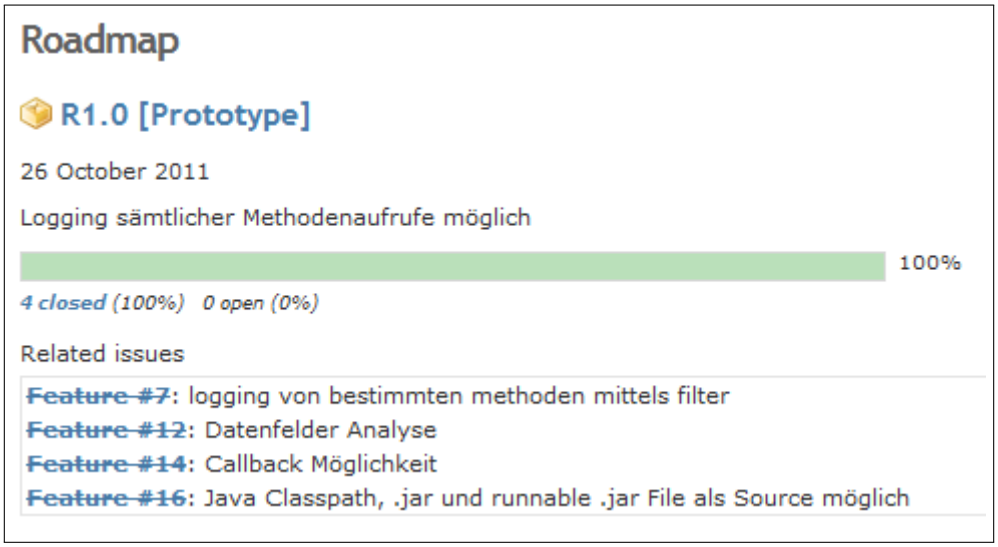

Figure 2: Roadmap mit geschlossenen Issues

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 [ecl] Indigo mit AspecJ [asp] Plugin. Beim Versionsmanagement [vmg] wird ein entsprechender Server eingesetzt.

Die Projektmanagement Applikation Redmine kann unter folgender URL erreicht werden:

`http://www.jainfra.com/`

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 Refactoring und schlechte Designentscheidungen werden somit schneller erfasst und korrigiert. Zusätzlich hilft es für den Know-How Transfer innherhalb des Teams.

9.2 Versionsmanagement [vmg]

Die Projektrelevanten Dateien und Source-Code 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. Ausserdem muss es möglich sein, Dateien und Ordner in jeder Version wieder hervorzurufen.

9.3 Automatischer Build

Nach jeder Änderung am Source-Code werden mittels Jenkins [jen] Build-Server automatisch die Unit 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 [ecl]

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 JaInFra Logiken sollte mit mindestens 90% gedeckt sein. Das heisst, dass die Mechanismen (beispielsweise Verwendungen von Filtern) getestet wird. Eine Simulation, welche eine unbekannte Applikation traced und inspiziert, wird nicht mittels JUnit [jun], sondern von den Entwicklern, dem Kunde und unabhängigen Testpersonen direkt getestet.

Literaturverzeichnis

[asp] Offizelle Aspectj Projekt Website. URL. http://www.eclipse.org/aspectj; zuletzt besucht im Dezember 2011.

[ecl] Offizelle Eclipse Projekt Website. URL. http://eclipse.org/; zuletzt besucht im Dezember 2011.

[jen] Offizelle Jenkins Projekt Website. URL. http://jenkins-ci.org/; zuletztbesucht im Oktober 2011.

[jun] Offizelle JUnit Projekt Website. URL. http://www.junit.org/; zuletzt besucht im Oktober 2011.

[Lar09] Craig Larman. *Applying UML and Patterns*. Prentice Hall PTR, 2009.

[red] Offizelle Redmine Projekt Website. URL. http://www.redmine.org/; zuletzt besucht im November 2011.

[vmg] Offizelle JUnit Projekt Website. URL. http://de.wikipedia.org/wiki/Versionsverwaltung; zuletzt besucht im Dezember 2011.

Roadmap & Artefakte

Technischer Bericht	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
	Developer Guide				V1.0	V2.0	V3.0
	Releasepapers		V1.0		V2.0	V3.0	V4.0
	QS & Testauswertung					V1.0	V2.0
	Implementation		V1.0		V2.0	V3.0	V4.0
	Architektur & Design		V1.0		V2.0	V3.0	V4.0
	UML Modell					V1.0 V2.0	
	Anforderungsspezifikationen		V1.0		V2.0		
	Projektplan		V1.0		V2.0	V3.0	V4.0
	Zeitanalyse	**					
	Protokoll Besprechung	*					
	Semesterwoche	1 2 3	4 5 6	7 8	9 10	11 12	13 14
	Phase	Inception	Elaboration	Construction 1	Construction 2	Construction 3	Transition 1

[Artefaktenliste_v4.0.pdf]

Versionen und Beschreibung

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

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

Projektplan
Beschreibt wie das Projekt abgewickelt wird inkl. Termine und Organisation
v1.0 Projektidee, Projektorganisation und Termine. Grober Beschrieb der Releases
v2.0 Genauerer Beschrieb der weiteren Releases
ProjecStyleGuide.pdf neu im Projektplan
v3.0 UnitTesting ausgearbeitet
v4.0 Letzte Releasebeschreibung

Anforderungsspezifikationen
"Was muss das Produkt können". Enthält weder Termine noch Teamangaben.
v1.0 Use Case und Funktionalitäten beschrieben
v2.0 Sämtliche Anforderungen definiert

UML Modell
Anhand des Modell wird das Zusammenspiel der einzelnen Klassen und Interfaces aufgezeigt.
v1.0 UML Modell grösstenteils fertiggestellt
v2.0 UML-Modell und die Implementation v3.0 sind synchron

Architektur & Design
Das A&D zeigt Architektur und Design Entscheide innerhalb der Implementation.
v1.0 Planung und Entscheide mit Begründung, Grobe Package Struktur,
Verwendeter Kompilierbefehl und Weaving Prozess
v2.0 Verfeingerung der Planung und erste Designentscheide
v3.0 Designentscheide mit Begründungen
Enhält neu das Designmodell
v4.0 Sämtliche Diagramme und Grafiken aktualisiert

Implementation
Source Code, Unit Tests und Kundendemo von JaInFra
v1.0 Logging von Methodenaufrufen funktioniert
v2.0 Implementierung Advanced Modus
v3.0 Sourcen und UML-Modell sind synchron.
v4.0 Refactoriung und Fertigstellung aller Tests

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

Releasepapers
Beschreibt die Features zum jeweils aktuellen Release

Developer Guide
Erklärt die Verwendung von JaInFra anhand von Beispielen für den Kunden
v1.0 Basic Modus
v2.0 Filtering und Tracing für den Advanced Modus
v3.0 Injection und Logging

Tools und Infrastruktur
Zeigt die vom Team eingesetzten Tools und Werkzeuge
v1.0 Vorgehen automatische Builds und GIT
v2.0 Verwendung checkStyle, FindBugs

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.

Bericht - Technischer Bericht der Arbeit
Enthält Einleitung und Übersicht, Ergebnisse und Schlussfolgerungen
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 Bis Dato aktuelles Glossar
v2.0 Fertiggestelltes Glossar

Bericht - Poster
v1.0 Zusammenfassung der Arbeit auf einem Poster.

Risikomanagement JaInFra

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

Legende
Infrastrukturelle Risiken
Risiken der Implementation

Priorität	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 (AspectJ, Reflection, Dynamic Weaving) 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) 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**	

JaInFra
Anforderungsspezifikationen

December 20, 2011

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	07.10.2011	dmengelt	Erstellung des Dokuments mit ersten Anforderungen
1.0rc02	17.10.2011	dmengelt	Anpassung Highlevel Context Grafik, Use Case
1.0rc03	21.10.2011	f1egli	Änderungen der optionalen Funktionalitäten
1.0	24.10.2011	dmengelt	Anpassungen der Anforderungen für Release 1.0
2.0rc01	01.11.2011	dmengelt	Advanced Modus Anforderungen
2.0rc02	18.11.2011	dmengelt	Sämtliche Anforderungen definiert
2.0rc03	21.11.2011	dmengelt	Semesterarbeit neu Studienarbeit
2.0	21.11.2011	dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Produktperspektive **3**
3.1 Ausgangslage . . . 3
3.2 Verwendung von JaInFra . . . 3

4 Anforderungen **5**
4.1 Basic Modus . . . 5
4.1.1 Funktionalitäten . . . 5
4.2 Advanced Modus . . . 5
4.2.1 Funktionalitäten . . . 5
4.3 Pfad- und Resourcenangaben . . . 6
4.4 Optionale Funktionalitäten . . . 7

5 Konkreter Anwendungsfall (Use Case) **8**

6 Einschränkungen & Annahmen **9**

Literaturverzeichnis **10**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Produktperspektive

JaInFra ermöglicht das Debuggen und Analysieren von bereits kompiliertem Code (Java *.class Dateien) innerhalb eines ausführbaren JAR Files oder eines angegebenen Verzeichnisses. Es ist ein API (Library) welches direkt in eine Entwicklungsumgebung eingebunden werden kann.

Das API kann dabei helfen, Tools und Programme für die Analyse von kompiliertem Code zu erstellen. Zudem ist es möglich zur Laufzeit einen Programmablauf zu unterbrechen um dann den aktuellen Stand der Objekte zu ermitteln.

3.1 Ausgangslage

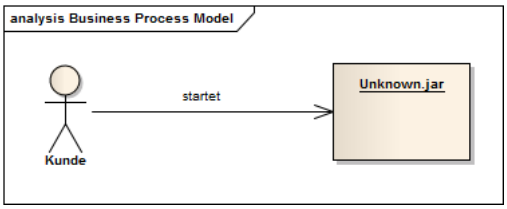

Figure 1: Ohne die Verwendung von JaInFra

Der Kunde startet eine Applikation. Es ist kein Logging und Debugging möglich. Der Sourcecode fehlt und somit bleiben keine Möglichkeiten um das Programm zu analysieren.

3.2 Verwendung von JaInFra

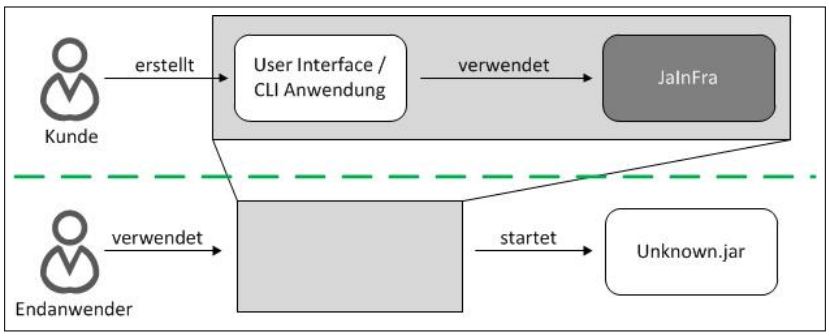

Figure 2: Verwendung von JaInFra

Der "direkte" Kunde erstellt ein User Interface oder eine Command Line Anwendung [cli] und verwendet dazu die JaInFra Library. In einem zweiten Schritt kann dann der Endanwender die erstellte Applikation einsetzen, um ein unbekanntes JAR File zu Analysieren.

4 Anforderungen

Das JaInFra API soll sowohl für erfahrene Endanwender, als auch für einfache CLI Applikationen genutzt werden können.

4.1 Basic Modus

Das Ziel des Basic Modus ist es, dem Endanwender möglichst schnell das Tracing [tra] von Methodenaufrufen und Datenfelderzuweisungen zu ermöglichen. Dieser Modus eignet sich besonders gut für die Erstellung einer einfachen CLI Applikation.

4.1.1 Funktionalitäten

Folgende Funktionalitäten müssen im Basic Modus unterstützt werden:

- Laden & starten von Java Programmen

  Die Library kann die angegebene Pfad- bzw. Resourcenangabe laden und zur Ausführung bringen.

- Tracing von Methodenaufrufen

  Sämtliche Methodenaufrufe können zur Laufzeit getraced werden. Der Benutzer kann selbst bestimmen in welchem Ausmass. Zusätzlich bietet das API Möglichkeiten zur Filterung an. Beispielsweise können nur bestimmte (Expression) Aufrufe getraced werden.

- Tracing beim Aufruf von Klassen- und Instanzvariablen

  Nebst Methodenaufrufen können auch Zugriffe auf Klassen- und Instanzvariablen getraced werden.

- Exceptions werden an den Aufrufer delegiert

  Tritt während der Analyse einer externen Applikation eine Exception auf, wird diese an den Verwender des API weitergeleitet.

4.2 Advanced Modus

Der Advanced Modus richtet sich an den erfahrenen Programmierer. AspectJ Begriffe wie Advice, Before, After, Around sind ihm bereits bekannt. JaInFra soll dem Programmierer im Advanced Modus viele Konfigurationsmöglichkeiten für das Tracen von Methoden und Felder bieten. Das Setzen und Kombinieren von unterschiedlichen Filtertypen muss möglich sein. Auch die Negierung einer Filterfunktion muss der Advanced Modus ermöglichen.

4.2.1 Funktionalitäten

Folgende Funktionalitäten müssen im Advanced Modus unterstützt werden:

- Tracing von unterschiedlichen Rückgabetypen

  Der gewünschte Rückgabetyp einer Methode kann angegeben werden. Es werden dann nur die Methoden mit dem angegebenen Typ getraced.

- Methoden Parameter (Anzahl und Typ)

  Als Filter für das Methoden Tracing können auch die Anzahl Paramter einer Methoden angegeben werden oder der jeweilige Type eines Parameters. Nur Methoden mit der gewünschten Anzahl an Parametern bzw. dem gewünschten Typ eines Parameters werden getraced.

- Tracing beim Aufruf und setzen von Klassen- und Instanzvariablen

- Tracing von Methodenaufrufen

- JoinPoint [joi] Kategorie Tracing

  Die JoinPoint Kategorie kann selbstständig gewählt werden. Beispiele sind:
  call(), execution(), get(), set()

- Advice Type Tracing

  Der Benutzer kann selbst bestimmen, wann (Before, After) ein Klassenfeld bzw. eine Methode getraced wird.

- Ablauf nicht im main() Thread

  Der JaInFra Advanced Modus läuft in einem separaten Thread. Interaktionen mit der zu tracenden Applikation sind weiterhin problemlos möglich.

4.3 Pfad- und Resourcenangaben

JaInFra muss mit folgenden Pfad- bzw. Resourcenangaben umgehen können:

- Ausführbare JAR Files
  Beinhaltet ebenfalls eine "Manifestdatei" mit der Angabe wo sich die .class Datei mit der main() Methode befindet. Zum Beispiel:

  ```
  Main-class: myApp.Launcher
  ```

- "Einfaches" JAR File
  Da es sich hier nicht zwingend um eine ausführbare .jar Datei handelt, muss die .class Datei mit der main() Methode manuell angegeben werden.

- Pfadangabe zu Java .class Dateien
  Zusätzlich muss manuell der classpath zur .class Datei mit der main() Methode angegeben werden. Zum Beispiel:

  ```
  Pfad: /home/user/app/

  Classpath:  myApp.Launcher
  ```

4.4 Optionale Funktionalitäten

- Code Injection [inj] zur Laufzeit

  Zur Laufzeit kann beispielsweise ein Methodenaufruf verändert werden. Übergebene Argumente einer Methoden können modifiziert werden. Zuweisungen von Instanz- und Klassenvariablen können geändert werden.

