Extension Contracts

– ein pragmatischer Ansatz zur formalen Beschreibung der

Erbenschnittstelle von Rahmenwerken

Diplomarbeit

vorgelegt von:
Rolf Tyzuk
Langenhorner Chaussee 15
22335 Hamburg Matrikel-Nr.: 4725330
eMail: rolf@tyzuk.de

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Hamburg, Februar 2004

Erstbetreuer: Prof. Dr. Heinz Züllighoven
Zweitbetreuer: Prof. Dr. Michael Kölling

Danksagung

Ich möchte an dieser Stelle allen Menschen danken, die mir bei der Erstellung dieser Arbeit geholfen haben.

Ich danke

- Prof. Dr. Heinz Züllighoven für die Erstbetreuung dieser Arbeit
- Prof. Dr. Michael Kölling für die Zweitbetreuung dieser Arbeit
- Axel Schmolitzky für anregende Gespräche und Ideen
- meinen Kollegen, die Korrektur gelesen und mich von einiger Arbeit entlastet haben
- meinen Freunden und Verwandten, dass sie mir immer wieder Mut zugesprochen und mich angespornt haben

Schließlich gilt mein ganz besonderer Dank meiner Schwester, die mich immer unterstützt hat und mir zu Hause die Arbeit an der vorliegenden Diplomarbeit so weit wie möglich erleichtert hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Im Java-Umfeld gibt es inzwischen unzählige Klassenbibliotheken und Rahmenwerke für unterschiedlichste Anwendungen und in unterschiedlichsten Größen. Viele Klassenbibliotheken oder kleine Rahmenwerke unterstützen Entwickler bei kleinen Aufgaben, zum Beispiel bei der Verarbeitung von XML-Dateien. Größere Rahmenwerke bestimmen teilweise den Aufbau ganzer Anwendungsprogramme. Am Arbeitsbereich Softwaretechnik der Universität Hamburg wird zum Beispiel seit einigen Jahren das JWAM-Rahmenwerk (vgl. [JWAM03], [Züllighoven98]) entwickelt. Es handelt sich dabei um ein in Java geschriebenes Rahmenwerk, dass die Entwicklung von Anwendungen, die dem sogenannten WAM-Ansatz (Werkzeug, Automat & Material) entsprechen, unterstützt. Das JWAM-Rahmenwerk enthält viele Basisklassen, zu denen vor allem für die Entwicklung von Werkzeugen und Materialien Unterklassen entwickelt werden. Im täglichen Umgang wird das Rahmenwerk dabei als White-Box-Rahmenwerk angesehen, es wird also vorausgesetzt, dass Entwickler, die dieses Rahmenwerk benutzen, sich auch mit dem inneren Aufbau auseinandersetzen.

Das Rahmenwerk wird in unterschiedlichen universitären und professionellen Softwareprojekten eingesetzt. Deshalb ist es nötig, den Entwicklern, die mit JWAM arbeiten, eine möglichst genaue und umfassende Dokumentation des Rahmenwerkes zur Verfügung zu stellen. Für die Einarbeitung in das JWAM-Rahmenwerk und als Dokumentation dienen kleine Beispielprogramme und mit Javadoc erzeugte Schnittstellenbeschreibungen. Damit ist die Dokumentation von JWAM vergleichbar mit der anderer Rahmenwerke und Funktionsbibliotheken wie zum Beispiel JavaHelp, Struts und vielen weiteren, die zur Zeit im Java-Umfeld entstehen.

1.1 Vertragsmodell

Für die Schnittstellenbeschreibungen von JWAM wurden zusätzliche Javadoc-Tags definiert, mit denen mit Einschränkungen Vor- und Nachbedingungen nach dem Vertragsmodell von Meyer (Design by contract, vgl.[Mey97]) dokumentiert werden können. Diese Vor- und Nachbedingungen können mit Hilfe der im Rahmenwerk enthaltenen Klasse „Contract“ zur Laufzeit geprüft werden. Durch die konsequente Benutzung der Javadoc-Tags geben die Schnittstellenbeschreibungen in vielen Fällen sehr genau an, mit welchen Werten bestimmte Methoden einer Klasse aufgerufen werden dürfen und welchen Rückgabewert diese liefern. So ist zum Beispiel bei Indizes mit einem Blick auf die Vorbedingungen ersichtlich, ob die Indizes bei 0 oder bei 1 beginnen. Die

Dokumentation beschreibt bisher also mittels der Klassen- und Methodenschnittstellen und des Vertragsmodells von Meyer vor allem, wie Klassen und Methoden benutzt werden.

1.2 Das Problem der Erbenschnittstelle

Deutlich schlechter, wenn überhaupt, ist beschrieben, wie von Klassen aus dem Rahmenwerk geerbt werden kann. Anhand der Schnittstelle ist bei Klassen zum Beispiel ersichtlich, welche Methoden von Unterklassen implementiert werden müssen („abstract“ Methoden) und welche Methoden nicht redefiniert werden dürfen („final“ Methoden). Die Unterscheidung in „public“ und „protected“ Methoden auf der einen Seite und „private“ Methoden auf der anderen Seite erlaubt auch die Unterscheidung zwischen Methoden, die von einer Unterklasse prinzipiell aufgerufen bzw. nicht aufgerufen werden dürfen. Aus der Schnittstellenbeschreibung ist also ersichtlich, welche Methoden implementiert bzw. redefiniert werden dürfen. Nicht ersichtlich ist hingegen, wie eine bestimmte Methode in einer Unterklasse implementiert werden muß.

Sun versucht dieses Problem zu lösen, indem in Klassen des *Java Development Kit* JDK (vgl. [J2SDK1.3] bzw. [J2SDK1.4]) in Prosa-Text und mit Hilfe mathematischer Formeln beschrieben wird, wie sich einzelne Methoden verhalten sollen. Beispiele sind die Methoden „equals“, „hashCode“ und „clone“ der Klasse „Object“, bei denen sowohl das beabsichtigte Verhalten der jeweiligen Methode, als auch die Beziehungen der Methoden zueinander beschrieben sind. Aber ähnlich wie beim Vertragsmodell wird nur definiert, welches erlaubte Eingabewerte und welches mögliche Ergebniswerte sind.

Nicht beschrieben wird allerdings, ob die Methodenimplementierung gewissen Einschränkungen in Bezug auf die anderen Methoden einer Klasse unterliegt. Es stellt sich zum Beispiel die Frage, ob die Implementierung der Oberklasse aufgerufen werden muß, oder ob die Implementierung der Unterklasse die der Oberklasse vollständig ersetzt. Weiterhin wird nicht deutlich, welche Methoden der Oberklasse aufgerufen werden können, ohne in potentielle Fehlersituationen, wie zum Beispiel in [Blo01] und [RuLe00] beschrieben, zu geraten. Die Klasse *HashMap* aus der Java-Standardbibliothek erwartet z.B., dass zwei gleiche Objekte (d.h. die Methode *equals* liefert für diese Objekte den Wert *true*) auch den gleichen Hashcode haben. Wenn nun eine Klasse die Methode *equals* überschreibt, ohne gleichzeitig die Methode *hashCode* zu überschreiben, kann es passieren, dass die Methode *hashCode* für zwei nach Methode *equals* gleiche Objekte unterschiedliche Werte liefert. Dies kann zu undefiniertem Verhalten führen, wenn diese Objekte in *HashMap*s abgelegt werden. Es ist auch möglich, durch falsches Überschreiben einer Methode Endlosschleifen im Programmablauf zu erzeugen.

Aus dem Beispiel wird deutlich, dass es der Sprache Java an einer Möglichkeit fehlt, die Erbenschnittstelle formal so exakt definieren zu können, dass deren Einhaltung automatisch überprüfbar ist.

1.3 Bestehende Lösungsansätze

Es existieren schon einige theoretische Ansätze, wie die Vererbungsschnittstelle einer Klasse mit sogenannten „Reuse Contracts“ beschrieben werden kann. Reuse Contracts werden in einer formalen Sprache notiert, so dass deren Einhaltung durch entsprechende Programme überprüft werden kann.