- Setzen von Breakpoints zur Laufzeit

  Um das Debuggen einer bestehenden ausführbaren Applikation zu vereinfachen ermöglicht JaInFra das setzen von Breakpoints zur Laufzeit. Das zu untersuchende Programm kann "pausiert" werden. Dies ermöglicht die Analyse beispielsweise von Instanzvariablen zu einem beliebig gewählten Zeitpunkt.

5 Konkreter Anwendungsfall (Use Case)

Umfang	JaInFra Applikation
Ebene	Anwenderziel
Primäraktor	API Verwender (Programmierer)
Stakeholder und Interessen	• Der Kunde kann mit seinen bestehenden Java Kenntnissen einfach ein Programm Analysieren und Debuggen. • Unter Verwendung der JaInFra Bibliothek kann der Kunde selbst bestimmen ob er eine Command Line oder eine GUI Applikation erstellt, welche anschliessend die Analyse ermöglicht. • Die initiale Konfiguration ist schnell gemacht.
Vorbedingungen	JaInFra wurde als Bibliothek in die Programmierumgebung eingebunden.
Standardablauf	1. Der Kunde gibt den Pfad zur Applikation an, welche Analysiert werden soll. 2. Er bestimmt wann eine Methode oder ein Datenfeld getraced werden soll und implementiert dafür das Receiver Interface. 3. Zusätzlich legt er einen Filter fest. Nur Aufrufe welche mit dem festgelegten Filter übereinstimmen werden analysiert. 4. Mittels Callback werden dann sämtliche Aufrufe an den implementierten Receiver geliefert.
Erweiterungen	-
Spezielle Anforderungen	-
Liste der Technik- und Datenvariationen	-
Häufigkeit des Auftretens	Fortlaufend
Verschiedenes	-

6 Einschränkungen & Annahmen

Handelt es sich beim zu analysierende Programm um ein JAR File, muss diese die Spezifikationen von Oracle einhalten:

`http://download.oracle.com/javase/6/docs/technotes/guides/jar/jar.html`

Zusätzlich gelten folgende Annahmen:

- Kunde verwendet Java 1.6 oder höher
- Kunde benötigt keine AspectJ Kenntnisse
- Das AspectJ Plugin muss für den Gebrauch von JaInFra nicht installiert sein.
- Es sind keine weiteren Bibliotheken für den Gebrauch von JaInFra einzubinden.

Literaturverzeichnis

[cli] Erklärung CLI. Website. http://en.wikipedia.org/wiki/Command-line_interface; besucht im November 2011.

[inj] Erklärung Code injection. Website. http://en.wikipedia.org/wiki/Code_injection; besucht im November 2011.

[joi] AspectJ JoinPoints. Website. http://www.eclipse.org/aspectj/doc/released/progguide/language-joinPoints.html; besucht im Oktober 2011.

[tra] Funktionsweise Tracing. Website. http://en.wikipedia.org/wiki/Tracing_(software); besucht im Dezember 2011.

JaInFra
Architektur & Design

December 22, 2011

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	17.10.2011	f1egli	Package Struktur, Weaving Prozess
1.0	21.10.2011	f1egli, dmengelt	Kompilierbefehl angepasst, Grobübersicht, Packagestruktur angepasst
2.0rc01	01.11.2011	dmengelt	Runtime Weaving Entscheid
2.0rc02	08.11.2011	dmengelt	Designmodel
2.0rc03	22.11.2011	dmengelt	Designentscheide & Umstrukturierung
2.0	25.11.2011	dmengelt	Version 2.0
3.0rc01	05.12.2011	dmengelt	CatchedObject, AspectHelper
3.0rc02	06.12.2011	f1egli	Filter und verworfene Ideen erklärt
3.0	12.12.2011	f1egli	Version 3.0
4.0rc01	12.12.2011	dmengelt	Update Designmodell
4.0rc02	16.12.2011	dmengelt	Einschränkungen
4.0rc03	20.12.2011	dmengelt	Anpassungen Package Grafiken
4.0	20.12.2011	f1egli, dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Architektur **3**
3.1 Logische Sicht . . . 3
3.1.1 Übersicht Packages . . . 3
3.1.2 Package application . . . 4
3.1.3 Package aspects . . . 4
3.1.4 Package tests . . . 4
3.2 Deployment Sicht . . . 5
3.2.1 Vorgehensweise . . . 5
3.3 Prozess Sicht . . . 6
3.3.1 Starten der Applikation (Unknown.jar) . . . 6
3.4 Technologiewahl . . . 7

4 Design **7**
4.1 Designmodell . . . 7
4.2 JaInFra Basic und Advanced . . . 8
4.3 Filtering - Das Interpreter Pattern [int] . . . 9
4.4 Aspects . . . 10
4.4.1 BasePointCutDefinition . . . 10
4.4.2 AspectHelper . . . 11
4.4.3 Anwendung der Hilfsklassen im Advanced Modus . . . 11
4.5 Callbacks . . . 12
4.5.1 Das CatchedObject . . . 12
4.5.2 Wann erfolgen die Callbacks? . . . 13
4.6 Inspecting mit Reflection . . . 14
4.7 Weitere Entscheide . . . 15
4.7.1 weaveAspects() - Load Time Weaving . . . 15
4.7.2 AspectJ und Runtime Weaving . . . 16
4.7.3 AspectJ Annotations . . . 16
4.7.4 Load Time Weaving - Suppress Warnings . . . 16
4.7.5 call() vs. exection() im Basic Modus . . . 16
4.8 Verworfene Ideen . . . 17
4.8.1 Callbacks über Sockets . . . 17
4.8.2 Markerinterface für Receiver . . . 17
4.9 Einschränkungen . . . 18
4.9.1 Maximale Länge einer Methode in Java . . . 18
4.9.2 Gleicher Fully Qualified Name . . . 18

Literaturverzeichnis **19**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Architektur

3.1 Logische Sicht

Beschreibt den Architektonischen Grobaufbau von JaInFra. Die wichtigsten Packages werden hier beschrieben.

3.1.1 Übersicht Packages

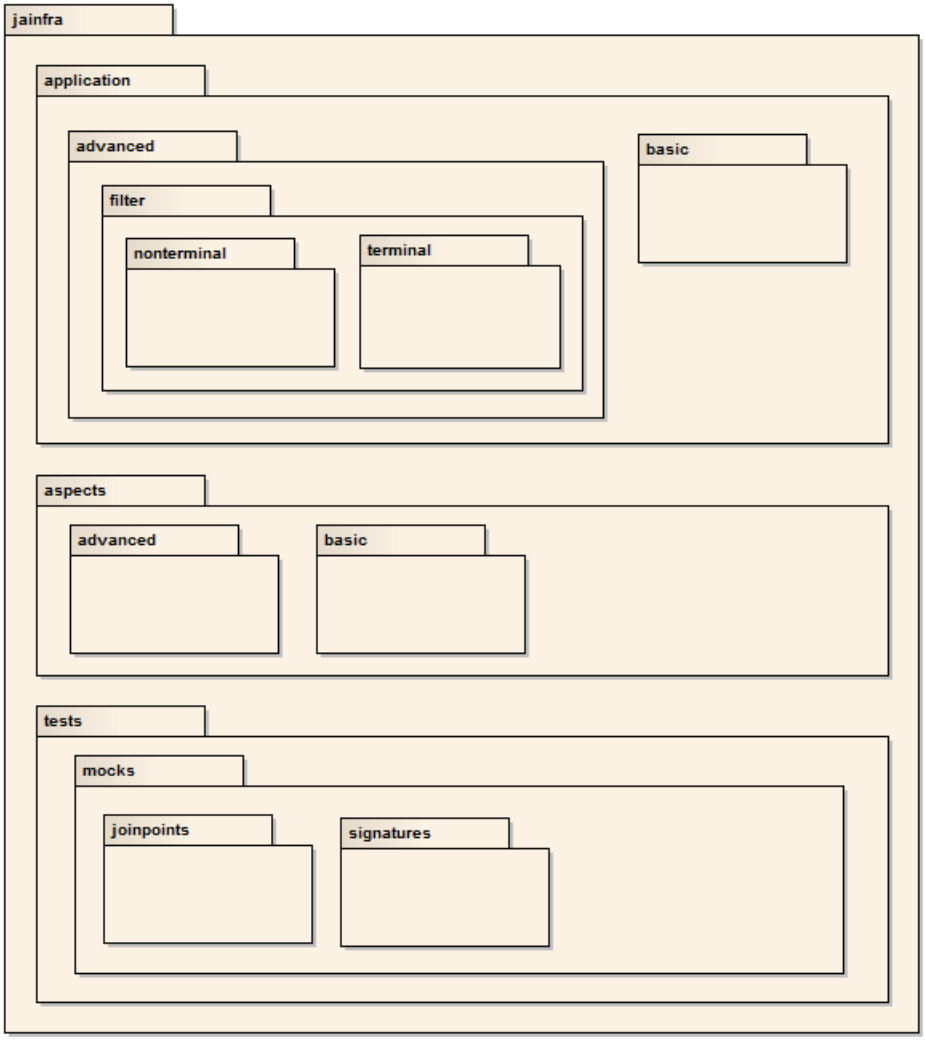

Figure 1: Package Struktur von JaInFra

Java und AspectJ Sourcefiles sind auf Packageebene getrennt. Ausserdem gibt es einzelne Packages für den Basic, sowie für den Advanced Modus.

3.1.2 Package application

Beinhaltet sämtlichen Java Code von JaInFra. Die beiden Modi, Basic und Advanced, sind separat in einzelne Packages aufgeteilt. Das ausschliesslich im Advanced Modus enthaltene Package filter, ist nochmals unterteilt in die Packages nonterminal und terminal:

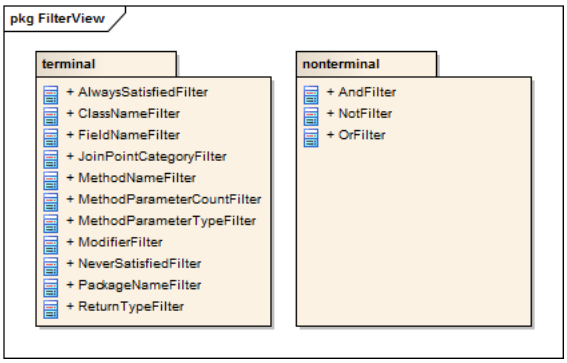

Figure 2: JaInFra Filtertypen

3.1.3 Package aspects

Beinhaltet die Aspect Klassen von JaInFra. Wie im application Package sind die Aspects für die beiden Modi in unterschiedliche Packges unterteilt:

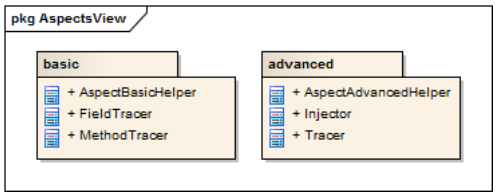

Figure 3: JaInFra Aspects

3.1.4 Package tests

Die Testklassen mit dem Package "mocks". Mocks sind nötig um die einzelnen Filtertypen zu testen. Es bestehen JoinPointMocks (JoinPoint) und SignatureMocks (Methoden oder Feldsignatur). Die Hilfsklasse JoinPointFinder erzeugt einen Array mit JoinPoints gemäss den angegeben Filtertypen.

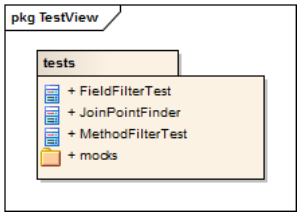

Figure 4: JaInFra Tests

3.2 Deployment Sicht

3.2.1 Vorgehensweise

Die folgende Grafik zeigt das Vorgehen bei der Erstellung der JaInFra Library bis hin zur Verwendung und dem Weaving mit der zu tracenden Applikation.

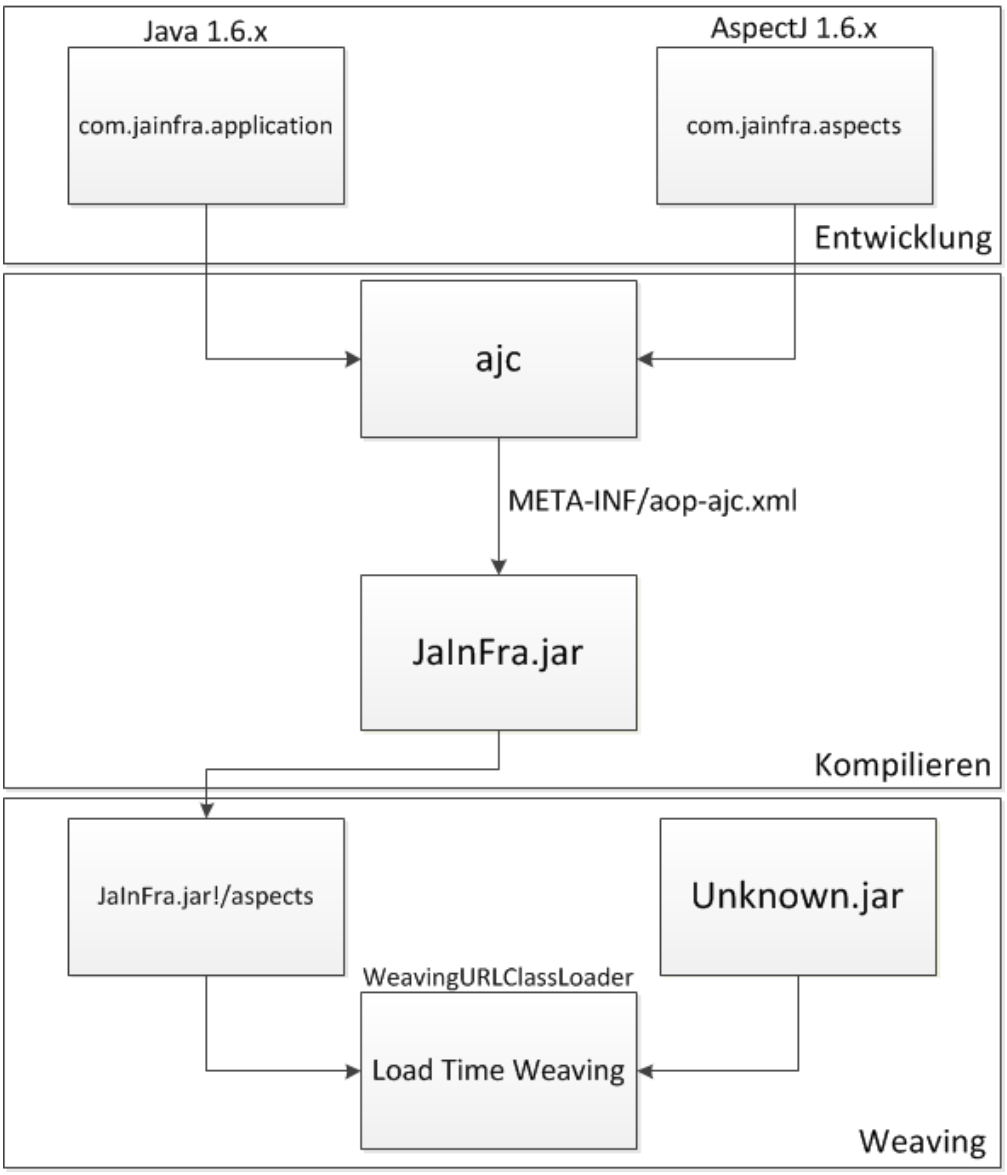

Figure 5: Ablauf JaInFra Build und Verwendung

Zu Beginn werden die Java und AspectJ Sourcefiles mittels ajc kompiliert. Das Resultat ist die JaInFra Library. Zusätzlich wurde innerhalb des neu erzeugten JAR Files auch eine Datei aop-ajc.xml [aop] erstellt, welche die vorhandenen Aspects zeigt. Diese Aspects (JaInFra.jar!/aspects) werden dann mit der unbekannten Applikation zur Ladezeit verwoben.