Allerdings liefert keiner der bisherigen Ansätze praxistaugliche Werkzeuge, mit deren Hilfe Reuse Contracts für Java-Programme mit zumutbarem Aufwand umgesetzt werden können. So berücksichtigt zum Beispiel der Ansatz von Ruby und Leavens (vgl. [RuLe00]) auch die Migration zwischen unterschiedlichen Rahmenwerksversionen, so dass die resultierende Beschreibungssprache JML (Java Modeling Language) unhandlich ist, und die entsprechenden Werkzeuge bisher nur ansatzweise implementiert wurden.

1.4 Ein pragmatischer Ansatz

In der vorliegenden Arbeit wird deshalb ein pragmatischer Ansatz vorgestellt, wie Reuse Contracts in Java-Programmen eingesetzt werden können, indem sich dieser Ansatz auf die Vererbung zwischen bekannten Klassen fokussiert. Zur Unterscheidung zu den bisherigen Ansätzen wird dieser Ansatz deshalb als „Extension Contract“ bezeichnet. Die Erbenschnittstelle einer Klasse betrifft potentiell zwei unterschiedliche Entwicklergruppen, nämlich einerseits die Entwickler einer Oberklasse, die in der Lage sein sollten, die Erbenschnittstelle möglichst formal beschreiben zu können und andererseits die Entwickler von Unterklassen. Sie profitieren von einer formaleren Beschreibung der Erbenschnittstelle, da mehr Fehler automatisch erkannt werden können. Zur Unterscheidung dieser beiden Gruppen werden im Folgenden Entwickler von Klassen, von denen geerbt werden kann, unter dem Begriff *Rahmenwerksentwickler* zusammengefasst. Im Unterschied dazu werden Entwickler, die Unterklassen zu vorhandenen Klassen programmieren, als *Anwendungsentwickler* bezeichnet.

Der Ansatz der *Extension Contract* beruht darauf, dass der Rahmenwerksentwickler mit Hilfe spezieller Javadoc-Tags (*Extension Contract Tags*) einen Vertrag für die Erbenschnittstellen seiner Klassen genau genug definieren kann, so dass der Anwendungsentwickler automatisch überprüfen kann, ob seine Unterklassen diesen Vertrag einhalten. Sowohl für den Rahmenwerks- als auch für den Anwendungsentwickler werden dafür Werkzeuge zur Verfügung gestellt, um die Erzeugung die Überprüfung der *Extension Contracts* zu automatisieren. Dadurch wird zusätzlich zur Fehlererkennung einerseits ein großer Zeitvorteil für die Entwickler erreicht, andererseits wird verhindert, dass der Anwendungsentwickler den *Extension Contract* einer Klasse übersieht.

Für die Weitergabe der *Extension Contracts* ist eine eigene Datei vorgesehen. Dies hat mehrere Vorteile. In kommerziellen Projekten ist es möglich – wenn dieses gewünscht ist –, die *Extension Contracts* ohne den Quelltext auszuliefern. Weiterhin bietet sich für

Anwendungsentwickler die Möglichkeit, die *Extension Contracts* nachträglich noch zu ändern. Dies kann zum Beispiel nützlich sein, wenn bei der Verwendung eines gegebenen Rahmenwerkes häufig gleiche Fehler gemacht werden, oder wenn in einem Projekt strengere Regeln als normalerweise üblich angelegt werden. Zum Beispiel ist es denkbar, dass in einem Projekt in jeder Klasse, die direkt von *Object* erbt, die Methoden *equals*, *hashcode*, *clone* und *toString* überschrieben werden sollen. Dies kann mit einer selbstgeschriebenen *Extension Contract*-Datei geprüft werden.

1.5 Vorgehen

Es werden zunächst einige typische Fehler aufgezeigt, die bei der Programmierung von Unterklassen auftreten können. Die Fehler werden daraufhin untersucht, wie diese erkannt bzw. umgangen werden können. Hieraus sollen eine Reihe von Javadoc-Tags (Extension Tags) hervorgehen, mit denen die Vererbungsschnittstelle von Javaklassen formal genauer als bislang möglich beschrieben werden kann. Damit diese Extension-Tags für Programmierer einen Nutzen haben und ihm nicht zusätzliche Arbeit aufladen, werden in dieser Arbeit auch entsprechende Werkzeuge erarbeitet, um die Einhaltung der Extension-Contracts automatisch überprüfen zu können.

So wird zum Beispiel ein Werkzeug entwickelt, das aus den Java Sourcecodes, in denen Extension Tags verwendet werden, die formale Beschreibungen der Vererbungsschnittstellen (Extension Contracts) erzeugt. Somit können Rahmenwerke zusammen mit den Extension Contracts ausgeliefert werden, ohne dass der Sourcecode offengelegt werden muss. Damit die Extension Contracts auch im Nachhinein ohne den Sourcecode verstanden und im Extremfall geändert werden können, werden sie in einem vom Entwickler lesbaren und leicht veränderbaren XML-Format gespeichert. Für Entwickler, die die Rahmenwerke benutzen, wird ein Werkzeug entwickelt, mit dem sie überprüfen können, ob ihre eigenen Klassen die Extension Contracts der Rahmenwerksklassen erfüllen.

Folgende Fragestellungen sollen auf dem Weg zu den Extension-Tags und den entsprechenden Werkzeugen unter anderem behandelt werden:

- Welche möglichen Fehler können mit Hilfe von Extension Contracts erkannt bzw. umgangen werden?
- Wie lassen sich Extension Contracts in Java umsetzen?
- Wie hoch ist der Aufwand für Entwickler von Basis- und Subklassen bei der Verwendung von Extension Contracts?
- Gibt es Extension Contracts, die von Softwarewerkzeugen automatisch erzeugt werden können?

1.6 Aufbau der Arbeit

Zunächst werden in Kapitel 2 anhand einiger realer Beispiele aus dem JWAM-Rahmenwerk und dem JDK mögliche Fehler bei der Programmierung von Unterklassen betrachtet.

In Kapitel 3 werden daraufhin mehrere Ansätze, die Vererbungsschnittstelle genauer zu beschreiben bzw. zu überprüfen, untersucht und verglichen. Dabei wird vor allem betrachtet, welche möglichen Fehler durch die vorgestellten Ansätze jeweils erkannt werden und welcher Vorteil und Mehraufwand sich durch ihren Einsatz für den Programmierer ergibt. Weiterhin wird aufgezeigt, an welchen Stellen die vorgestellten Ansätze nicht in der Lage sind, die in Kapitel 2 vorgestellten Fehler zu verhindern.

Auf diesen Überlegungen aufbauend wird dann in Kapitel 4 der neue Ansatz der *Extension Contract*s vorgestellt, mit dem die häufigsten Fehler erkannt werden können und der für die Programmierer mit entsprechenden praxistauglichen Werkzeugen einsetzbar ist. Dieser Ansatz wird anschließend mit den zuvor betrachteten Ansätzen verglichen.

In Kapitel 5 werden einige ausgewählte Implementationsdetails der Extension Contracts vorgestellt. Dabei werden vor allem die generischen Teile, an denen dieser Ansatz noch erweitert werden kann, genauer betrachtet.

Der Abschluß der vorliegenden Arbeit bietet eine kurze Zusammenfassung und einen Ausblick, inwieweit sich dieser Ansatz von Extension Contracts erweitern läßt. Dies könnte z. B. durch zusätzliche Javadoc-Tags oder neue Softwarewerkzeuge geschehen.

2 Reale Beispiele für mögliche Fehlerquellen

Im Folgenden werden reale Beispiele vorgestellt, die einige Fehlerquellen bei der Benutzung von Vererbung aufzeigen. Dabei wird davon ausgegangen, dass die Klassen, von denen geerbt werden soll, als Blackbox benutzt werden, also vor allem der Quelltext dieser Klassen für den Anwendungsentwickler nicht zugänglich ist. Rahmenwerke und Klassenbibliotheken werden zwar häufig als Whitebox Rahmenwerke vorausgesetzt, doch ist dies nicht allgemein gültig, wie die Klassen des JDK zeigen. Die nachfolgenden Beispiele werden allerdings verdeutlichen, dass der Anwendungsentwickler auch dann nicht vor den vorgestellten Fehlerquellen geschützt ist, wenn er Zugriff auf den Quelltext hat. Dies ist darin begründet, dass er sich teilweise in die Gedankenwelt des Rahmenwerksentwicklers versetzen muss. Aus den Beispielen werden allgemeine Fehlerquellen abgeleitet, so dass am Ende dieses Kapitels ein Forderungskatalog aufgestellt wird, nach dem die in dieser Arbeit vorgestellten Ansätze beurteilt werden können.