3.3 Prozess Sicht

3.3.1 Starten der Applikation (Unknown.jar)

Nachdem innerhalb der Kunden Applikation die start() Methode der Klasse JaInFra aufgerufen wurde, startet diese mittels Reflection die main() Methode der zu tracenden Applikation in einem separaten Thread:

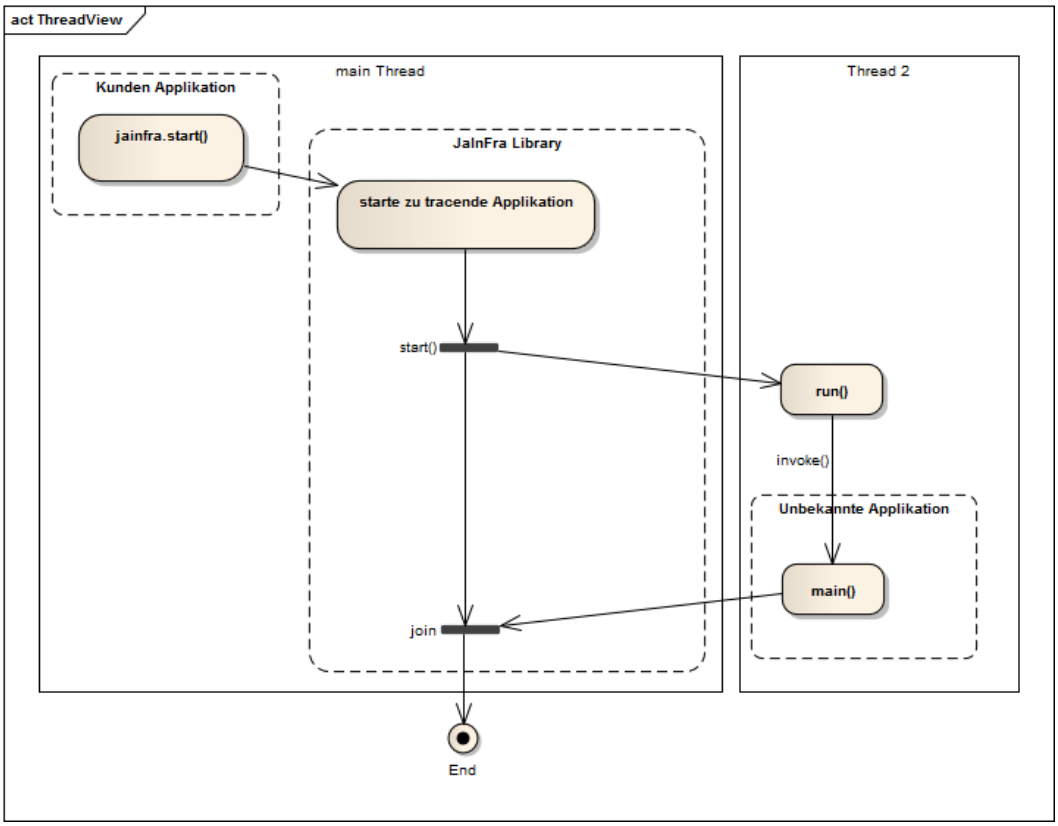

Figure 6: Starten der zu tracenden Applikation

Listing 1: Starten der Applikation (Unknown.jar)

```
1  new Thread(new Runnable() {
2      public void run() {
3          try {
4              unknownApp.getMethod("main", String[].class).invoke(null, (Object)
                   args);
5          } catch (Exception e) {
6              // Einzelne Exceptions weggelassen.
7              e.printStackTrace();
8          }
9      }
10 }).start();
```

Somit kann verhindert werden, dass die zu tracende Applikation die Ausführung der Library blockiert.

3.4 Technologiewahl

Folgende Technologien sind gemäss Anforderungsanalyse einzusetzen:

- Java 1.6.x
- AspectJ 1.6.x
- Eclipse Indigo
- Enterprise Architect 8.0.x

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

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

4 Design

4.1 Designmodell

Das Designmodell von JaInFra. Es zeigt die wichtigsten Klassen und deren Zugehörigkeiten.

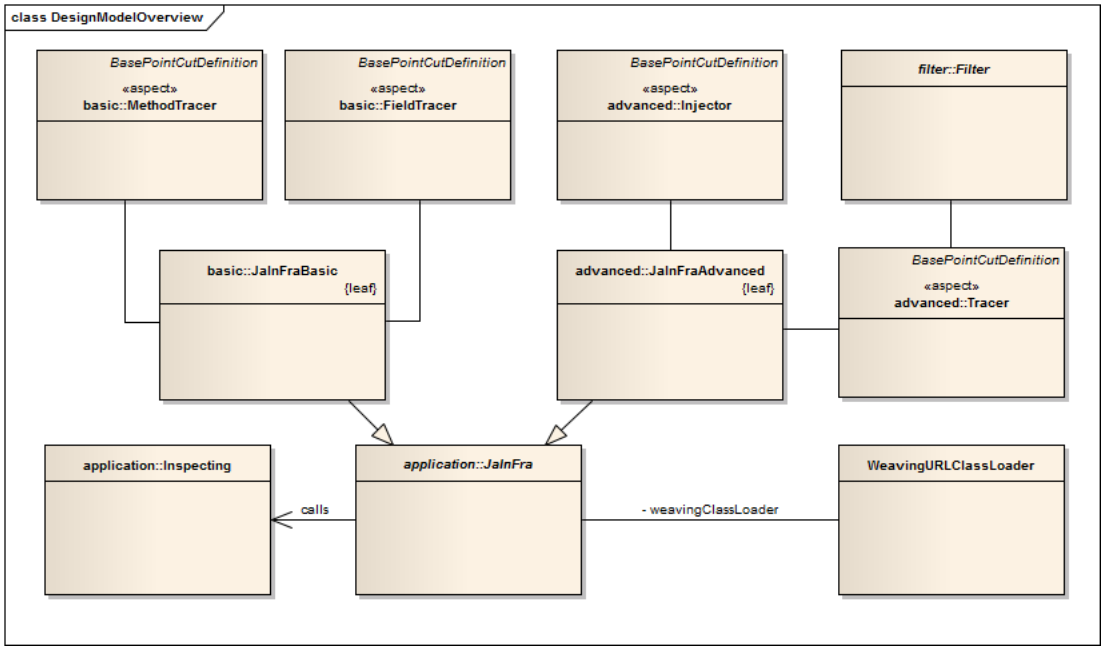

Figure 7: Highlevel Ansicht JaInFra

Es zeigt die unterschiedlichen Konzepte des Tracings. Während es im Basic Modus lediglich den Method- und Fieldtracer gibt, bieten die Filter im Advanced Modus eine viel grössere Möglichkeit um das Tracing zu konfigurieren. Das Weaving der unbekannten Applikation mit den definierten Aspects übernimmt in beiden Fällen der weavingURL-ClasssLoader [url].

4.2 JaInFra Basic und Advanced

Die beiden Varianten Basic und Advanced erhalten ihre Grundfunktionalität von der abstrakten Oberklasse JaInFra. Diese abstrakte Klasse macht dann die statischen Aufrufe auf die Inspecting Klasse und kreiert auch den WeavingURLClassLoader. Die Klassen JaInFraBasic und JaInFraAdvanced sind beides Singleton Implementierungen. Somit wird sichergestellt, dass JaInFra nur genau einmal Instanziiert werden kann.

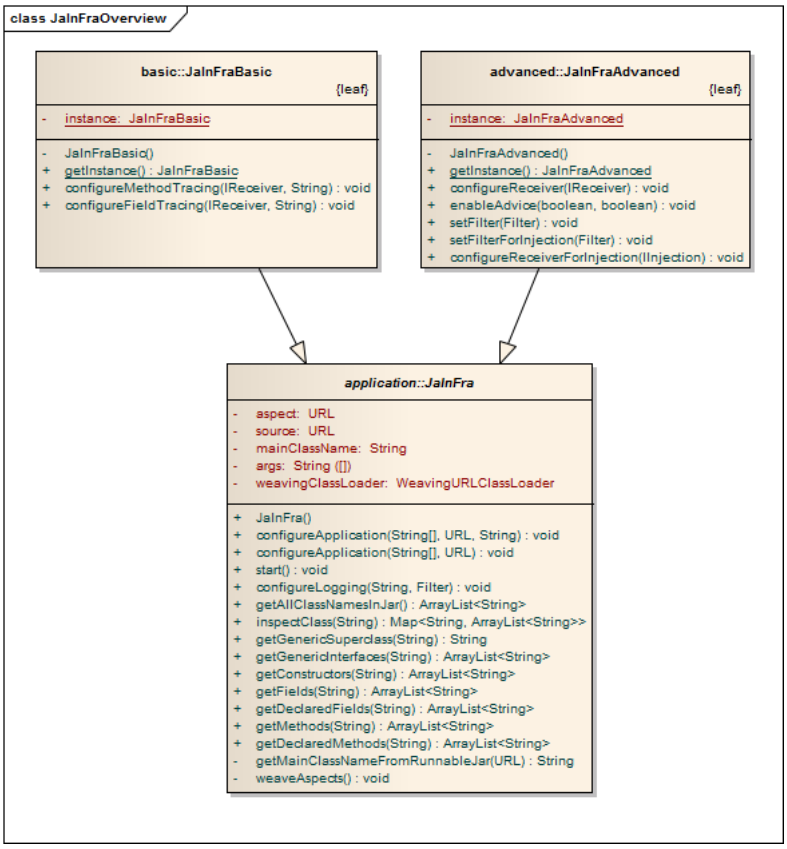

Figure 8: Aufbau Basic und Advanced Modus

Die Klasse JaInFra beinhaltet die von beiden Modi verwendeten Methoden. Sowohl die Konfiguration der zu tracenden Applikation über configureApplication() als auch das Starten über die Methode start() bleiben gleich.

4.3 Filtering - Das Interpreter Pattern [int]

Im Advanced Modus kann prinzipiell alles getraced werden. Würde man immer alle Informationen zum Kunden schicken, müsste dieser die wichtigen Abschnitte selber herausfiltern. Dies wäre aus Performancegründen ungünstig, da ein Callback in jedem Fall ausgeführt würde, ob der Kunde das nun will oder nicht. Vordefinierte statische Filter, sind schlichtweg unpraktikabel, weil die Permutationen verschiedener Filtereigenschaften praktisch unendlich viele Filter hervorrufen würde. Um die volle Konfigurationsfreiheit des Kunden erfüllen zu können, haben wir uns deshalb für das Interpreter Pattern entschieden:

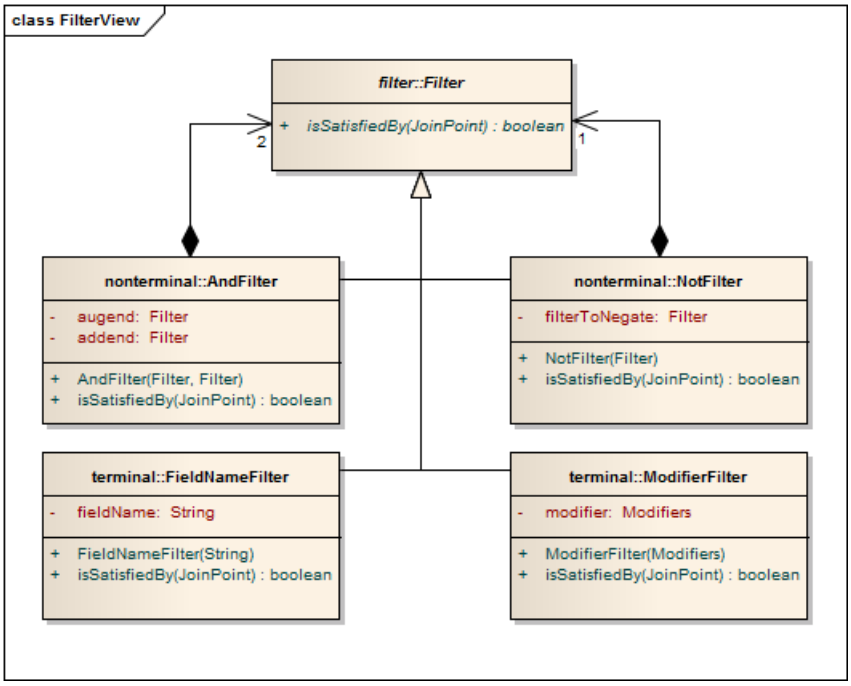

Figure 9: JaInFra Filtering

Es wird zwischen nonterminal und terminal Filter unterschieden. Terminalfilter sind, wie der Name sagt, Terminal, können also nicht mit anderen Filtern kombiniert werden.

Listing 2: TerminalFilter

```
1 Filter f = new MethodNameFilter("foo");
```

Nonterminale Filter beinhalten ein oder mehrere Filter. Diese können wiederum nonterminal sein oder aber terminal.

Listing 3: NonTerminal mit Terminal Filter

```
1 Filter f = new NotFilter(new MethodNameFilter("foo"));
```

Somit lassen sich alle erdenklichen Kombinationen erstellen. Die genaue Verwendung und Beschreibung aller Filter, kann im DeveloperGuide nachgelesen werden:

04_Architektur_und_Design/00_Developer_Guide/Developer_Guide_VX.X.pdf

4.4 Aspects

Die AspectJ Klassen (.aj Dateien) werden zur Laufzeit (LTW) mit der zu tracenden Applikation verwoben. Sowohl im Basic als auch im Advanced Modus gibt es unterschiedliche Aspects.

4.4.1 BasePointCutDefinition

Damit man bei den Poincuts in den einzelnen Aspects nicht mehrere gleiche Deklarationen hat (Duplicated Code), wurde die Klasse BasePointCutDefinition eingeführt:

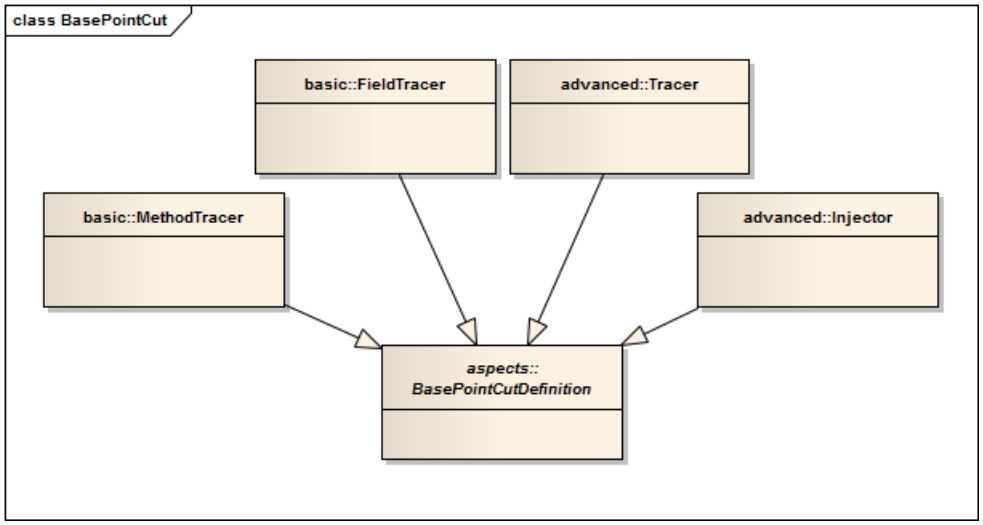

Figure 10: Aspects erben von der BasePointCutDefinition Klasse

Somit können gemeinsam genutzte Deklarationen ausgelagert werden.