2.1 Von einander abhängige Methoden

Häufig kommt es in Klassen vor, dass Methoden von anderen Methoden abhängen, indem z. B. Methoden andere aufrufen oder mehrere Methoden auf die gleiche Membervariable zugreifen und diese verändern. Diese Abhängigkeit kann sowohl nur in einer Richtung vorhanden sein als auch in beide Richtungen gelten. Ein Beispiel für eine Abhängigkeit in einer Richtung findet sich in der Klasse *Object* des Packages *java.lang*. Die Methode *hashCode* wird in der Klasse *Object* definiert. Dies bedeutet, dass jede Klasse diese Methode von *Object* erbt. Wenn die Methode *hashCode* in einer Klasse überschrieben wird, muss dies folglich im Sinne der Klasse *Object* geschehen. Sun hat die Eigenschaften der Methode *hashCode* im Javadoc-Kommentar der Klasse *Object* festgelegt und schreibt unter anderem vor (vgl. [J2SDK1.3]):

> If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

Auch wenn dieser Kommentar nicht formal, sondern nur als Prosatext geschrieben ist, wird damit vergleichbar dem Vertragsmodell von Meyer (vgl. [Mey97]) das Verhalten

der Methode *hashCode* festgelegt. Objekte, die diese Methode aufrufen, verlassen sich auf diese Eigenschaft. Zum Beispiel funktionieren einige Containerklassen nur wie erwartet, wenn die Methode *hashCode* für gleiche Objekte auch den gleichen Wert liefert.

Ab der JDK Version 1.4 (vgl. [J2SDK1.4]) steht zusätzlich im Kommentar der Methode *equals* ein Hinweis auf die Methode *hashcode*:

> Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

Betrachtet man im Vergleich die Dokumentation zur JDK Version 1.3 ([J2SDK1.3]), so fällt auf, dass der zitierte Absatz zur Methode *equals* noch nicht vorhanden war. Dies lässt darauf schließen, dass der Vertrag der Methode *hashCode* von Programmierern häufig nicht beachtet wurde, wenn die Methode *equals* überschrieben wurde. Daher hat Sun den Zusammenhang der Methoden *equals* und *hashCode* in der Dokumentation ab der Version 1.4 noch einmal explizit hervorgehoben.

Dieses Beispiel zeigt einerseits, dass es wichtig ist, Abhängigkeiten zwischen Methoden beschreiben zu können. Andererseits wird deutlich, dass es im allgemeinen nicht reicht, dies als Fließtext im Javadoc-Kommentar einer Klasse oder Methode zu tun, da auf diese Weise eine automatische Überprüfung nicht möglich ist.

2.2 Zusammengehörige Methoden

Im Folgenden wird nun ein Bespiel für gegenseitig voneinander abhängigen Methoden betrachtet. Die Klasse *DomainvalueImpl* ist Bestandteil des JWAM-Rahmenwerkes ([JWAM03]). Sie erlaubt die Definition von Fachwertklassen, deren Instanzen sich wie Werte verhalten sollen. Dazu werden eigene Klassen als Unterklassen von *DomainvalueImpl* definiert.

In der Klasse *DomainvalueImpl* sind mehrere Methoden definiert, die voneinander abhängen. Sie sind als Einschubmethoden vorgesehen, sind also leer implementiert. Es handelt sich um die Methoden *canCreateFromString()*, *valueImpl(String)* und *isValid(String)*. Da diese Methoden voneinander abhängen, müssen sie in einer Unterklasse alle überschrieben werden, sobald eine von ihnen überschrieben wird. Dieses steht auch im Javadoc-Kommentar zu diesen Methoden, zum Beispiel lautet der Kommentar zur Methode *valueImpl(String)*:

> Notice: You have to redefine canCreateFromString(), valueImpl(String) and isValid(String) always together to ensure consistency.

Da die Methode *valueImpl(String)* erst im Laufe der Rahmenwerksweiterentwicklung aus der Methode *value(String)* hervorgegangen ist, haben sich allerdings Inkonsistenzen bei den Kommentaren ergeben. So wird zum Beispiel im Kommentar der Methode *canCreateFromString* auf die Methode *value(String)* statt *valueImpl(String)* verwiesen:

> Notice: You have to redefine canCreateFromString(), value(String) and isValid(String) always together to ensure consistency.

Dieses Beispiel zeigt, dass ein Anwendungsentwickler, dessen Klassen von vorhandenen Rahmenwerksklassen erben sollen, nicht nur den Kommentar der Rahmenwerksklasse sehr sorgfältig lesen muss, sondern diesen teilweise auch korrigierend interpretieren muss. Hier muss der Entwickler für sich die Frage beantworten, ob er nun die Methode *value(String)* oder *valueImpl(String)* überschreiben soll. Zusätzlich wird deutlich, dass Rahmenwerksentwickler bisher keine Unterstützung erhalten, um Kommentare konsistent zueinander zu halten, wenn Methoden im Laufe der Zeit verändert werden.

2.3 Methoden erweitern vs. überschreiben

Die Klasse *ToolFunctionalityImpl* ist ebenfalls Bestandteil der JWAM Rahmenwerkes. Sie ist die Basisklasse für Funktionskomponenten von Werkzeugen, die nach dem IAK/FK Muster (vgl. [Züllighoven98], S. 530*ff*) entworfen werden. Der Entwickler einer speziellen Funktionskomponente stößt beim Lesen des Javadoc Kommentars dieser Klasse unter anderem auf folgende Hinweise:

```
/**
 * This method has to be redefined by subclasses if they want to create
 * sub-functionalities.
 */
protected void equip();

/**
 * Use the given material in the tool. This operation has to be
 * redefined to react in a specific way to the use of a new material
 * in the tool.
 */
protected void doUseMaterial(Thing material);
```

Diese beiden Methoden sind also dafür vorgesehen, dass sie in Unterklassen redefiniert werden. Aber erst, wenn man in den Quelltext sieht, oder einen Entwickler des Rahmenwerkes fragt, kann man dieses sinnvoll tun. Das liegt daran, dass „redefined“ ein zu allgemeiner Begriff ist und bei den beiden Methoden jeweils unterschiedliches meint.

redefinieren Eine Methode wird in einer Unterklasse *redefiniert*, wenn diese Methode (Methodenname und Parametertypen) sowohl in der Oberklasse als auch in der Unterklasse definiert wird.

Nachfolgend wird der Quelltext der beiden genanten Methoden betrachtet:

```
private boolean _isEquipped = false;

protected void doUseMaterial (Thing material)
{
  // this operation has to be redefined to act in a specific way
}

protected void equip ()
{
  _isEquipped = true;
}
```

Der Quelltext der beiden Methoden zeigt, dass die eine Methode bisher leer implementiert ist, wohingegen die andere Methode schon Quelltext enthält. Bei letzterer muss nun ein Anwendungsentwickler entscheiden, ob die Implementierung der Oberklasse aufgerufen wird, wenn er diese Methode überschreibt. Die Tatsache, dass dort die als *final* deklarierte Membervariable *_isEquipped* auf einen neuen Wert (*true*) gesetzt wird, deutet darauf hin, dass die gegebene Implementierung auf jeden Fall ausgeführt werden muss, da von einer Unterklasse aus kein Zugriff auf diese Variable besteht.

Wird der Quelltext der Klasse weiter betrachtet, wird deutlich, dass es auch darauf ankommt, wann die Implementierung der Oberklasse ausgeführt wird. Da für die folgenden Aktionen vorausgesetzt wird, dass *_isEquipped* auf *true* gesetzt ist, muss die Implementierung der Oberklasse vor dem Code der Unterklasse ausgeführt werden.

Die Methode *doUseMaterial* hingegen kann auf beliebige Weise redefiniert werden. Da sie in der Oberklasse leer definiert ist, können Unterklassen diese Methode überschreiben oder erweitern, wobei es unerheblich ist, wann die Implementierung der Oberklasse aufgerufen wird. Sinnvoller Weise wird man in einem solchen Fall diese Methode überschreiben.

Diese Beispiele haben gezeigt, dass der Begriff *redefinieren* oftmals zu unspezifisch ist. Die entscheidende Frage ist, ob beim Redefinieren einer Methode die Implementierung der Oberklasse aufgerufen werden muss oder nicht. Nach Lamping und und Kieczales (vgl. [KiLa92], S.444) werden für diese beiden Varianten im Folgenden die Begriffe *überschreiben* (engl.: to override) und *erweitern* (engl.: to extend) benutzt:

erweitern Eine Methode wird in einer Unterklasse *erweitert*, wenn sie *redefiniert* wird und die Implementierung der Oberklasse von der Implementierung der Unterklasse aufgerufen wird.

überschreiben Eine Methode wird in einer Unterklasse *überschrieben*, wenn sie *redefiniert* wird und die Implementierung der Oberklasse von der Implementierung der Unterklasse **nicht** aufgerufen wird

2.4 Methodenaufrufe aus Konstruktoren

Die Java-Sprache erlaubt es, im Gegensatz zu z.B. C++, aus Konstruktoren heraus Methoden polymorph aufzurufen. Dies stellt eine große Fehlergefahr dar, da so Methoden eines Objektes einer Unterklasse aufgerufen werden können, bevor das Objekt komplett initialisiert wurde.

```
public class Zahl
{
  public Zahl(int x)
  {
    _wert = berechneWert(x);
  }

  public int berechneWert(int x) {
    return x;
  }

  public int getWert()
  {
    return _wert;
  }

  private int _wert = 0;
}

public class ZahlMal2 extends Zahl
{
  public static void main(String[] args)
  {
    ZahlMal2 zahl = new ZahlMal2(15);
    System.out.println(zahl.getWert());
  }

  public ZahlMal2(int x)
  {
    super(x);
  }

  public int berechneWert(int x)
  {
    return x * _faktor;
  }

  private int _faktor = 2;
}
```

Das obige Beispiel zeigt zwei Klassen *Zahl* und *ZahlMal2*. Die Klassen demonstrieren einen mißglückten Versuch, zu den übergebenen Zahlen einen bestimmten Wert zu berechnen und zu speichern. Aus dem Konstruktor der Klasse *Zahl* wird die Methode *berechneWert* polymorph aufgerufen. Dies kann, wie an diesem Beispiel zu sehen ist, zu einem unerwünschten Verhalten führen, wenn in einer Unterklasse die Methode *berechneWert* überschrieben und dabei auf Membervariablen zugegriffen wird. In diesem Fall liefert die Methode *berechneWert* der Unterklasse beim Programmstart nämlich nicht den Wert *30* (15 * 2) sondern den Wert *0* zurück. Dies liegt daran, dass die Methode aufgerufen wird, bevor der Konstruktor der Unterklasse aufgerufen wird, da zuerst der Konstruktor der Oberklasse ausgeführt wird und dieser direkt die Methode *berechneWert* aufruft. Solange aber der Konstruktor der Unterklasse nicht ausgeführt wird, sind auch die Membervariablen noch nicht initialisiert und so hat die Variable *_faktor* noch den Wert *0*, wenn die Methode *berechneWert* darauf zugreift. In diesem Beispiel könnte das Problem in der Oberklasse dadurch gelöst werden, dass die Methode *berechneWert* erst aus der Methode *getWert* aufgerufen wird. In der Unterklasse könnte das Problem umgangen werden, indem die Variable *_faktor* als Klassenvariable definiert wird.

Die Klasse *JTable* ist ein Beispiel einer Swing-Klasse, bei der trotzdem aus dem Konstruktor heraus bestimmte Methoden polymorph aufgerufen werden. Wenn eine dieser Methoden überschrieben wird, muss also darauf geachtet werden, dass nicht auf Membervariablen zugegriffen wird. Die beiden Beispiele zeigen, dass es für einen Anwendungsentwickler von entscheidender Bedeutung ist, zu wissen ob und welche Methoden einer Klasse polymorph aus einem Konstruktor der Oberklasse heraus aufgerufen werden. Hat er diese Informationen nicht, kann es passieren, dass auf Membervariablen zugegriffen wird, bevor diese initialisiert sind, was wiederum zu falschen, schwer nachvollziehbaren Resultaten führt.

2.5 Zusammenfassung

Die vorgestellten Beispiele zeigen Fehlerquellen, die größtenteils durch mangelhafte Kommentare bzw. deren fehlende automatische Überprüfbarkeit verursacht werden. Das Beispiel der Methodenaufrufe aus Konstruktoren heraus zeigt, dass die Eigenschaften von Java im Vergleich zu anderen Sprachen zu weiteren Fehlerquellen führen.

Die ersten drei Beispiele zeigen, dass gewisse Informationen über die Implementierung der Oberklasse bekannt sein müssen, um eine der Intention nach korrekte Unterklasse zu schreiben. An dieser Stelle kann also das Geheimnisprinzip(vgl. [Par72]) (Information hiding) nicht vollständig aufrecht gehalten werden. Alan Snyder stellt dazu fest (vgl. [Sny86]):

> Inheritance breaks Encapsulation

Damit ist gemeint, dass eine Klasse, von der andere Klassen erben können, nicht als eine in sich geschlossene und von außen unveränderbare Einheit angesehen werden darf.

Einerseits müssen, wie die Beispiele gezeigt haben, gewisse Implementierungsdetails bekannt sein, um eine semantisch korrekte Unterklasse schreiben zu können. Andererseits ist es – versehentlich oder mutwillig – möglich, in Unterklassen Methoden der Oberklasse derart zu überschreiben, dass sich andere Methoden der Oberklasse nicht mehr wie erwartet verhalten. Dies schreibt auch Szyperski: (vgl.: [Szy98], S.109)

> However, the general claim still holds: a subclass can interfere with the implementation of its superclasses in a way that breaks the superclasses. Likewise, an evolutionary change of a superclass can break some of its existing subclasses.

Szyperskis zweiter Gedanke ist, dass sich verändernde Klassen, zum Beispiel Rahmenwerksklassen, dazu führen, dass deren Unterklassen an die neuen Versionen angepasst werden müssen. Dieser Aspekt wird bei der Untersuchung der bisherigen Ansätze zur Beschreibung der Vererbungsschnittstelle wieder aufgegriffen werden.

Die Beispiele der Methode *hashCode* und der Klasse *DomainvalueImpl* zeigen, dass viele Fehler vermieden bzw. frühzeitig erkannt werden könnten, wenn formal definiert werden kann, welche Methoden zusammen beziehungsweise in Abhängigkeit von anderen Methoden überschrieben werden müssen.

Das Beispiel der Klasse *ToolFunctionalityImpl* zeigt, dass der Entwickler von Anwendungsprogrammen präzisere Informationen dazu benötigt, wie er einzelne Methoden redefinieren muss. Die wesentliche Frage ist zunächst, ob er die Methode überschreiben oder erweitern soll. Wenn eine Methode erweitert werden soll, schließt sich als nächste Frage an, an welcher Stelle die Implementierung der Oberklasse aufgerufen werden soll.

Durch Einschubmethoden kann ein Entwickler von Unterklassen von diesen Fragen befreit werden. Allerdings müssten dann in jeder Ebene der Vererbungshierarchie entsprechende Einschubmethoden neu definiert werden. Für Standardfälle, in denen die Implementierung der Oberklasse vor oder nach der Implementierung der Unterklasse ausgeführt wird, erscheint die Einführung von Einschubmethoden sehr umständlich.

Am letzten Beispiel ist zu erkennen, dass es für den Entwickler von Unterklassen von entscheidender Bedeutung ist, zu wissen, welche Methoden polymorph aus dem Konstruktor der Oberklasse heraus aufgerufen werden.

Um einen Anwendungsentwickler vor den vorgestellten Fehlerquellen warnen, und ihn gegebenenfalls auf einen Fehler hinweisen zu können, sind also folgende Anforderungen zu erfüllen:

- In der Erbenschnittstelle muss so definiert werden können, ob eine Methode überschrieben oder erweitert werden soll, dass dieses automatisch überprüfbar ist.
- Die Beachtung von Abhängigkeiten von Methoden in Unterklassen muss automatisch überprüfbar sein.