4.4.2 AspectHelper

Neben Poincut Deklarationen und AspectJ Code beinhalten die AspectJ Klassen auch Java Code. Die AspectHelper Klassen werden ebenfalls benötigt um Duplicated Code zu minimieren:

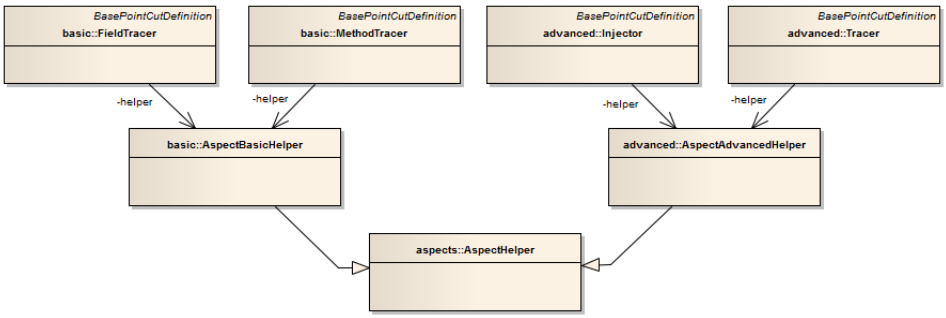

Figure 11: Grobübersicht der Hilfsklassen

Weil die AspectJ Pointcuts immer static sind und damit eine Lösung mit Vererbung nicht möglich ist, Verwenden die AspectJ Klassen jeweils ein Datenfeld helper.

4.4.3 Anwendung der Hilfsklassen im Advanced Modus

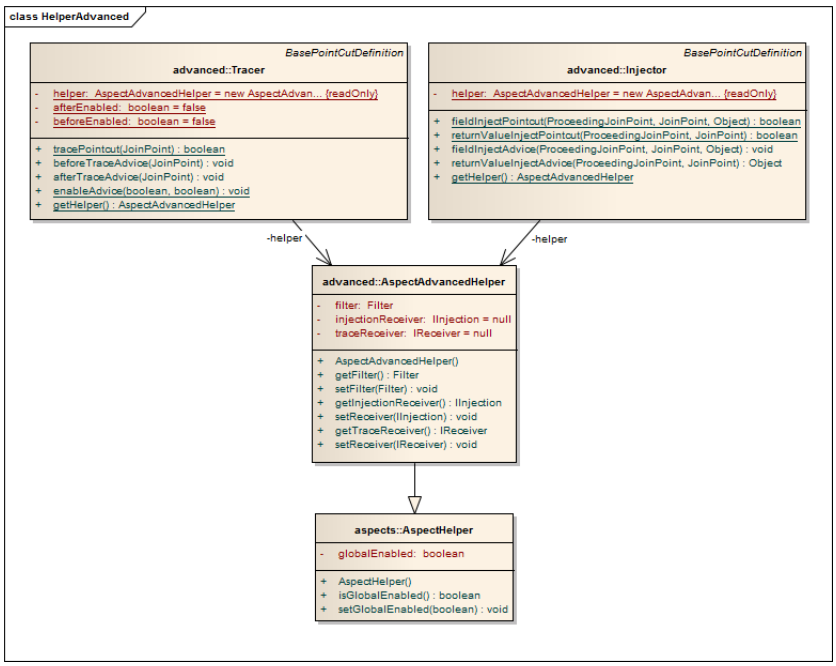

Figure 12: Die AspectJ Advanced Klassen mit dem Helper Datenfeld

Beide AspectJ Klassen im Advanced Modus (Tracer, Injector) besitzen ein Datenfeld helper. Somit können Receiver und der aktuelle Filter mittels getReceiver() bzw. getFilter() abgefragt werden. Zusätzlich regelt die Oberklasse AspectHelper das ein- und ausschalten eines Pointcuts.

4.5 Callbacks

Die beiden Modi von JaInFra bieten unterschiedliche Callback Interfaces:

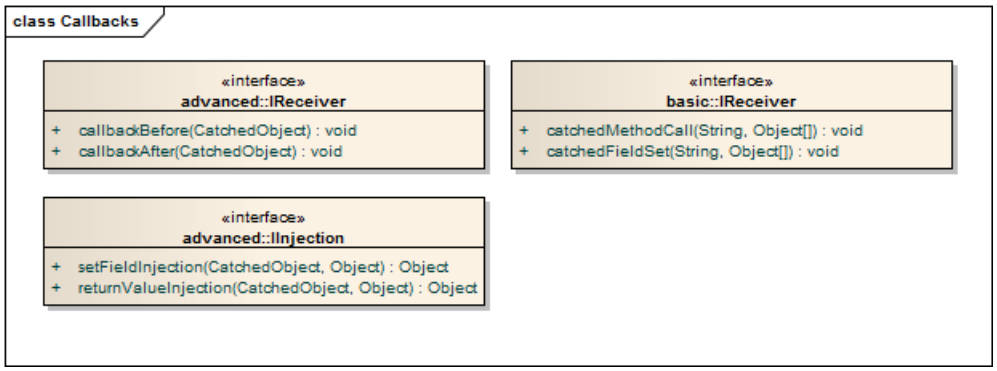

Figure 13: JaInFra Callbacks

Während beim IReceiver Interface des Basic Modus lediglich ein String des Methoden- bzw. Feldernamen und deren Argumente zurückgegeben wird, erhält man im Advanced Modus ein CatchedObject zurück:

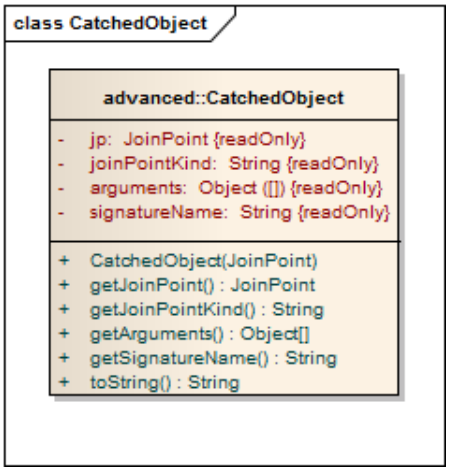

Figure 14: JaInFra CatchedObject Klasse

Der Einsatz von Callback Interfaces bietet dem Kunden die Möglichkeit selbst zu entscheiden, was bei einem Callback gemacht werden muss.

4.5.1 Das CatchedObject

Die CatchedObject Klasse kapselt den AspectJ JoinPoint und stellt zusätzlich die wichtigsten Methoden wie getSignatureName() zur Verfügung. Somit muss der Kunde nicht zwangsläufig AspectJ Kenntnisse haben. Falls gewünscht, kann mittels getJoinPoint() weiterhin auf das JoinPoint Objekt zugegriffen werden.

4.5.2 Wann erfolgen die Callbacks?

Im Basic Modus wird entweder ein Callback ausgelöst, wenn ein Feld (catchedFieldSet) gesetzt wird, oder vor dem Aufruf einer Methode (catchedMethodCall).

Beim Advanced Modus wird zuerst der festgelegte Filter ausgewertet und anschliessend je nach gewähltem Advicetype (Before, After) der Callback ausgelöst.

4.6 Inspecting mit Reflection

Sowohl der Basic als auch der Advanced Modus erben von der Klasse JaInFra. Diese Oberklasse greift mittels "Method Chaining" auf folgende Methoden der Klasse Inspecting zu:

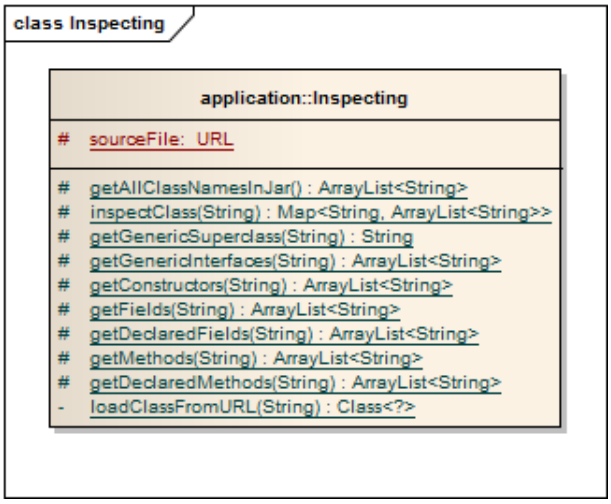

Figure 15: JaInFra Inspecting Klasse

Methode	Beschreibung
getAllClassNamesInJar()	Liefert alle Java .class Files in einem JAR Archiv.
inspectlass()	Liefert eine Map mit sämtlichen Informationen zur Klasse. Dafür werden die folgenden get*() Methoden aufgerufen und in einer Map zusammengefügt.
getGenericSuperclass()	Liefert die Oberklasse. (extends)
getGenericInterfaces()	Liefert das Interface. (implements)
getConstructor()	Liefert die Konstruktoren.
getFields()	Liefert sämtliche Felder der Oberklasse.
getDeclaredFields()	Liefert alle deklarierten Felder dieser Klasse. Dies beinhaltet Felder mit den Modifiers public, private, protected und default.
getdMethods()	Liefert die Methoden der Oberklasse
getDeclaredMethods()	Liefert alle deklarierten Methoden dieser Klasse. Dies beinhaltet Methoden mit den Modifiers public, private, protected und default.

Die Inspection Klasse ermöglicht die statische Analyse des unbekannten Programms. Somit können beispielsweise Methoden oder Felder der zu tracenden Applikation herausgefunden werden. Mit diesen Informationen können im späteren Verlauf des Tracings die Filter genauer deklariert werden.

4.7 Weitere Entscheide

4.7.1 weaveAspects() - Load Time Weaving

Das Verweben der Aspects mit dem Unknown.jar erfolgt bei JaInFra zur Ladezeit (LTW). Die Klasse WeavingURLClassLoader hilft, das weaving programmatisch vorzunehmen.

Listing 4: Verwendung WeavingURLClassLoader in der Methode weaveAspects

```
1 WeavingURLClassLoader (sourceList, aspectsList, Thread.currentThread().
    getContextClassLoader());
```

Wichtig ist, dass der "Aspect" Pfad (hier aspectsList) ein XML File enthält, bei dem die Aspects definiert sind. Somit weiss der Weaver welchen Code (hier sourceList) mit welchen Aspects zu weaven sind. Danach kann man die Main Methode der gewünschten Klasse via Reflection starten.

Listing 5: Beispiel aop-ajc.xml

```
1 <aspectj>
2    <aspects>
3       <aspect name="com.jainfra.aspects.advanced.Tracer"/>
4       <aspect name="com.jainfra.aspects.basic.MethodTracing"/>
5       <aspect name="com.jainfra.aspects.basic.FieldTracing"/>
6       <aspect name="com.jainfra.aspects.advanced.Injector"/>
7    </aspects>
8 </aspectj>
```

4.7.2 AspectJ und Runtime Weaving

Die opitmale Lösung wäre der Einsatz von Runtime Weaving. Leider bietet AspectJ kein Unterstützung für die Anwendung von Advices zur Laufzeit.

Entscheid: Das Weaving erfolgt zur Ladezeit (LTW). Mittels AspectJ if() "Checks" in den Pointcuts kann aber trotzdem zur Laufzeit eine Bedingung überprüft werden. Dementsprechend wird dann im Advice ein Callback gemacht oder nicht.

4.7.3 AspectJ Annotations

Bei der aspektorientierten Programmierung gibt es zwei verschiedene Arten.

Standard:

Listing 6: AspectJ Pointcut ohne Annotations

```
1 pointcut anyCall() : call(* *.*(..));
```

Annotations:

Listing 7: AspectJ Pointcut mit Annotation

```
1 @Pointcut("call(* *.*(..))")
2    void anyCall() {}
```

Beide Pointcutdeklarationen sind aequivalent.

Entscheid: Die Programmierung wird im Annotations Stil vorgenommen. Dies bringt uns den Vorteil, dass das Kompilieren sowohl mit dem AspectJ Compiler (ajc) als auch mit dem Java Compiler (javac) möglich ist.

4.7.4 Load Time Weaving - Suppress Warnings

Weil die Aspects bzw. Advices erst zur Load Time "verwoben" werden, kann die AspectJ IDE kein "matching" zur Compiletime machen. Damit diese Warnungen verschwinden, muss vor jedem Advice folgende AspectJ Codezeile eingefügt werden:

Listing 8: SuppressAjWarnings

```
1 @SuppressAjWarnings({"adviceDidNotMatch"})
```

4.7.5 call() vs. exection() im Basic Modus

Wir mussten im Basic Modus entscheiden, ob wir für das Tracen von Methoden calls oder execution nehmen sollten.

Entscheid: Im Basic Modus wird call() verwendet. Es ist interessanter, wann eine Methode aufgerufen wird, als das Tatsächliche ausführen der Methode. So können auch "Fremdaufrufe" getraced werden (Beispiel: Random.nextInt())

4.8 Verworfene Ideen

4.8.1 Callbacks über Sockets

Es wurde getestet, ob anstelle von zwei Threads, zwei eigene Prozesse gestartet werden sollen. Somit hätte ein AWT-EventQueue Thread niemals den AWT-EventQueue Thread des Kunden blockieren können. (Beispielsweise bei der Kreierung eines GUIs um Breakpoints zu setzen). Die Lösung mit zwei Prozessen hätte aber verlangt, dass man die Kommunikation zwischen den Programmen mit Sockets löst. Dies war dann allerdings zu Umständlich, da alle Callbacks über den Socket verschickt wurden. Das Setzen der Filter musste ebenfalls über einen Socket erledigt werden. Also musste man einen "Server" in das Unbekannte Programm einweben. Dies führte zu einer schlechteren Performance. Wir haben aus diesem Grund die Threadlösung beibehalten. Der Kunde kann selbstverständlich im Callbackinterface einen zweiten Prozess starten und würde allfällige Threadverklemmungen somit beheben.

4.8.2 Markerinterface für Receiver

Für das Tracen im Basicmodus sowie im Advancedmodus gibt es ein Callbackinterface. Beide heissen IReceiver. Wir haben uns gefragt, ob wir ein "Oberinterface" erstellen wollen und beide IReceivers davon ableiten lassen. Somit hätten wir eine einheitliche Methode gehabt um das Callbackinterface zu setzen. Jedoch wäre dann ein instanceof und ein Cast in das richtige Callbackinterface bei jedem Advice erfoderlich gewesen. Aus Performancegründen haben wir die Markerinterface Idee wieder gestrichen.