- Polymorphe Methodenaufrufe aus Konstruktoren heraus müssen in der Vererbungsschnittstelle definiert werden können.
- Idealerweise kann die Erbenschnittstelle auch noch nachträglich unabhängig vom Quelltext beschrieben werden, wodurch sich z. B. die Vererbungsschnittstelle der Klasse *Object* spezifizieren ließe.
- Es muss ein Werkzeug geben, mit dem sich überprüfen lässt, ob sich der Anwendungsentwickler an die Spezifikation der Vererbungsschnittstellen gehalten hat.

3 Bestehende Lösungsansätze

In diesem Kapitel werden unterschiedliche Ansätze, die sich mit der Spezifikation der Vererbungsschnittstelle auseinandersetzen, untersucht. Dafür werden zunächst Ansätze, die im akademischen Umfeld entstanden sind, in ihrer zeitlichen Reihenfolge vorgestellt. Dabei wird deutlich, dass die späteren Ansätze Ideen der vorangegangenen aufgegriffen und fortgeführt haben. Im Anschluss daran werden pragmatische Ansätze aus dem *Open Source*-Bereich vorgestellt. Alle Ansätze werden unter anderem daraufhin untersucht, inwieweit sie die im vorangegangenen Kapitel aufgezeigten Fehlerquellen beseitigen können. Den Abschluss dieses Kapitels bildet ein Vergleich der vorgestellten Lösungsansätze.

3.1 Lampings „specialization interface“

John Lamping et al. haben in mehreren Artikeln das Problem besprochen, wie die Erbenschnittstelle (*specialization interface*) von Klassen mit Hilfe von kleinen Spracherweiterungen genauer beschrieben werden kann, um leichter korrekte Unterklassen schreiben zu können. Sie betrachten dabei die Programmiersprachen *Common Lisp Object System* (*CLOS*, vgl. [DeGa87] und [Kee89]) und *Eiffel*(vgl. [Mey92]).

3.1.1 Konzept

John Lamping grenzt das *specialization interface* strikt von der Benutzungsschnittstelle ab. Der Unterschied besteht darin, dass die Benutzungsschnittstelle die Beziehung zwischen Objekten, z. B. Methodenaufrufe beschreibt und den inneren Aufbau der Objekte dafür außer acht läßt. Das *specialization interface* hingegen beschreibt was beim Umgang mit Klassen zu beachten ist. Lamping schreibt dazu (vgl. [Lam93], S.202):

> The *client interface* is used to access the functionality of objects. The Interface is accessed by sending messages to objects, and the internal structure of the objects shouldn't be evident. The *specialization interface*, on the other hand is used to extend and modify classes. That is, it operates on classes, not objects.

Seiner Meinung nach bildet eines der Kernkonzepte Objektorientierter Programmierung das Hauptproblem. Beim Beerben von Rahmenwerksklassen kann die Implementation einer Unterklasse von der Implementation der Oberklasse aufgerufen werden

und sich dadurch wie ein Teil des Rahmenwerkes verhalten. Wie auch schon in Kapitel 2.5 gezeigt, müssen also zusätzliche Informationen bereitgestellt werden, um korrekte Unterklassen bilden zu können. Dazu schreiben Kiczales und Lamping (vgl. [KiLa92], S. 438):

> The problem then is how to say enough about the internal workings of the library that the user can write replacement modules, without saying so much that the implementor has no room to work.

Bei den Informationen, die ein Anwendungsentwickler benötigt, dessen Klassen von vorhandenen Klassen erben sollen, unterscheiden beide Autoren die zwei folgenden Kategorien: einerseits den Klassengraphen und die Vererbungsbeziehung zwischen den Klassen, andererseits, und das ist für diese Arbeit von besonderer Bedeutung, das Protokoll der Erbenschnittstelle. Lamping definiert Protokoll wie folgt(vgl. [Lam93], S.204):

> A protocol is a description of what is available at an interface. A protocol need not be completely formal, but it must be complete enough to tell how to use the interface correctly.

Das Protokoll zur Erbenschnittstelle muss beschreiben, welche Methoden an der Schnittstelle verfügbar sind, und wie sie bei der Unterklassenbildung zu redefinieren sind. Etwas präziser für die Erbenschnittstelle formulieren Kiczales und Lamping(vgl. [KiLa92], S. 438):

> A Second Part of the problem has to do with protocol. The user needs to know, when they subclass *button*[1], what methods they must define, what behavior those methods can rely on and what those methods must and must not do.

Unter dem Protokoll wird also mehr als nur die *public*- und *protected*- Schnittstelle einer Klasse verstanden. Zusätzlich fallen darunter im wesentlichen die Abhängigkeiten zwischen Methoden. Dies kann z. B. der Aufruf anderer Methoden sein oder auch die Abhängigkeit mehrerer Methoden von der selben Membervariablen (vgl. [KiLa92], S.442*ff*). Falls es solche Abhängigkeiten gibt, muss ein Anwendungsentwickler dafür Sorge tragen, dass die betroffenen Methoden alle entsprechend redefiniert werden, sobald nur eine einzige von ihnen in einer Unterklasse redefiniert wird. Lamping schlägt am Beispiel von Eiffel vor (vgl. [Lam93] S.204*ff*), die Programmiersprache so zu erweitern, dass die benötigten Informationen, nämlich die Abhängigkeiten zwischen Methoden einer Klasse, dargestellt werden können.

[1] button ist eine Beispielklasse für ein Grafikobjekt aus dem zitierten Artikel

Ein weiterer Punkt, der das Protokoll betrifft, sind Methoden, die nicht überschrieben werden sollen, da ihre Implementierung Grundlage für das Funktionieren der Klasse ist. Das Überschreiben von Methoden kann in vielen Fällen mit Mitteln der verwendeten Programmiersprache verhindert werden, z. B. *final*-Methoden in Java. Dieses verhindert allerdings, dass ein Anwendungsentwickler einer Unterklasse die Implementierung der betreffenden Methode sinnvoll erweitern kann. Als Ausweg können in Java Einschubmethoden definiert werden, wobei der Entwickler der Rahmenwerksklasse dabei vorhersehen können muss, an welchen Stellen dies notwendig werden kann. CLOS bietet durch sein *Metaobject Protocol* (vgl. [Kee89], S. 219*ff*) darüber hinaus die Möglichkeit, zu definieren, dass vor und nach einer Methodenimplementierung andere Implementierungen ausgeführt werden müssen (vgl. [KiLa92] und [Kee89] S.113f).

3.1.2 Beispiel

Am Beispiel einer einfachen Klasse *set*, die einen Behälter für Objekte darstellt, soll gezeigt werden, wie mit dem beschriebenen Ansatz einem Anwendungsentwickler wichtige Informationen mitgegeben werden können, um leichter korrekte Unterklassen schreiben zu können.

Zunächst soll eine mögliche Implementierung der Klasse *set* in der Programmiersprache *Eiffel* ausschnittsweise betrachtet werden (vgl. [Lam93] S. 203 Figure 1:Part of the class *set*):

```
class set
feature
  add (new_object: any) is do ... end;
  add_all (new_objects: set) is do ... end;
  remove (old_object: any) is do ... end;
  remove_if_present (old_object: any): boolen is do ...
    end;
  for_each (action action) is do ... end;
end
```

Der Quelltext enthält keine Information, was bei der Implemtierung einer Unterklasse zu der gezeigten Klasse beachtet werden muss. Es wird nur deutlich, welche Methoden vorhanden sind, und ohne zusätzliche Informationen kann selbst der Sinn der Methoden nur mit Hilfe des Namens hergeleitet werden. Die Methoden *add* und *add_all* dienen zum Einfügen eines Objektes bzw. eines *set*s, welches beliebig viele Objekte enthalten kann. Die Methoden *remove* und *remove_if_present* dienen dazu, ein Objekt aus dem *set* zu entfernen. Mit der Methode *for_each* lässt sich eine Aktion (*action*) auf jedes Objekt dieses *set*s anwenden.

Ein Anwendungsentwickler erhält keinerlei Informationen darüber, ob er in einer Unterklasse der Klasse *set* bestimmte Methoden unabhängig von anderen redefinieren kann, bzw. welche Methoden zusammen redefiniert werden müssen.

Lamping schlägt nun vor, die Methoden und Attribute auf unterschiedliche Klassen aufzuteilen, um dann mit Hilfe einer Spracherweiterung zu Eiffel die Abhängigkeiten zwischen den Methoden darstellen zu können (vgl. [Lam93] S. 205 Figure 2: Typing the methods' dependencies):

```
class set_core
feature
  add (new_object: any) is deferred end;
  remove_if_present (old_object: any): boolen is deferred
    end;
  for_each (action action) is deferred end;
end

class set
feature
  add_all (new_objects: set) is deferred end;
  remove (old_object: any) is deferred end;
end

class set_rep
feature
  hidden_representation:
end

class default_set
inherit set;set_rep
feature
  add [set_rep] (new_object: any) is do ... end;
  add_all [set_core] (new_objects: set) is do ... end;
  remove [set_core] (old_object: any) is do ... end;
  remove_if_present [set_rep] (old_object: any): boolen is
    do ... end;
  for_each [set_rep] (action action) is do ... end;
end
```

In dieser zweiten Version sind die Methoden und Attribute der ursprünglichen Klasse in mehrere Klassen aufgeteilt. Zusätzlich ist bei den Methoden der Klasse *default_set* in eckigen Klammern angegeben, von welcher Klasse sie abhängen. Die Methode *add* hängt z. B. nur von der Klasse *set_rep* ab, das bedeutet, dass sie nur auf Methoden und Attribute der angegebenen Klasse zugreift, in diesem Fall also nur von der angedeuteten internen Darstellung *hidden_representation* der ursprünglichen Klasse. Dies bedeutet, dass die Methode *add* nicht geändert werden braucht, so lange sich die Klasse *set_rep* und deren Methoden in ihrem Verhalten nicht ändern. Das gleiche gilt auch für die Methoden *remove_if_present* und *for_each*, die in der selben Klasse wie die Methode *add* definiert sind und ebenfalls nur von der Klasse *set_rep* abhängen. Die letztgenannten Methoden können in diesem Beispiel auch unabhängig voneinander geändert werden, da sie nicht voneinander abhängen. Wäre z. B. die Methode *for_each* von der Methode *add* abhängig, müßte dieses in der Klassenbeschreibung noch angegeben werden. Dies könnte wie folgt aussehen:

```
for_each [set_rep, set_core] (action action) is
    do ... end;
```

Das Beispiel zeigt, wie mit Hilfe einer solchen Spracherweiterung beschrieben werden kann, welche Methoden redefiniert werden können, ohne dass andere Methoden entsprechend angepasst werden müssen. Weiterhin kann beschrieben werden, dass bestimmte Methoden von anderen abhängen und deshalb gegebenenfalls redefiniert werden müssen.

3.1.3 Bewertung

Der Ansatz von Lamping liefert wichtige Grundlagen für die Beseitigung der beschriebenen Fehlerquellen. So wird einerseits streng zwischen der Vererbungsschnittstelle und der Benutzungsschnittstelle unterschieden. Außerdem liefert Lamping Ansätze, was in der Vererbungsschnittstelle beschrieben werden muss. Doch der Ansatz greift nur die Abhängigkeiten zwischen Methoden auf. Eine Unterscheidung, ob eine Methode in einer Unterklasse überschrieben oder erweitert werden soll, wird nicht vorgenommen. Da sich Lamping in diesem Fall auf CLOS und Eiffel bezogen hat, kann der Ansatz auch nicht die in Java möglichen polymorphen Methodenaufrufe aus Konstruktoren heraus erkennen. Weiterhin läßt sich Lampings Ansatz nicht eins zu eins auf Java übertragen, da es in Java keine Mehrfachvererbung gibt.

3.2 Reuse Contracts

Der Ansatz der *Reuse Contracts* wurde von Patrick Steyaert, Kim Mens und anderen im Jahr 1996 vorgestellt (vgl. [SLMH96]). Das Hauptaugenmerk der *Reuse Contracts* liegt auf den Konsequenzen, die die Evolution von Rahmenwerken mit sich ziehen. Mit der Hilfe von *Reuse Contracts* soll ein Anwendungsentwickler erkennen, ob und welche Klassen geändert werden müssen, wenn eine neue Version eines benutzten Rahmenwerkes eingesetzt werden soll.

3.2.1 Konzept

Durch *Reuse Contracts* werden die Abhängigkeiten der Methoden untereinander beschrieben. Dadurch ist es einem Anwendungsentwickler möglich zu erkennen, auf welche Methoden einer Unterklasse er sein Hauptaugenmerk legen muss, wenn er eine neue Version einer Oberklasse einsetzen möchte. Ein *Reuse Contract* bezieht sich auf eine Klasse und enthält alle für die Vererbungsschnittstelle relevanten Methoden. In einer streng getypten Sprache wie Java können die *Reuse Contracts* automatisch erzeugt werden. Welche Methoden relevant sind, entscheidet dabei der Rahmenwerksentwickler, denn nur er weiß, wo es zu Problemen beim Programmieren von Unterklasse zu seinen Klassen kommen kann(vgl. [SLMH96], S.273):

> In that case, reuse contracts could be semi-automatically constructed on the basis of the calling structure (the programmer only has to delete the descriptions and names of methods that should not be exposed).

Für jede Methode wird der Name der Methode angegeben, welche anderen Methoden dieser Klasse diese Methode aufruft, und ob die Methode abstrakt oder konkret ist. Steyaert et al definieren *Reuse Contract* entsprechend:

> A *reuse contract* is an interface, i.e., a set of method descriptions each consisting of
>
> - a unique name,
> - an annotation abstract or concrete
> - a (possible empty) specialization clause.

Nach dieser Definition muss für jede Methode ein eindeutiger Name angegeben werden. Es muß angegeben werden, ob diese Methode in der entsprechenden Rahmenwerksklasse abstrakt oder konkret ist. Als letztes wird in der sogenannten *specialization clause* angegeben, welche relevanten anderen Methoden aufgerufen werden.

Da in der vorliegenden Arbeit das Hauptaugenmerk auf der Programmiersprache Java liegt, haben die ersten beiden Punkte eine geringere Bedeutung als die *specialization clause*. Zum einen erlaubt Java für mehrere Methoden den gleichen Namen, vorausgesetzt, sie haben unterschiedliche Parameterlisten. Zum anderen erkennt der Java-Compiler, ob Methoden in einer Klasse abstrakt oder konkret definiert sind, so dass dieses nicht noch in einer zusätzlichen Schnittstellenbeschreibung angegeben werden muss.

Wenn sich eine Methode einer Rahmenwerksklasse von einer Rahmenwerksversion zu einer anderen ändert, kann es sein, dass diese Methode bestimmte Methoden nicht mehr aufruft oder dass sie neue Methoden aufruft. Dies bedeutet gleichzeitig, das sich der *Reuse Contract* dieser Rahmenwerksklasse entsprechend ändert. Für den Anwendungsentwickler hat dies zur Folge, dass er die Methoden, die in einem *Reuse Contract* neu hinzugekommen oder entfallen sind, betrachten und evtl. ändern muss. Das gleiche gilt für Methoden, die in einer *specialization clause* einer anderen Methode dieses *Reuse Contracts* neu hinzugekommen oder entfallen sind.

Wenn der Anwendungsentwickler nur mit einer Version einer Klasse in Kontakt kommt, helfen ihm die *Reuse Contracts* nur noch wenig. Der Anwendungsentwickler kann aber immerhin anhand der *specialization clause* der einzelnen Methoden erkennen, welche Methoden voneinander abhängen. Bei diesen Methoden muß der Anwendungsentwickler dann besonders aufpassen, ob die übrigen Methoden sich noch wie gewünscht verhalten, wenn er eine dieser Methoden überschreibt.

Steyaert et al haben die möglichen Änderungen, die sich zwischen zwei Versionen eines *Reuse Contracts* ergeben können, untersucht und auf elementare Änderungen, sogenannte *Reuse Contract Operatoren* zurückgeführt. Zwei *Reuse Contracts* lassen sich also

entweder durch einen *Reuse Contract Operator* oder durch eine Kombination mehrerer *Reuse Contract Operatoren* ineinander überführen.

Die für die vorliegende Arbeit wichtigsten *Reuse Contract Operatoren* sind *Refinement* und *coarsening*, die angeben, ob die betreffende Methode neue Methoden aufruft bzw. bestimmte Methoden nicht mehr aufruft:

Refinement *Refinement* eines *Reuse Contracts* bedeutet, dass die *specialization clause* einer Methode um neue Methoden erweitert wird.