4.9 Einschränkungen

4.9.1 Maximale Länge einer Methode in Java

Die maximale Länge einer Java Methode ist gemäss Spezifikationen der Java Virtual Machine [jvm] auf 64 KB beschränkt.
Dies kann beim weaven zu Problemen führen. Existiert nähmlich in einer zu tracenden Applikation ein sehr lange Methode, kann es vorkommen dass dieser Wert überschritten wird. Dann wir folgenden Fehlermeldung generiert:

Listing 9: Maximale Länge einer Methode überschritten

```
1 org.aspectj.bridge.AbortException: problem generating method
2 <package>.<class>.<method> : Code size too big: 228330
```

4.9.2 Gleicher Fully Qualified Name

Es kann vorkommen, dass der Fully Qualified Name (FQN) einer Klasse innerhalb der zu tracenden Applikation identisch ist, wie ein FQN im Programm, welches die JaInFra Library verwendet. Dies hat zur Folge, das der WeavingClassLoader über die Methode loadClass() die "falsche" Methode starten könnte. Die Reihenfolge der Classpaths kann nicht eingestellt werden. Es resultiert folgenden Fehlermeldung:

Listing 10: Exception bei der FQN Problematik

```
1 Exception in thread "main" java.lang.NoClassDefFoundError: main (wrong name
      : Main)
```

Literaturverzeichnis

[aop] Funktionsweise aop-ajx.xml. Website. http://www.eclipse.org/aspectj/doc/next/devguide/ltw-configuration.html#configuring-load-time-weaving-with-aopxml-files; besucht im November 2011.

[int] Interpreter Pattern. Website. http://en.wikipedia.org/wiki/Interpreter_pattern; besucht im Dezember 2011.

[jvm] Java Virtual Machine Spezifikationen. Website. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html; besucht im November 2011.

[url] AspectJ WeavingURLClassloader. Website. http://www.eclipse.org/aspectj/doc/next/weaver-api/org/aspectj/weaver/loadtime/WeavingURLClassLoader.html; besucht im Oktober 2011.

JaInFra
Developer Guide

December 20, 2011

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	24.10.2011	f1egli	HowTo aus A&D übernommen
1.0rc02	25.10.2011	f1egli	Developer Guide in zwei Kategorien unterteilt
1.0rc03	01.11.2011	f1egli	Basic Modus ausgearbeitet
1.0rc04	22.11.2011	f1egli	Advanced Modus eingeführt
1.0	25.11.2011	f1egli	Version 1.0
2.0rc01	06.12.2011	f1egli	Filter ergänzt, Injection und Logger eingeführt
2.0	12.12.2011	f1egli	Version 2.0
3.0rc01	12.12.2011	f1egli	MethodParameterTypeFilter hinzugefügt
3.0rc02	13.12.2011	f1egli	ClassNameFilter hinzugefügt
3.0rc03	13.12.2011	f1egli	Beispiele ergänzt
3.0rc04	20.12.2011	f1egli	Codesnippet akualisiert
3.0	20.12.2011	f1egli, dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Developer Guide **3**

4 Generelles **3**
4.1 JavaDoc . . . 3
4.2 Inspecting . . . 3
4.2.1 Alle Klassen in einem Jar File auflisten . . . 3
4.2.2 Informationen einer einzelnen Klasse erhalten . . . 3

5 Basic **6**
5.1 Callbacks mittels IReceiver . . . 6
5.2 Main Methode für foo(), bar() und x, y Inspizierung . . . 6
5.2.1 Mehrere Filterwörter angeben . . . 7
5.2.2 Alle Felder bzw. Methoden Tracen . . . 7
5.2.3 Nur Felder oder nur Methoden Tracing . . . 7

6 Advanced **9**
6.1 Tracing . . . 9
6.1.1 CallBackInterface . . . 9
6.1.2 Advice [adv] . . . 10
6.2 Injection . . . 12
6.3 Filter . . . 13
6.3.1 NonTerminal Filter . . . 13
6.3.2 Terminal Filter . . . 13

7 Logger **16**

8 Beispiele **17**
8.1 Filter . . . 17
8.2 Verschachtelung [Lad10] eines Programmes analysieren . . . 17
8.3 Breakpoints simulieren . . . 19
8.4 Eigener Filter . . . 22
8.5 Verwendung von configureApplication . . . 22

Literaturverzeichnis **23**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Developer Guide

JaInFra bietet zwei Möglichkeiten zur Benutzung an:

- Basic: in diesem Modus soll es möglichst einfach sein, Methoden und Felder zu tracen. Typischerweise zur Benutzung einfacher CLI-Applikationen.
- Advanced: in diesem Modus ist der Befehlssatz viel grösser. JaInFra kann also spezifischer genutzt werden. Typischerweise zur Benutzung von GUI-Erstellung sowie komplizierte traces.

4 Generelles

4.1 JavaDoc

Im JaInFra.jar gibt es einen Ordner JavaDoc. Dieser sollte bei Benutzung von JaInFra verwendet werden um allfällige Fragen zu klären.

4.2 Inspecting

Das Inspizieren von Jar Files und einzelnen Klassen funktioniert sowohl im Basic, wie auch im Advanced Modus.

4.2.1 Alle Klassen in einem Jar File auflisten

Möchte man wissen, welche Klassen sich in einem Jar befinden, bietet JaInFra die Methode getAllClassNamesInJar() an:

Listing 1: getAllClassNamesInJar()

```
1 JaInFraBasic jif = JaInFraBasic.getInstance();
2 jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"));
3
4 System.out.println(jif.getAllClassNamesInJar());
```

Listing 2: Consolenoutput von Listing 1

```
1 [Wuerfel.class, View$1.class, View$2.class, View.class]
```

4.2.2 Informationen einer einzelnen Klasse erhalten

Mit der Methode inspectClass("Classname") erhält man alle Informationen, die über diese Klasse vorhanden sind. Folgender Code soll dies verdeutlichen:

Listing 3: Alle Informationen der Klasse Wuerfel

```
1  JaInFraBasic jif = JaInFraBasic.getInstance();
2  jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"));
3
4  Map<String, ArrayList<String>> classInfoMap = jif.inspectClass("Wuerfel");
5
6  for (String key : classInfoMap.keySet()) {
7     System.out.println("### " + key);
8        for (int i = 0; i < classInfoMap.get(key).size(); i++) {
9           System.out.println(classInfoMap.get(key).get(i));
10       }
11 }
```

Listing 4: Consolenoutput von Listing 3

```
1  ### methods
2  public static void Wuerfel.main(java.lang.String[])
3  public int Wuerfel.wuerfle()
4  public int Wuerfel.getAnzahl()
5  public int Wuerfel.getSumme()
6  public boolean java.lang.Object.equals(java.lang.Object)
7  public java.lang.String java.lang.Object.toString()
8  public native int java.lang.Object.hashCode()
9  public final native java.lang.Class java.lang.Object.getClass()
10 ### genericInterfaces
11 ### genericSuperclass
12 class java.lang.Object
13 ### declaredFields
14 private int Wuerfel.anzahl
15 private int Wuerfel.summe
16 ### declaredMethods
17 public static void Wuerfel.main(java.lang.String[])
18 private void Wuerfel.setAnzahl(int)
19 private void Wuerfel.setSumme(int)
20 public int Wuerfel.wuerfle()
21 public int Wuerfel.getAnzahl()
22 public int Wuerfel.getSumme()
23 ### constructors
24 public Wuerfel()
25 ### fields
```

Natürlich können auch nur einzelne Informationen (zum Beispiel nur die Konstruktoren) herausgefunden werden. Folgende Liste zeigt, welche Methoden von JaInFra angeboten werden:

- String getGenericSuperclass(String className)
  Liefert die Oberklasse.
- ArrayList<String>getGenericInterfaces(String className)
  Liefert die implementierten Interfaces.
- ArrayList<String>getConstructors(String className)
  Liefert die implementierten Konstruktoren.
- ArrayList<String>getFields(String className)
  Liefert die Felder der Oberklasse.

- ArrayList<String>getDeclaredFields(String className)
  Liefert die Felder dieser Klasse.
- ArrayList<String>getMethods(String className)
  Liefert die Methoden der Oberklasse.
- ArrayList<String>getDeclaredMethods(String className)
  Liefert die Methoden dieser Klasse.

5 Basic

Im Basic Modus können Felder, sowie Methoden getraced werden. Wichtig ist hierbei das Methoden bei einem **call()** und Felder bei einem **set()** inspiziert werden. Call() bedeutet, dass bei der Aufrufenden Klasse getraced wird. Set() bedeutet, dass getraced wird, wenn das Feld geschrieben (nicht gelesen) wird.

5.1 Callbacks mittels IReceiver

Als erstes muss ein Receiver implementiert werden. Sobald ein Feld oder eine Methode getraced wird, ruft JaInFra die richtige Methode in dieser Klasse auf.

Listing 5: MyJaInFraReceiver implementiert IReceiver. Ausgaben mittels Sysout

```
1  import com.jainfra.application.basic.IReceiver;
2
3  public class MyJaInFraReceiver implements IReceiver {
4
5      @Override
6      public void catchedMethodCall(String methodSignature, Object[] arguments
           ) {
7          System.out.print("Komme in Methode " + methodSignature);
8
9          printToSysout(arguments);
10     }
11
12     @Override
13     public void catchedFieldSet(String fieldName, Object[] value) {
14         System.out.print("Setze Feld " + fieldName);
15         printToSysout(value);
16     }
17
18     private void printToSysout(Object[] arg1) {
19         if (arg1.length != 0) {
20             System.out.print(" --- : ");
21         }
22         for (int i = 0; i < arg1.length; i++) {
23             System.out.print(arg1[i] + " ");
24         }
25         System.out.println();
26     }
27 }
```

5.2 Main Methode für foo(), bar() und x, y Inspizierung

Danach kann die Main Methode erstellt werden. In diesem Beispiel werden die Methoden foo und bar, sowie die Felder x und y getraced:

Listing 6: Main Programm und JaInFraBasic

```
1  import java.net.URL;
2  import java.util.ArrayList;
3  import java.util.Map;
4
5  import com.jainfra.application.basic.IReceiver;
6  import com.jainfra.application.basic.JaInFraBasic;
```

```
7
8  public class Main {
9      public static void main(String[] args) {
10
11         JaInFraBasic jif = JaInFraBasic.getInstance();
12         IReceiver myReceiver = new MyJaInFraReceiver();
13
14         try {
15           jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"));
16           jif.configureMethodTracing(myReceiver, "set", "bar");
17           jif.configureFieldTracing(myReceiver, "x", "y");
18           jif.start();
19         } catch (Exception e) {
20           e.printStackTrace();
21         }
22     }
23 }
```

5.2.1 Mehrere Filterwörter angeben

Die Methoden configureFieldTracing() und configureMethodTracing() können beliebig viele Strings entgegennehmen. Möchte man zum Beispiel nur das Feld "anzahl" Tracen erreicht man dies mit:

- jif.configureFieldTracing(myReceiver, "anzahl");

Möchte man N-Felder Tracen:

- jif.configureFieldTracing(myReceiver, "x", "x1", "x2", "xn");

  Das gleiche funktioniert natürlich auch mit der configureMethodTracing() Methode.

5.2.2 Alle Felder bzw. Methoden Tracen

Möchte man alle Felder bzw. Mehtoden Tracen kann man dies mit einem leeren String erreichen:

- jif.configureFieldTracing(myReceiver, "");
- jif.configureMethodTracing(myReceiver, "");

Eine Andere Möglichkeit wäre, keine Expression anzugeben:

- jif.configureFieldTracing(myReceiver);
- jif.configureMethodTracing(myReceiver);

5.2.3 Nur Felder oder nur Methoden Tracing

Möchte man nur Felder bzw. Methoden Tracen, kann die jeweilige Methode einfach ausgelassen werden.

Listing 7: Nur Felder x und y Tracen

```
1 .....
2 public class Main {
3 .....
4     jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"));
5     jif.configureFieldTracing(myReceiver, "x", "y");
6     jif.start();
```

6 Advanced

Mit dem Advanced Modus hat man die Möglichkeit, das Tracing mit Filtern gezielt anwenden zu können. Ausserdem besteht die Möglichkeit, Code zur Laufzeit zu verändern.

Das Grundgerüst, um eine Unbekannte Applikation zu starten sieht folgendermassen aus:

Listing 8: starten einer Applikation

```
1  public class Main {
2     public static void main(String[] args) {
3
4        JaInFraAdvanced jif = JaInFraAdvanced.getInstance();
5        try {
6           jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"))
                ;
7           jif.start();
8        } catch (Exception e) {
9           e.printStackTrace();
10       }
11    }
12 }
```

Man beachte, dass so die Applikation zwar gestartet wird, jedoch keine Traces/Injections angesprochen werden, da man sie (noch) nicht definiert hat.

6.1 Tracing

Mit den Tracing Mechanismen können Applikationen einen Callback zu einem bestimmten Filter auslösen. Als Beispiel: Es soll eine Meldung geben, nachdem ein Feld mit dem Namen "x" gelesen wird.

6.1.1 CallBackInterface

Das Interface IReceiver muss Kundenseitig implementiert werden, damit die Callbacks, welche JaInFra auslöst, ausgeführt werden können. Eine einfache Implementation des Interfaces könnte wie folgt aussehen:

Listing 9: Implementation von IReceiver

```
1  import org.aspectj.lang.JoinPoint;
2  import com.jainfra.application.advanced.CatchedObject;
3  import com.jainfra.application.advanced.IReceiver;
4
5  public class MyJaInFraReceiver implements IReceiver {
6
7     @Override
8     public void callbackBefore(CatchedObject co) {
9        System.out.println("Before " + co);
10    }
11    @Override
12    public void callbackAfter(CatchedObject co) {
13       System.out.println("After " + co);
14    }
15 }
```

Es ist zu beachten, dass das Interface IReceiver bei dem Basic und bei dem Advanced Modus gleich heissen. Im Falle des Advanced Modus muss das Importstatement folglich:

```
import com.jainfra.application.advanced.IReceiver;
```

lauten.

Die Methoden, welche überschrieben werden, bedeuten folgendes:

- callbackBefore(CatchedObject co)
  Diese Methode wird von JaInFra aufgerufen, bevor der Pointcut (die Stelle wo getraced wird) durchlaufen ist.
- callbackAfter(CatchedObject co)
  Diese Methode wird von JaInFra aufgerufen, nachdem der Pointcut durchlaufen ist.

Die Klasse CatchedObject kapselt einen JoinPoint. Folgende Methoden stehen in dieser Klasse bereit:

- public final JoinPoint getJoinPoint()
  Liefert den instanzierten JoinPoint zurück.
- public final String getJoinPointKind()
  Liefert die JoinPoint Art zurück (call, get, etc.)
- public final Object[] getArguments()
  Liefert die Argumente als Objectarray.
- public final String getSignatureName()
  Liefert den Signaturennamen zurück.
- public String toString()
  Führt "return jp.toString()" aus.

Die CallBack Klasse, hier MyJaInFraReceiver, muss nun bei JaInFra registriert werden. Man beachte, dass gegen das Interface deklariert wird.

```
IReceiver myReceiver = new MyJaInFraReceiver();
jif.configureReceiver(myReceiver);
```

6.1.2 Advice [adv]

Advices können über die Methode jif.enableAdvice(boolean before, boolean after) ein - bzw. ausgeschaltet werden. Folgende vier Möglichkeiten können daraus entstehen:

- jif.enableAdvice(false, false);
  Dies ist die standard Einstellung. Weder Before noch After Advices werden aktiviert.
- jif.enableAdvice(true, false);
  Nur Before Advices werden aktiviert - dementsprechend wird JaInFra auch nur die Methode callbackBefore() als Callback ausführen.