Coarsening *Coarsening* eines *Reuse Contracts* ist das Gegenteil von *Refinement* und bedeutet somit, dass Methoden aus der *specialization clause* einer Methode entfernt werden.

3.2.2 Beispiel

Als Beispiel kann eine Rahmenwerksklasse *Set* dienen, die unter anderem die Methoden *add* und *addAll* besitzt, wobei die Methode *addAll* zum Einfügen der einzelnen Elemente die Methode *add* benutzt. Wenn nun ein Anwendungsentwickler eine Unterklasse *CountableSet* schreibt, die beim Einfügen von neuen Elementen die Zahl der Elemente mitzählt, so reicht es die Methode *add* zu erweitern.

Die Abbildung 3.1 zeigt die beiden Klassen *Set* und *CountableSet* und gibt an, dass der *Reuse Contract* der Klasse *CountableSet* mit Hilfe des *Reuse Contract Operators refinement* aus dem *Reuse Contract* Klasse *Set* hervorgeht. Der Grund hierfür liegt darin, dass die Methode *add* der Klasse *CountableSet* die Methode *count* aufruft. Weiterhin ist in der Abbildung die Klasse *OptimizedSet*, eine neue Version der Klasse *Set*, zu sehen. In der Klasse *OptimizedSet* wurde die Methode *addAll* effizienter implementiert, so dass sie nicht mehr die Methode *add* aufruft. Die *Reuse Contracts* betreffend entspricht das einem *coarsening*, da die Methode *addAll* nicht mehr die Methode *add* aufruft.

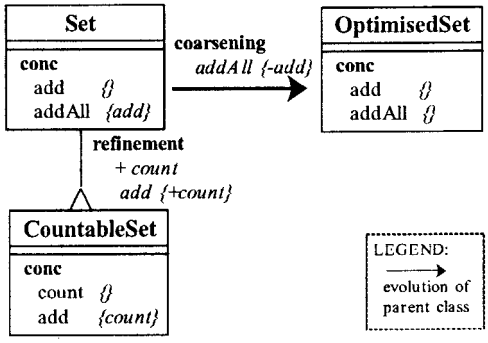

Abbildung 3.1: Inconsistent Methods (vgl. [LSM97])

Stellt man sich nun vor, dass die Klasse *OptimizedSet* die Klasse *Set* ersetzt und deren Namen übernimmt, so weiß der Anwendungsentwickler der Klasse *CountableSet* anhand der *Reuse Contract Operators addAll{-add}*, dass er nun auch die Methode *addAll* redefinieren muss, da in der Unterklasse bereits die angegebene Methode *add* redefiniert wird.

3.2.3 Bewertung

Wie gezeigt, behandeln *Reuse Contracts* vor allem Probleme beim Einsatz unterschiedlicher Versionen bestimmter Klassen. Von den vorgestellten Fehlerquellen kann nur eine einzige mit Hilfe von *Reuse Contracts* umgangen werden, wobei die ursprüngliche Bedeutung der *specilization clause* verbogen wird. Ein Rahmenwerksentwickler kann von einander abhängige Methoden beschreiben, indem er in die *specialization clause* der abhängigen Methode die andere Methode aufnimmt. Dieses ist nicht im ursprünglichen Sinn der *specialization clause*, da dort nur Methoden aufgeführt werden sollen, die auch aufgerufen werden. Die übrigen Fehlerquellen lassen sich mit *Reuse Contracts* nicht umgehen. Damit sind *Reuse Contracts* insgesamt nicht geeignet um die vorgestellten Fehlerquellen zu umgehen.

3.3 Java Modeling Language (JML)

An der Universität von Iowa ist unter der Leitung von Gary Leavens die Java Modeling Language (JML) entstanden. JML ist eine Metasprache, mit der das Verhalten von Java-Programmen beschrieben werden kann. JML wurde ursprünglich entwickelt, um die Korrektheit von Java-Programmen verifizieren zu können. Erst im Laufe der Zeit wurden Elemente hinzugefügt, mit denen die Vererbungsschnittstelle näher beschrieben werden kann.

3.3.1 Konzept

JML-Spezifikationen werden entweder im Javadoc-Kommentar einer Methode oder in einer eigenständigen Datei definiert. Mit Hilfe eines speziellen Compilers wird aus der JML-Spezifikation Java-Quelltext erzeugt, der zur Laufzeit des Programms die definierten Bedingungen überprüft. Der *JML*-Compiler ist selbst in Java geschrieben und kann deshalb im Prinzip wie jedes andere Java Programm auch installiert und gestartet werden.

In JML können Vor- und Nachbedingungen entsprechend des Vertragsmodells von Meyer (vgl. [Mey97]) zu jeder Methode definiert werden. Dazu bietet JML die Schlüsselworte *requires* und *ensures*. Darüber hinaus bietet JML auch die Möglichkeiten, eigene Variablen zu definieren und auf die Membervariablen vor und nach dem Aufruf der

Methode zuzugreifen. Mit Hilfe von mathematischen Operatoren lassen sich Invarianten und genaue Bedingungen definieren.

Relevanter für die vorliegende Arbeit ist, dass mit JML definiert werden kann, welche Membervariablen eine Methode verändert und welche anderen Methoden diese aufruft. Dazu existieren die Schlüsselworte *assignable* und *callable*. Das Schlüsselwort *assignable* gibt an, dass die entsprechende Methode die aufgeführten Membervariablen verändern darf. Dabei bedeutet *assignable \nothing*, dass die entsprechende Methode keine Membervariablen verändern darf.

Das Schlüsselwort *callable* wird in der aktuellen JML-Version noch nicht unterstützt. Es soll in einer späteren Version zu jeder gewünschten Methode automatisch erzeugt werden und alle Methoden auflisten, die die gegebene Methode aufruft. Dadurch soll es möglich sein, mit JML das Konzept der *Reuse Contracts* nachzubilden, das Schlüsselwort *callable* übernimmt dabei die Bedeutung der *specialization clause*.

Weiterhin soll JML den Programmierer dabei unterstützen, zu vorhandenen Klassen korrekte Unterklassen zu schreiben. Dazu schreibt Leavens (vgl. [RuLe00], S. 208):

> A correct superclass method can only be problematic when a new subclass overrides some methods, leading to downcalls. A downcall occurs when a superclass method calls a method that is overridden in the subclass.

Als Konsequenz sieht Leavens, dass zu jeder Methode einer vorhandenen Klasse geprüft wird, welche anderen Methoden diese aufruft. Für diesen Zweck ist das Schlüsselwort *callable* vorgesehen. Dabei bezieht er Konstruktoren mit ein vgl. [RuLe00], S. 211:

> The *callable* clause lists the signatures of methods (and constructors) directly called by the method being specified. Because constructor calls can also cause downcall problems, we include constructors when we use the term "method".

In einer Erweiterungsstufe sollen mit dem JML Compiler diese Abhängigkeiten zwischen Methoden erkannt werden. Aus den ermittelten Informationen sollen, angelehnt an den Ansatz der Reuse Contracts, Bedingungen erzeugt werden, die angeben, welche Methoden mit welchen anderen überschrieben werden müssen. Während der Entstehung der vorliegenden Arbeit war diese Funktionalität noch nicht implementiert.

3.3.2 Beispiel

Am Beispiel der Methode *iSqrt*, die den ganzzahligen Teil der Wurzel aus einer gegebenen natürlichen Zahl berechnet, sollen im Folgenden die wesentlichen Eigenschaften von JML erläutert werden. Dabei wird die JML-Spezifikation als Javadoc-Kommentar vor der Methode geschrieben:

```
public class IntMathOps {
  /*@ public normal_behavior
    @ requires y >= 0;
    @ assignable \nothing;
    @ ensures -y <= \result && \result <= y
    @ && \result * \result <= y
    @ && y < (Math.abs(\result) + 1) *
    @        (Math.abs(\result) + 1);
    @*/
  public static int iSqrt(int y) {
    return (int) Math.sqrt(y);
  }
}
```

Die Methode soll die kleinste ganze Zahl, die kleiner gleich der Quadratwurzel der gegebenen Zahl *y* ist, berechnen. Dazu wird im Javadoc-Kommentar als erstes definiert, dass diese Methode nicht mit einer Exception (*normal_behavior*) beendet wird, wenn der Parameter *y* den Vorbedingungen genügt. Weiterhin kann man ersehen, dass sehr präzise Vor- und Nachbedingungen definiert wurden (*requires / ensures*), z. B. darf für *y* nur eine (als *int* darstellbare) natürliche Zahl übergeben werden. Das Schlüsselwort *assignable* unterscheidet sich von den anderen Schlüsselwörtern, da es Informationen über die interne Funktionsweise dieser Methode bekannt gibt. Mit Hilfe dieses Schlüsselwortes wird definiert, auf welche Membervariablen diese Methode zugreifen darf. Im genannten Beispiel darf die Methode also auf keine Membervariable zugreifen.