- jif.enableAdvice(false, true);
  Nur After Advices werden aktiviert - dementsprechend wird JaInFra auch nur die Methode callbackAfter() als Callback ausführen.
- jif.enableAdvice(true, true);
  Beide Advices werden aktiviert

6.2 Injection

Auch bei der Injection gibt es ein CallBackInterface. Es heisst IInjection und befindet sich im Package:

`com.jainfra.application.advanced.IInjection`

Eine einfache Implementation des Interfaces (ohne Veränderung irgendwelcher Daten) könnte wie folgt aussehen:

Listing 10: Implementation von IInjection

```
1  import com.jainfra.application.advanced.CatchedObject;
2  import com.jainfra.application.advanced.IInjection;
3
4  public class MyJaInFraInjection implements IInjection {
5
6      @Override
7      public Object setFieldInjection(CatchedObject co, Object oldValue) {
8          System.out.println("set Field injection " + oldValue);
9          return  oldValue;
10     }
11
12     @Override
13     public Object returnValueInjection(CatchedObject co, Object oldValue) {
14         System.out.println("im returnValueInjection " + oldValue);
15         return oldValue;
16     }
17
18 }
```

Die Methoden, welche überschrieben werden, bedeuten folgendes:

- Object setFieldInjection(CatchedObject catchedObject, Object oldValue)
  Diese Methode wird von JaInFra aufgerufen, wenn ein Datenfeld gelesen oder gesetzt wird. Der Wert des Feldes wird dann mit dem returnierten Wert der Methode setFieldInjection überschrieben.

- Object returnValueInjection(CatchedObject catchedObject, Object oldValue)
  Diese Methode wird von JaInFra aufgerufen, wenn eine Methode einen Wert returniert. Wird also eine Methode Injected, die einen String returniert, kann man mit returnValueInjection, diesen Wert verändern. (Zum Beispiel mit return "InjectedString";

Für die Erklärung zu CatchedObject siehe Unterkapitel Tracing/CallBackInterface.

Die CallBackKlasse, hier MyJaInFraInjection, muss nun bei JaInFra registriert werden. Man beachte, dass gegen das Interface deklariert wird.

```
IInjection myInjectionReceiver = new MyJaInFraInjection();
jif.configureReceiverForInjection(myInjectionReceiver);
```

6.3 Filter

Die Filter sind das Herzstück des Advanced Modus. Sie können beliebig kombiniert und ergänzt werden, um die Tracing- und Injectionkriterien einzugrenzen. Mit folgenden Befehlen wird ein Filter für das Tracing gesetzt :

```
Filter f = new .... ;
jif.setFilter(f);
```

Um einen Filter für Injection zu setzen benötigt man :

```
Filter f = new .... ;
jif.setFilterForInjection(f);
```

Die konkreten Implementierungen sind dann in zwei Typen unterteilt:

- NonTerminal Filter
- TerminalFilter

6.3.1 NonTerminal Filter

NonTerminale Filter werden mit mindestens einem Terminalen oder nicht Terminalen Filter ergänzt. Es gibt die folgenden drei NonTerminalFilter:

- AndFilter
  Kombiniert zwei Filter, wobei beide erfüllt sein müssen.

  ```
  Filter f = new AndFilter(augend, addend);
  ```

- OrFilter
  Kombiniert zwei Filter, wobei einer der beiden erfüllt sein muss.

  ```
  Filter f = new OrFilter(leftFilter, rightFilter);
  ```

- NotFilter
  Negiert den Filter

  ```
  Filter f = new NotFilter(filterToNegate)
  ```

6.3.2 Terminal Filter

TerminalFilter können für sich alleine stehen, oder werden mit den NonTerminal Filtern verknüpft.

6.3.2.1 AlwaysSatisfiedFilter

Dieser Filter ist immer zufrieden gestellt. So kann man beispielsweise alles Tracen lassen.

```
Filter f = new AlwaysSatisfiedFilter();
```

6.3.2.2 NeverSatisfiedFilter

Dieser Filter ist nie zufrieden gestellt. Mit diesem Filter wird also nichts getraced. Dies kann nützlich sein, wenn vorgängig ein Filter gesetzt wurde und man zwischenzeitlich nichts mehr Inspizieren möchte.

```
Filter f = new NeverSatisfiedFilter();
```

6.3.2.3 FieldNameFilter

Mittels einem String kann angegeben werden, wie ein Feld heissen soll. Der String wird als Regular Expression intepretiert.

```
Filter f = new FieldNameFilter(fieldName);
```

6.3.2.4 MethodNameFilter

Mittels einem String kann angegeben werden, wie eine Methode heissen soll. Der String wird als Regular Expression intepretiert.

```
Filter f = new MethodNameFilter(methodName);
```

6.3.2.5 ClassNameFilter

Mittels einem String kann angegeben werden, wie eine Klasse heissen soll. Der String wird als Regular Expression intepretiert.

```
Filter f = new ClassNameFilter("Wuerfel");
```

6.3.2.6 PackageNameFilter

Mittels einem String kann angegeben werden, wie das Package heissen soll. Der String wird als Regular Expression intepretiert.

```
Filter f = new PackageNameFilter(packageName);
```

6.3.2.7 JoinPointCategoryFilter

Über einen Enum, den JoinPointCategory [jpk], kann der Pointcutmechanismus (get, call, execution etc.) angegeben werden.

```
Filter f = new JoinPointCategoryFilter(joinPointCategory);
```

Der Enum beinhaltet folgende Kategorien:

- METHOD_EXECUTION
- METHOD_CALL
- CONSTRUCTOR_EXECUTION

- CONSTRUCTOR_CALL
- CLASS_INITIALIZATION
- FIELD_READ_ACCESS
- FIELD_WRITE_ACCESS
- EXECPTION_HANDLER_EXECUTION
- OBJECT_INITIALIZATION
- OBJECT_PRE_INITIALIZATION

6.3.2.8 ModifierFilter

Über einen Enum, den Modifiers, kann der Modifikator (pulbic, static etc.) angegeben werden.

```
Filter f = new ModifierFilter(modifier);
```

Der Enum beinhaltet folgende Modifikatoren:

- PUBLIC
- PRIVATE
- PROTECTED
- STATIC
- FINAL
- SYNCHRONIZED
- VOLATILE
- TRANSIENT
- NATIVE
- INTERFACE
- ABSTRACT
- STRICT

6.3.2.9 ReturnTypeFilter

Mittels einem String kann der Rückgabetyp (int, float, void, Punkt etc.) angegeben werden.

```
Filter f = new ReturnTypeFilter(returnType);
```

6.3.2.10 MethodParameterCountFilter

Mittels einem int kann angegeben werden, wie viele Parameter eine Methode enthalten soll.

```
Filter f = new MethodParameterCountFilter(parameterCount);
```

6.3.2.11 MethodParameterTypeFilter

Mittels einem String kann angegeben werden, welcher Parametertyp in einer Methode mindestens einmal vorhanden sein soll.

```
Filter f = new MethodParameterTypeFilter(parameterTypeName);
```

7 Logger

Um JaInFra selbst zu loggen, können ebenfalls die Filter verwendet werden. Mit der Methode:

```
public void configureLogging(String filename, Filter f);
```

kann der Logging Prozess gestartet werden. Dies funktioniert im Basic, wie auch im Advanced Modus. Mittels filename kann man den Dateinamen des LogFiles bestimmen. Folgendes Listing zeigt, wie man das Logging aktivieren könnte:

Listing 11: Logger beim Kunden aktivieren

```
1  import java.net.MalformedURLException;
2  import java.net.URL;
3  import java.util.Random;
4
5  import org.aspectj.lang.JoinPoint;
6
7  import com.jainfra.application.advanced.IReceiver;
8  import com.jainfra.application.advanced.JaInFraAdvanced;
9  import com.jainfra.application.advanced.filter.Filter;
10 import com.jainfra.application.advanced.filter.terminal.
       AlwaysSatisfiedFilter;
11
12 public class Main {
13    public static void main(String[] args) {
14       JaInFraAdvanced jif = JaInFraAdvanced.getInstance();
15       try {
16
17          jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"))
                ;
18          jif.configureLogging("log.txt", new AlwaysSatisfiedFilter());
19
20          jif.start();
21
22       } catch (Exception e) {
23          e.printStackTrace();
24       }
25    }
26 }
```

Listing 12: Auszug eines LogFiles

```
1 Tue Dec 06 16:10:44 CET 2011 AFTER - method-execution of void com.jainfra.
    application.JaInFra.configureLogging(String, Filter) Args: log.txt com.
    jainfra.application.advanced.filter.terminal.
    AlwaysSatisfiedFilter@1b52513a
2 Tue Dec 06 16:10:44 CET 2011 BEFORE - method-execution of void com.jainfra.
    application.JaInFra.start()
```

8 Beispiele

8.1 Filter

Zum Ideenanstoss hier ein paar Filterbeispiele:

- Die Filter FieldNameFilter, MethodNameFilter, ClassNameFilter und PackageNameFilter, können als Argument einen Regex String entgegennehmen. Möchte man zum Beispiel die Methoden: get, set, getter, setter etc. tracen kann man das mit nur einem Filterstatement erreichen:

  ```
  Filter f = new MethodNameFilter("[gs]et.*");
  ```

- Einen Filter der dann greift, wenn die Variable x gelesen wird:

  ```
  Filter f = new AndFilter(
    JoinPointCategoryFilter(JoinPointCategory.FIELD_READ_ACCESS),
    FieldNameFilter("x"));
  ```

- Einen Filter der dann greift, wenn eine private Methode ohne Rückgabewert aufgerufen oder ausgeführt wird:

  ```
  Filter f =
  new AndFilter(
    ReturnTypeFilter("void"),
    ModifierFilter(Modifiers.PRIVATE));
  ```

8.2 Verschachtelung [Lad10] eines Programmes analysieren

Möchte man die Verschachtelung (Aufrufe von Methoden, setzen von Felder etc.) eines Programms sehen kann man wie folgt vorgehen. Als erstes muss ein entsprechendes Interface implementiert werden:

Listing 13: MyJaInFraReceiver für Verschachtelung

```
1 import org.aspectj.lang.JoinPoint;
2 import com.jainfra.application.advanced.CatchedObject;
3 import com.jainfra.application.advanced.IReceiver;
4
5 public class MyJaInFraReceiver implements IReceiver {
6    private int callDepth;
7    @Override
```

```
8     public void callbackBefore(CatchedObject co) {
9        print("Before", co.getJoinPoint());
10       callDepth++;
11    }
12    @Override
13    public void callbackAfter(CatchedObject co) {
14       callDepth--;
15       print("After", co.getJoinPoint());
16    }
17    private void print(String prefix, JoinPoint jp) {
18       for (int i = 0; i < callDepth; i++) {
19          System.out.print("  ");
20       }
21       System.out.print(prefix + " ");
22       Object[] args = jp.getArgs();
23       if (args.length == 0) {
24          System.out.println(jp);
25       } else {
26          System.out.print(jp + " Args: ");
27          for (Object object : args) {
28             System.out.print(object + " ");
29          }
30          System.out.println();
31       }
32    }
33 }
```

Danach kann man den Code erstellen, welcher JaInFra und MyJaInFraReceiver verwendet:

Listing 14: Klasse des Kunden

```
1  public class Main {
2     public static void main(String[] args) {
3        JaInFraAdvanced jif = JaInFraAdvanced.getInstance();
4        IReceiver myReceiver = new MyJaInFraReceiver();
5        try {
6
7           jif.configureApplication(null, new URL("file:///tmp/Unknown.jar"))
               ;
8           jif.enableAdvice(true, true);
9           jif.configureReceiver(myReceiver);
10
11          Filter f = new AlwaysSatisfiedFilter();
12          jif.setFilter(f);
13
14          jif.start();
15       } catch (Exception e) {
16          e.printStackTrace();
17       }
18    }
19 }
```

Für dieses unbekannte Programm, ist folgender Output entstanden:

Listing 15: Output der Verschachtelung

```
1 Before staticinitialization(View.<clinit>)
```

```
2 After staticinitialization(View.<clinit>)
3 Before execution(void View.main(String[])) Args: null
4   Before call(View.1())
5     Before staticinitialization(View.1.<clinit>)
6     After staticinitialization(View.1.<clinit>)
7     Before preinitialization(View.1())
8     After preinitialization(View.1())
9     Before initialization(View.1())
10      Before initialization(java.lang.Runnable())
11      After initialization(java.lang.Runnable())
12      Before execution(View.1())
13      After execution(View.1())
14    After initialization(View.1())
15  After call(View.1())
16  Before call(void java.awt.EventQueue.invokeLater(Runnable)) Args:
      View$1@72cbe322
17  After call(void java.awt.EventQueue.invokeLater(Runnable)) Args:
      View$1@72cbe322
18 After execution(void View.main(String[])) Args: null
```

8.3 Breakpoints simulieren

Möchte man durch das unbekannte Programm "stepen", könnte man folgende Implementierung verwenden.

Als erstes braucht man eine Helper-Klasse, die Befehle von System.in entgegen nimmt:

Listing 16: MyBreakPointController

```
1 import java.io.IOException;
2 import java.util.Observable;
3
4 public class MyBreakPointController extends Observable {
5
6    public MyBreakPointController() {
7       super();
8       new Thread(new Runnable() {
9          public void run() {
10            System.out.println("Enter 'q' to stop or 's' to start/stepOver"
                 );
11            char cmd = 0;
12            while (true) {
13               try {
14                  cmd = (char) System.in.read();
15                  switch (cmd) {
16                  case 's':
17                     setChanged();
18                     notifyObservers("EnableOrGoOneStep");
19                     break;
20                  case 'q':
21                     setChanged();
22                     notifyObservers("DisableBreakPoints");
23                     break;
24
25                  default:
26                     break;
27                  }
```

```
28                } catch (IOException e) {
29                    e.printStackTrace();
30                }
31            }
32        }
33    }).start();
34    }
35 }
```

Danach kann die Implementierung des IReceivers in Angriff genommen werden:

Listing 17: MyJaInFraReceiver

```
1  package test;
2
3  import java.util.Observable;
4  import java.util.Observer;
5
6  import javax.swing.JButton;
7
8  import com.jainfra.application.advanced.CatchedObject;
9  import com.jainfra.application.advanced.IReceiver;
10
11 public class MyJaInFraReceiver implements IReceiver, Observer {
12    private MyBreakPointController breakController;
13    private boolean shoudWait;
14    private boolean stepover;
15    private boolean firstTime;
16
17    public MyJaInFraReceiver() {
18       firstTime = true;
19       shoudWait = false;
20       stepover = false;
21       breakController = new MyBreakPointController();
22       breakController.addObserver(this);
23    }
24
25    @Override
26    public void callbackBefore(CatchedObject catchedObject) {
27       System.out.println("Before " + catchedObject);
28       waitNow();
29    }
30
31    @Override
32    public void callbackAfter(CatchedObject catchedObject) {
33       System.out.println("After " + catchedObject);
34       waitNow();
35    }
36
37    private void waitNow() {
38       while (shoudWait) {
39          try {
40             if (stepover) {
41                break;
42             }
43             Thread.sleep(100);
44             if (stepover) {
45                break;
46             }
```

```
47
48          } catch (InterruptedException e) {
49              e.printStackTrace();
50          }
51      }
52      stepover = false;
53  }
54
55  @Override
56  public void update(Observable o, Object arg) {
57      if (((String) arg).equals("DisableBreakPoints")) {
58          System.out.println("Disable BreakPoints");
59          shoudWait = false;
60          firstTime = true;
61      } else if (((String) arg).equals("EnableOrGoOneStep")) {
62          if (firstTime) {
63              System.out.println("Starting BreakPoint");
64              shoudWait = true;
65              firstTime = false;
66              stepover = false;
67          } else {
68              System.out.println("StepOver");
69              shoudWait = true;
70              stepover = true;
71          }
72      }
73  }
74 }
```

Der Code, welcher JaInFra und den MyJaInFraReceiver verwendet, ändert sich nicht vom vorherigen Beispiel (Verschachtelung eines Programmes analysieren).

Folgender Output wurde mit dem Filter:

```
Filter f = new MethodNameFilter("setText");
```

erzeugt:

Listing 18: Output des Breakpoint Programms

```
1  Enter 'q' to stop or 's' to start/stepOver
2  s
3  Starting BreakPoint
4  Before call(void javax.swing.JTextField.setText(String))
5  s
6  StepOver
7  After call(void javax.swing.JTextField.setText(String))
8  s
9  StepOver
10 Before call(void javax.swing.JTextField.setText(String))
11 q
12 Disable BreakPoints
13 After call(void javax.swing.JTextField.setText(String))
14 Before call(void javax.swing.JTextField.setText(String))
15 After call(void javax.swing.JTextField.setText(String))
```

Wie man sieht, kann man durch Eingabe von "s" den Breakpointprozess starten, durch erneutes drücken von "s" eine Methode weiter "stepen" und durch drücken von "q" den Breakpointprozess stoppen.

8.4 Eigener Filter

Sollte dem Kunde die vorgegeben Filterauswahl nicht ausreichen, können mit anonymen inneren Klassen eigene Filter hinzugefügt werden.

Listing 19: Neue Filter definieren

```
1          Filter ownFilter = new Filter() {
2             @Override
3             public boolean isSatisfiedBy(JoinPoint arg0) {
4                return (new Random()).nextBoolean();
5             }
6          };
```

8.5 Verwendung von configureApplication

Diese Methode ist überladen. Möchte man ein JAR File als unbekanntes Programm angeben, macht man dies über die configureApplication Methode mit zwei Parametern:

```
jif.configureApplication(null, new URL("file:///tmp//x.jar"));
```

Wenn dem unbekannten Programm zusätzlich noch Parameter mitgegeben werden sollen, dann verwendet man als ersten Parameter das entsprechende Array:

```
jif.configureApplication(
new String[] {"hallo", "welt"}, new URL("file:///tmp//x.jar"));
```

Besitzt man kein JAR File, sondern nur ein Verzeichnis mit Java Classfiles, wird die configureApplication Methode mit drei Parametern verwendet:

```
jif.configureApplication(
null, new URL("file:///tmp/bin/"), "Foo");
```

Auch hier kann man Parameter für die Main Methode mitgeben:

```
if.configureApplication(
new String[] {"hallo", "welt"}, new URL("file:///tmp/bin/"), "Foo")
```

Literaturverzeichnis

[adv] Erklärung AspectJ Advice. Website. http://www.eclipse.org/aspectj/doc/released/adk15notebook/ataspectj-pcadvice.html; besucht im November 2011.

[jpk] AspectJ JoinPoint Kind. Website. http://www.eclipse.org/aspectj/doc/released/adk15notebook/join-point-signatures.html; besucht im Oktober 2011.

[Lad10] Ramnivas Laddad. *AsepectJ in Action*. Manning, 2010. Seite 62.

JaInFra
Qualitätssicherung & Testauswertung

December 22, 2011

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	1.11.2011	f1egli	Dokument erstellt, FindBugs und Checkstyle eingefügt
1.0rc02	29.11.2011	dmengelt	Beschrieb JUnit Tests und Abdeckung
1.0	01.12.2011	dmengelt	Version 1.0
2.0rc01	18.12.2011	dmengelt	Checkstyle und Logging
2.0rc02	18.12.2011	dmengelt	Usability Tests und Code Review
2.0rc03	19.12.2011	dmengelt	Update JUnit Testing
2.0	20.12.2011	dmengelt	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Review des Codes **3**

4 Testing **3**
4.1 Usability Test 3
4.2 Unit Testing 3
4.2.1 Testabdeckung - EclEmma 5

5 Sonstige Tools & Hilfsmittel **6**
5.1 FindBugs 6
5.2 CheckStyle 6
5.3 Logging 6

6 Informelles zum Code **7**
6.1 Anzahl automatische Builds 7

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

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. Somit wurde ebenfalls gleich ein Code Review durchgeführt. Während des Pair Programmings wurden auch immer gleich Tools wie Checkstyle laufen gelassen. Die angezeigten Fehler wurden dann gemeinsam korrigiert.

4 Testing

4.1 Usability Test

Mit den Usability Tests wollten wir die Inbetriebnahme der JaInFra Library testen. Hierbei stellten wir die Developer Guide und die Library als JAR Datei externen Benutzern zur Verfügung. Selbständig und ohne Hilfe mussten die Benutzer ein unbekanntes Jar-File tracen. Wir konnten so wertvolle Informationen gewinnen. Beispielsweise, dass die statische Analyse mittels Reflection bereits vor dem Start der unbekannte zu tracenden Applikation gemacht werden muss, damit die Filter anschliessend vernünftig gesetzt werden können. Auch zusätzliche Filter wurden aufgrund der Feebacks implementiert.

4.2 Unit Testing

Für das Testen unserer Hauptfunkionalitäten haben wir JUnit verwendet. Der Fokus wurde dabei auf das "Filter" Package gelegt:

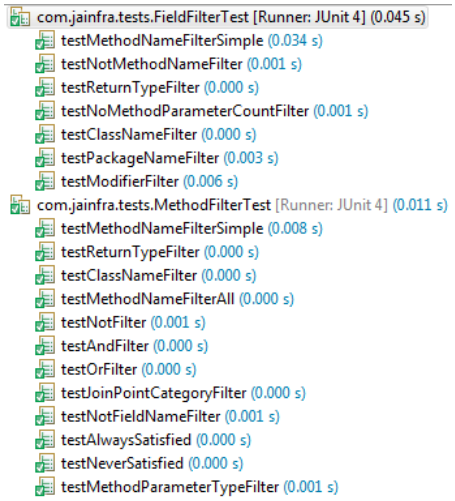

Figure 1: Auswertung JUnit Tests

Alle verfügbaren Filtermöglichketen werden von den beiden Testklassen "FieldFilterTest" und "MethodFilterTest" getestet.

4.2.1 Testabdeckung - EclEmma

Das Filter Package erreicht eine Testabdeckung von über 90%:

Package	Abdeckung
com.jainfra.advanced.filter	86.8%
com.jainfra.advanced.filter.terminal	98.1%
com.jainfra.advanced.filter.nonterminal	100%
Durchschnitt	94.96%

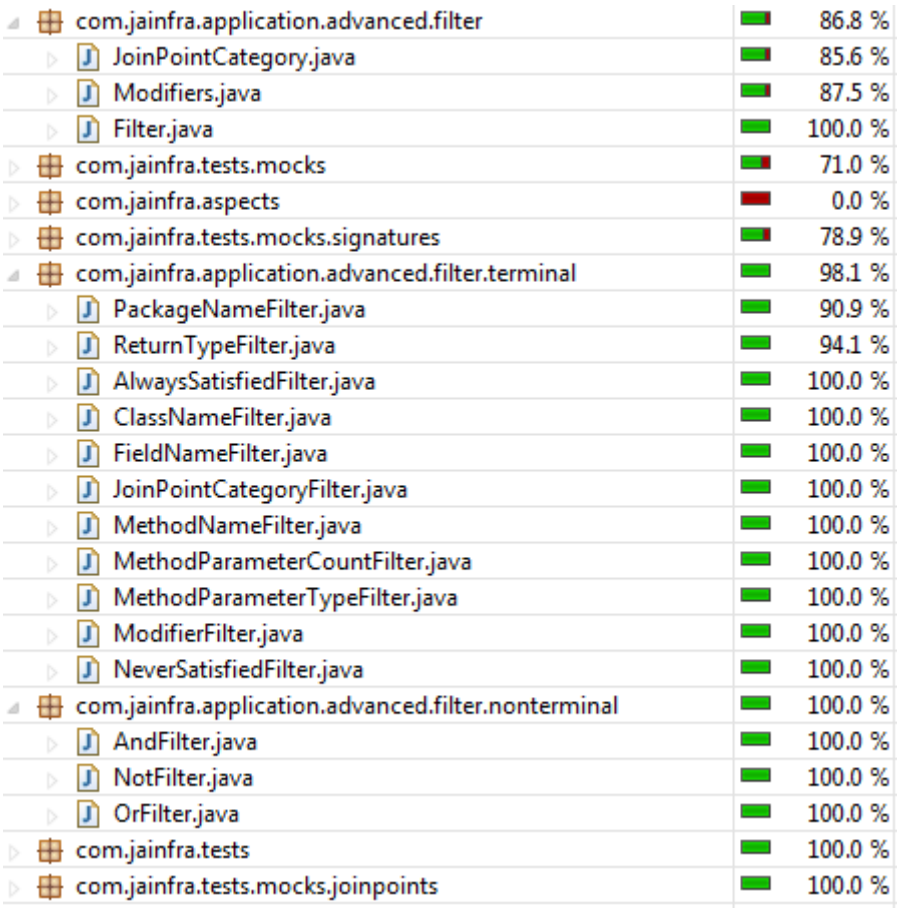

Figure 2: Pfadabdeckung der JUnit Tests im Filter Package

Sämtliche JaInFra Filtertypen "arbeiten" direkt mit einem oder mehreren AspectJ JoinPoints. Damit wir unsere Filter trotzdem testen können, wurden Mock Objekte der JoinPoint Klasse erstellt.

Listing 1: JoinPoint Mock

```
1 public class JoinPointMock implements JoinPoint {
2     // code weggelassen
3 }
```

5 Sonstige Tools & 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. Vor allem Schreiboperationen von Instanzmethoden auf statische Felder konnten reduziert werden.

5.2 CheckStyle

Um den Style des Programmcodes zu verbessern haben wir Checkstyle eingesetzt. Einfach Sachen wie unnötige Leerschläge oder falsche Modifiers konnten so vermieden werden.

Checkstyle violation type	Marker count
Der Parameter X sollte als 'final' deklariert sein.	1
'X' sollte durch eine Konstante definiert sein.	1
Der Bedingungsoperator sollte vermieden werden.	1

Figure 3: Checkstyle Ausgabe

Drei "Violations" konnten wir nicht eliminieren:

- Der Parameter X sollte als "final" deklariert sein.

  Hier konnten wir den final Modifier nicht anwenden. Innerhalb des Programmcodes wird dieser Parameter verändert.

- X sollte durch eine Konstante definiert sein.

  X wurde hier innerhalb der String Methode substring() verwendet. Da X genau nur einmal vorkommt, macht es keinen Sinn extra eine Konstante zu definieren.

- Der Bedingungsoperator sollte vermieden werden.

  Der Programmcode ist an dieser Stelle einfach zu verstehen. Wir haben uns deshalb entschieden den Bedingungsoperator zu belassen.

5.3 Logging

Das Logging der JaInFra Library wird über die Filterklassen gemacht. Folgendes Beispiel zeigt das Logging aller Exceptions in die Datei log.txt:

Listing 2: Logging von Exceptions

```
1 jif.configureLogging("log.txt", new JoinPointCategoryFilter(
    JoinPointCategory.EXECPTION_HANDLER_EXECUTION));
```

Ein möglicher Logging Output sieht folgendermassen aus:

```
Mon Dec 19 21:46:45 CET 2011 BEFORE - method-execution of void com.jainfra.application.JaInFra.start()
Mon Dec 19 21:46:45 CET 2011 BEFORE - method-call of void com.jainfra.application.JaInFra.weaveAspects()
Mon Dec 19 21:46:45 CET 2011 BEFORE - method-execution of void com.jainfra.application.JaInFra.weaveAspects()
Mon Dec 19 21:46:45 CET 2011 BEFORE - field-get of URL com.jainfra.application.JaInFra.source
Mon Dec 19 21:46:45 CET 2011 AFTER - field-get of URL com.jainfra.application.JaInFra.source
```

6 Informelles zum Code

Ein Ziel von uns war es, den Programmcode übersichtlich und knapp zu halten. Dies konnte unter anderem mit dem Einsatz des Interpreter Pattern für das Filtering erreicht werden. Zusätzlich war uns die Gliederung der Packages sehr wichtig. Wenn immer möglich, sollten Aspects (*.aj Sourcefiles) und Klassen (*.java Sourcefiles) getrennt werden. Weitere Informationen können der folgenden Tabelle entnommen werden.

Linien Code insgesamt	1354
Methoden	139
Klassen	38
Packages	13
Interfaces	3

6.1 Anzahl automatische Builds

Nach jedem neuen commit wird die JaInFra Library automatisch erstellt. Während des gesamten Projekts wurde dies über 275 Mal gemacht. Zu Beginn des Projekts gab es einige Konfigurationsschwierigkeiten mit Jenkins und dem AspectJ Compiler ajc. Dies war der Grund, wieso viele Builds fehlgeschlagen sind. Ab Build 100 hatten wir praktisch keine Probleme mehr. Einzig zum Schluss als wir gewisse Filterklassen nochmals umbauen mussten, sind die JUnit Tests nochmals fehlgeschlagen.

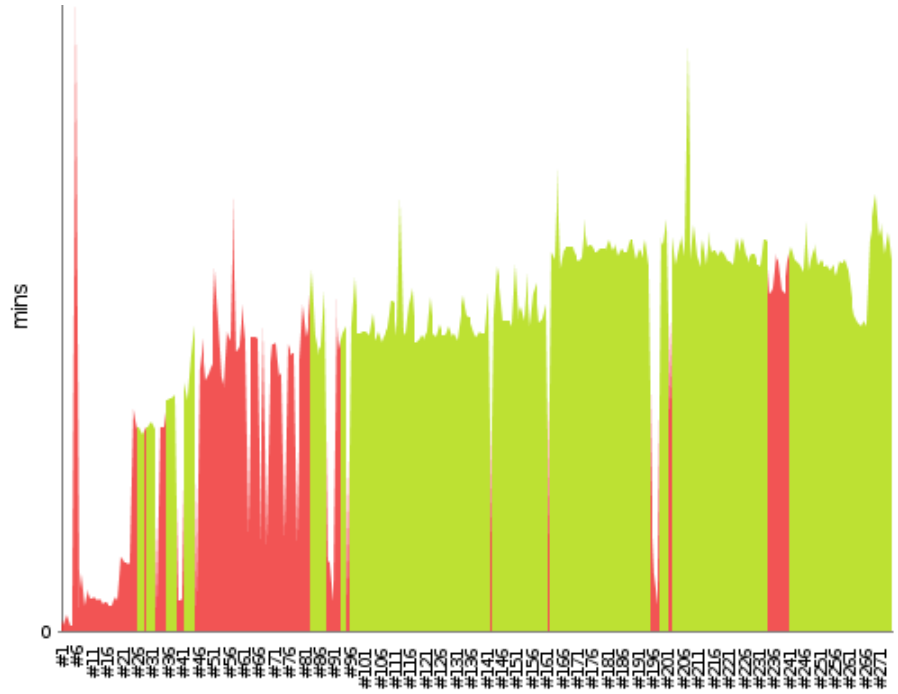

Figure 4: Fehlgeschlagene und normale Builds

JaInFra
Tools & Infrastruktur

December 20, 2011

Verantwortlich: Frederick Egli (f1egli)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	28.11.2011	dmengelt	Ant, Automatische Builds
1.0rc02	06.12.2011	f1egli	Git, Checkstyle, Eclemma, FindBugs
1.0rc03	06.12.2011	dmengelt	Redmine Projektmanagement Software
1.0rc04	09.12.2011	f1egli	aktuelle Versionen hinzugefügt
1.0	12.12.2011	f1egli	Version 1.0 erstellt
2.0rc01	19.12.2011	dmengelt	getSources.sh und sources.lst erklärt
2.0	20.12.2011	f1egli, dmengelt	Finale Version

http://www.jainfra.com

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Infrastruktur und Werkzeuge **3**
3.1 GIT . . . 3
3.1.1 GIT Installation . . . 3
3.1.2 Neues GIT-Repository erstellen . . . 3
3.1.3 Git Repositories Clonen . . . 3
3.1.4 Pull und Push . . . 4
3.1.5 Nützliche Befehle . . . 4
3.2 Jenkins . . . 6
3.3 Redmine . . . 6
3.4 ANT . . . 7
3.4.1 Target init . . . 7
3.4.2 Target compile . . . 7
3.4.3 Target tests . . . 8
3.4.4 Target reports . . . 8
3.4.5 Target publish . . . 8
3.5 Manuelles Kompilieren . . . 8
3.5.1 Angabe der Sourcefiles . . . 8
3.6 Eclipse Plugins . . . 9
3.6.1 Eclemma . . . 9
3.6.2 FindBugs . . . 9
3.6.3 CheckStyle . . . 9

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

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 jainfra.com gibt es das Verzeichnis "/var/gitrepos/". Darin können neue GIT-Repositories mit dem folgenden Befehlen erstellt werden:

Listing 1: Neue GIT-Repo

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

3.1.3 Git Repositories Clonen

Auf dem Server befinden sich zwei GIT-Repositories:

- JaInFra.git
- Scratch.git

Ersteres beinhaltet die Dokumentation (JaInFra/Documentation/) und den Sourcecode (JaInFra/Implementation/) von JaInFra. Die Scratch Repository ist für temporäre Tests und Experimente gedacht.
Um die Repositories lokal zu erstellen verwendet man den Befehl git clone.

```
git clone ssh://<user>@jainfra.com/var/gitrepo/JaInFra.git
git clone ssh://<user>@jainfra.com/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>@jainfra.com/var/gitrepo/JaInFra.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>@jainfra.com/var/gitrepo/JaInFra.git *
[new branch] Testing -> Testing
```

Kollegen, 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>@rapidulu.com/var/gitrepo/JaInFra.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://jainfra.com: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 dann direkt in Redmine vorgenommen werden.

3.4 ANT

Mittels ANT wird nach jedem Push auf das Git Repository ein automatischer Build ausgeführt. Somit kann die Stabilität des Codes gewährleistet werden. Es existieren folgende Targets, welche nacheinander ausgeführt werden:

Listing 2: ANT build.xml Targets

```
<target name="init">
<target name="compile">
<target name="tests">
<target name="reports">
<target name="publish">
```

3.4.1 Target init

Erstellt die benötigten Verzeichnisse in welche die JaInFra Library kopiert wird. Pro Build wird ein neues Verzeichnis mit dem aktuellen Zeitstempel erstellt. Zu einem späteren Zeitpunkt kann somit auf eine ältere Version der Library zurückgegriffen werden.

3.4.2 Target compile

Kompiliert die Sourcefiles mittels ajc. Zusätzlich werden die Dateien des AspectJWeavers in die Library kopiert:

Listing 3: ANT Target compile

```
<iajc argfiles="sources.lst" outjar="/var/www/downloads/library/${current.
    timestamp}/JaInFra.jar"
    source="1.6" target="1.6" outxml="true" outxmlfile="META-INF/aop-ajc.
        xml">
    <classpath>
        <pathelement location="./lib/aspectjweaver.jar"/>
        <pathelement location="./lib/aspectjrt.jar"/>
        <pathelement location="./lib/junit.jar"/>
    </classpath>
</iajc>

<jar destfile="/var/www/downloads/library/${current.timestamp}/JaInFra.jar"
      update="true">
    <zipfileset includes="org/**" src="./lib/aspectjweaver.jar" />
</jar>
```

Mittels "outjar" wird die JaInFra Library in das unter "init" erstellte Verzeichnis gelegt.

3.4.3 Target tests

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

Listing 4: Ausführung der Tests

```
<batchtest fork="yes" todir="/var/www/downloads/library/${current.timestamp
    }/tests">
  <zipfileset src="/var/www/downloads/library/${current.timestamp}/JaInFra
      .jar">
    <include name="**/jainfra/tests/*Test*.class"/>
  </zipfileset>
</batchtest>
```

3.4.4 Target reports

Kreiert einen Report der ausgeführten Tests. Für die Testauswertungen siehe:

02_QM/QS-Testauswertung_VX.X.pdf

3.4.5 Target publish

Publiziert einen neue Version der JaInFra Library in das downloads Verzeichnis:

http://downloads.jainfra.com/JaInFra.jar

3.5 Manuelles Kompilieren

Listing 5: Kompilieren mittels ajc

```
ajc -source 6 -classpath
"./lib/aspectjrt.jar;./lib/aspectjweaver.jar;./lib/junit.jar"
-outxml -outjar /tmp/JaInFra.jar -argfile sources.lst
```

3.5.1 Angabe der Sourcefiles

Sowohl beim automatisierten als auch beim manuellen Kompilieren, werden die Sourcefiles vorgängig mit einem Shell-Skript zusammengestellt und anschliessend dem AspectJ Compiler als Argument (sources.lst) übergeben:

Listing 6: Alle .java und .aj Files finden

```
#!/bin/bash
find . -type f -name "*.java" -o -name "*.aj" > sources.lst
```

Dies resultiert in einer Liste mit allen *.java und *.aj Files:

```
[...]
./src/com/jainfra/application/advanced/IInjection.java
./src/com/jainfra/application/advanced/IReceiver.java
./src/com/jainfra/application/advanced/JaInFraAdvanced.java
./src/com/jainfra/application/basic/IReceiver.java
./src/com/jainfra/application/basic/JaInFraBasic.java
[...]
```

3.6 Eclipse Plugins

3.6.1 Eclemma

Die neuste Version von Eclemma kann im Eclipse unter Help->Install New Software.. installiert werden. Die "Work with" URL lautet: `http://update.eclemma.org/`

Zur Entwicklungszeit wurde die Version 1.5.3 verwendet. Diese befindet sich unter : `Software/EclipsePlugins/Eclemma/eclemma-1.5.3.zip`

Bei offenen Fragen loht es sich, die Hersteller Webseite zu kontaktieren: `http://www.eclemma.org/`

3.6.2 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 2.0.0.20111130 verwendet. Diese befindet sich unter : `Software/EclipsePlugins/FindBugs/edu.umd.cs.findbugs.plugin.eclipse_2.0.0.20111130.zip`

Bei offenen Fragen lohnt es sich, die Hersteller Webseite zu kontaktieren: `http://findbugs.sourceforge.net/`

3.6.3 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 JaInFra 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/net.sf.eclipsecs-updatesite_5.5.0.201111092104-bin.zip`

Bei offenen Fragen lohnt es sich, die Hersteller Webseite zu kontaktieren: `http://checkstyle.sourceforge.net/`

JaInFra
Zeitauswertung

December 22, 2011

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	15.12.2011	dmengelt	Auswertungsgrafiken
1.0rc02	15.12.2011	dmengelt	Durchschnitt pro Projektmitglied
1.0	22.12.2011	f1egli, dmengelt	Finale Version

http://www.jainfra.com

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

3 Auswertung Arbeitswochen **3**

4 Auswertung pro Kategorie **4**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

3 Auswertung Arbeitswochen

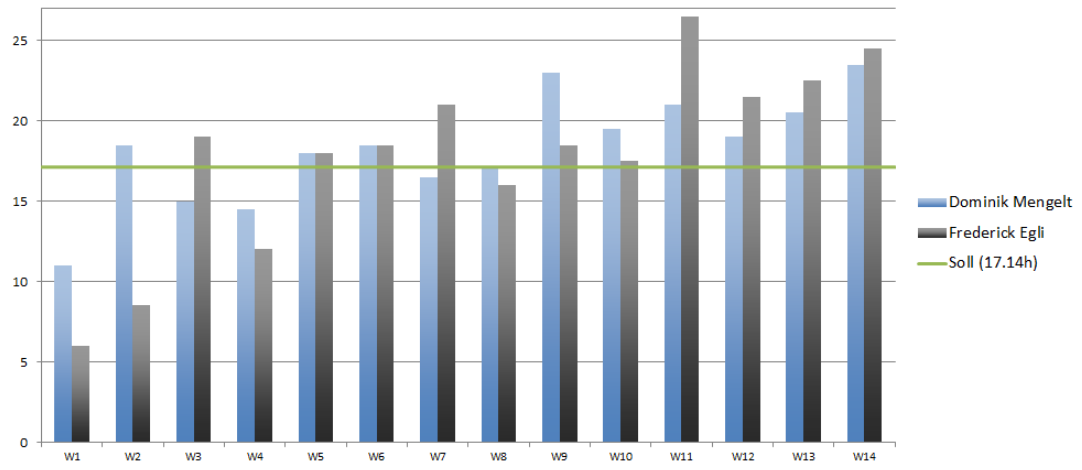

Figure 1: Anzahl Stunden pro Semesterwoche

Die aufgewendeten Stunden pro Projektmitglied. Der Schnitt von 17.14 Stunden wurde recht gut erreicht:

Mitglied	Aufgewendete Stunden pro Woche im Durchschnitt
Frederick Egli	17.9
Dominik Mengelt	17.6

4 Auswertung pro Kategorie

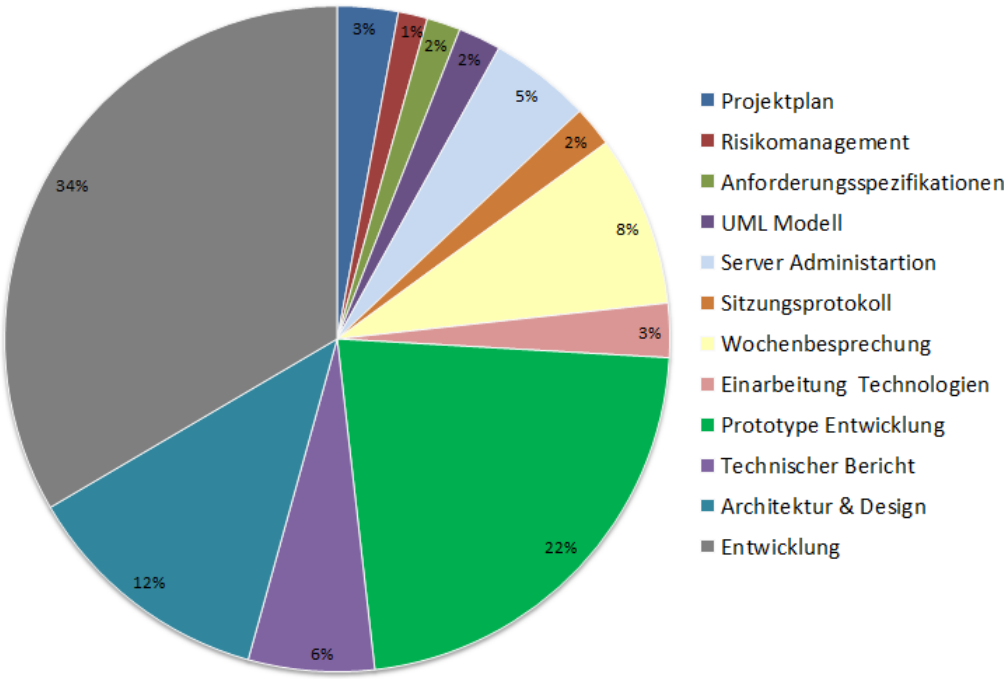

Figure 2: Verteilung der Stunden pro Kategorie

Total wurden 505 Stunden für das Projekt aufgewendet. Neben den grossen Teilen für den Prototype und die Entwicklung, fiel auch ein grosser Teil für die Architektur und das Design an.

JaInFra
Glossar

December 20, 2011

Verantwortlich: Dominik Mengelt (dmengelt)

1 Änderungsgeschichte

Version	Datum	Name	Bemerkung
1.0rc01	28.11.2011	dmengelt	Erste Version
1.0	01.12.2011	dmengelt	Version 1.0
2.0rc01	12.12.2011	dmengelt	Anpassungen und neue Begriffe
2.0rc02	13.12.2011	dmengelt, f1egli	Korrekturen
2.0	20.12.2011	dmengelt, f1egli	Finale Version

Inhaltsverzeichnis

1 Änderungsgeschichte **1**

2 Ziel und Zweck **3**

Glossar **3**

2 Ziel und Zweck

Siehe "Artefaktenliste" im Projektplan.pdf

Glossar

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.
ajc	AspectJ Java Compiler - Wird verwendet um die JaInFra Library zu kompilieren.
ANT	Werkzeug zum automatisierten Erzeugen von ausführbaren Programmen aus dem Quellcode.
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.
Before	Ist ein Advice Typ. Es wird vor der eigentlichen Operation getraced.
Endanwender	Verwendet die vom Kunden erstellte Applikation, welche JaInFra im Hintergrund einsetzt, um eine unbekannte Applikation zu tracen.
Filter	Durch das Einsetzen von Filtern kann die Ausgabe beim tracing einer Applikation verfeinert/eingegrenzt werden..
JaInFra	Java Inspection Framework. Name der Library.
Jenkins	Jenkins ist ein erweiterbares, webbasiertes System zur kontinuierlichen Integration in agilen Softwareprojekten.
JoinPoint	Ein JoinPoint ist der Teil, welcher gerade im Kontrollfluss des Programmes ist.
Kunde	Bezeichnet den direkten Kunden, welcher die JaInFra Library benutzt. Entweder erstellt der Kunde ein GUI, damit Endanwender eine unbekannte Applikation tracen können, oder der Kunde erstell eine CLI Applikation, damit er als Endanwender ein unbekanntes Programm tracen kann..
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. Bei JaInFra werden diese Bedingungen mittels Filter festgelegt.
rc	Release Candidate.
Reflection	Ermöglicht es, zur Laufzeit auf Klassen und Methoden zuzugreifen, deren Existenz oder genaue Ausprägung zur Zeit der Programmerstellung nicht bekannt war.
tracing	Die Hauptfunktionalität von JaInFra. Tracing bedeutet Loggen bzw. Identifizieren von beispielsweise Methodenaufrufen oder Lesezugriffe auf Felder..
Unknown.jar	Eine zu tracende unbekannte Applikation.

weaving Ein AspectJ Ausdruck. Die in der JaInFra Library definierten Aspects werden mit der zu tracenden Applikation verwoben.