Das vorgestellte Beispiel wird vom JML-Compiler in einen Java-Quelltext mit 25 Methoden und 383 Zeilen übersetzt. Der erzeugte zusätzliche Java-Quelltext sorgt zur Laufzeit dafür, dass die beschriebenen Bedingungen eingehalten werden. Dieses kleine Beispiel macht deutlich, dass der Schwerpunkt von JML auf der Verifikation von kleinen Programmteilen liegt und in größeren Projekten auf Grund des Umfanges des erzeugten Java-Quelltextes nicht überall benutzt werden kann.

3.3.3 Bewertung

Wie der Ansatz der *Reuse Contracts* ist auch der Ansatz von *JML* nicht in der Lage, die beschriebenen Fehlerquellen zu beheben. Da *JML* in Java und für Java entwickelt wurde, wäre es aber eine Möglichkeit, JML zum Erkennen von Methodenabhängigkeiten einzusetzen. Dies scheitert wiederum daran, dass das entsprechende Schlüsselwort *callable* noch nicht vom *JML*-Compiler interpretiert wird.

3.4 Checkstyle Eclipse Plug-in

Bisher wurden drei theoretische Lösungsansätze vorgestellt, die größtenteils noch nicht sinnvoll einsetzbar sind. Im Folgenden werden zwei Lösungen vorgestellt, die aus dem Open-Source-Bereich stammen und schon eingesetzt werden.

Das im Jahre 2003 von Oliver Burn veröffentliche *Checkstyle* Eclipse Plug-in (vgl. [Bur03]) ermöglicht es, während der Programmierung zu überprüfen, ob bestimmte Code-Styleguides eingehalten werden. Es ist eine Open-Source Software und liegt aktuell in der Version 3.1 vor.

3.4.1 Beschreibung

Das *Checkstyle Eclipse Plug-In* ist nach der Installation vollständig in Eclipse integriert. Über eine graphische Oberfläche kann eingestellt werden, welche Styleguides überwacht werden sollen. Dies sind einerseits triviale Styleguides, z. B., dass zu jeder Methode ein Javadoc Kommentar mit bestimmten Tags existieren soll oder bestimmte Formatierungen des Quelltextes einzuhalten sind.

Andererseits lassen sich auch bestimmte semantische Abhängigkeiten überprüfen. Das Plug-in kann auf Wunsch feststellen, ob alle importierten Klassen auch verwendet werden, oder ob eine lokale Variable eine andere verdeckt. Darüber hinaus kann sogar überprüft werden, ob bestimmte, in dem jeweiligen Projekt unerwünschte Muster bei der Programmierung verwendet wurden.

Als weitere Möglichkeit bietet das Plug-In die Überprüfung, ob auch die Methode hashCode überschrieben wird, falls dies für die Methode equals der Fall ist.

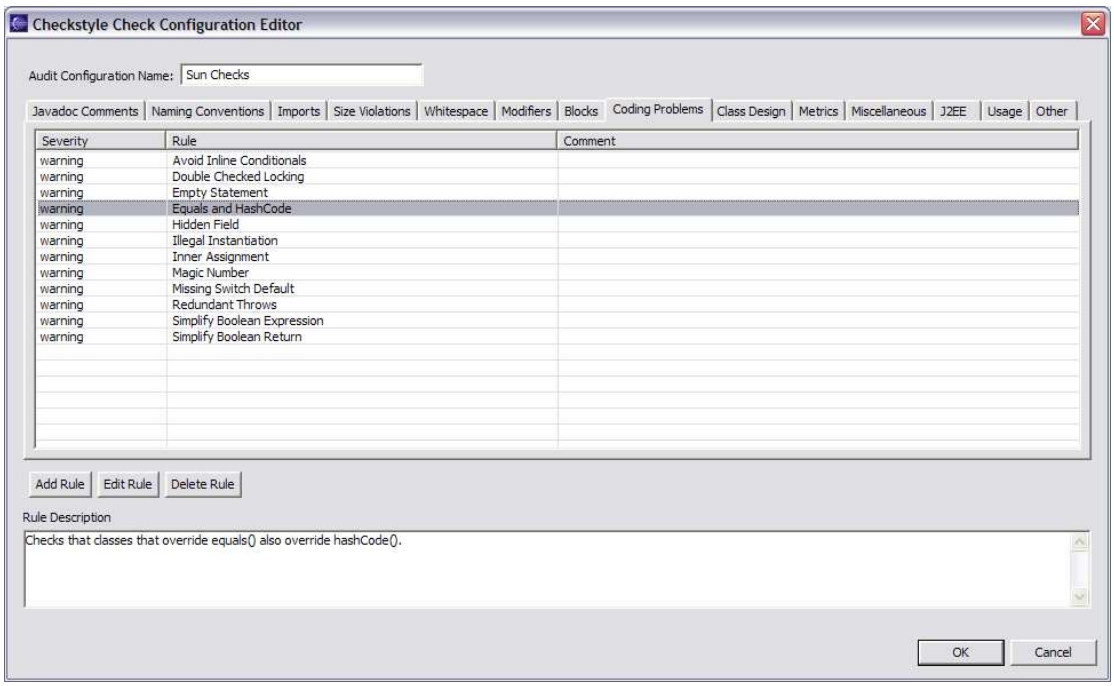

Abbildung 3.2: Checkstyle Eclipse Plug-In Konfigurationseditor

Die Abbildung 3.2 zeigt den Konfigurationseditor des *Checkstyle Eclipse Plug-In*. Im abgebildeten Fall ist das Plug-In so konfiguriert, dass es eine Warnung ausgibt, falls in einer Klasse die Methode *hashcode* nicht redefiniert wird, obwohl dies bei der Methode *equals* der Fall ist.

3.4.2 Bewertung

Mit dem *Checkstyle Eclipse Plug-in* ist es möglich, das in Kapitel 2.1 beschriebene, spezifische Problem der Methoden *hashcode* und *equals* zu beheben, indem es eine Warnung anzeigt, falls versehentlich vergessen wurde, die Methode *hashcode* zu redefinieren. Für die Überwachung anderer Methoden läßt sich das Plug-In nicht konfigurieren. Genausowenig ist es geeignet, die anderen Probleme zu beheben. Dies liegt daran, dass zwar die vorhanden Styleguides mit Hilfe von Mustern geändert werden können, diese Styleguides aber vor allem nur die Formatierung des Quelltextes bzw. den Aufbau einer Klasse betreffen.

3.5 PMD

Wie das *Checkstyle* Eclipse Plug-in ist *PMD* ein Open-Source Werkzeug, mit dem einzelne Java Quelltexte auf die Einhaltung bestimmter Styleguides hin untersucht werden können. Mit *PMD* kann ein Entwickler einerseits die Syntax eines Java Quelltextes überprüfen, z. B. ob bei jeder *while*-Schleife Klammern benutzt werden. Andererseits kann der Entwickler auch einige semantische Gegebenheiten überprüfen, z. B. ob imports doppelt vorhanden sind oder ob lokale Variablen unbenutzt sind.

3.5.1 Beschreibung

Die Grundlage für *PMD* ist immer ein einzelner Java Quelltext. Zuerst wird aus dem Java Quelltext mit Hilfe des Java Compiler Compilers (*JavaCC*) ein *Abstrakter Syntax Baum* (*AST*) erzeugt, auf dem PMD dann arbeitet. *PMD* überprüft, ob der *AST* allen bekannten Regeln entspricht. Die Regeln sind in *PMD* als Java Klasse definiert. Diese Regeln sind entweder schon Bestandteil von *PMD* oder können nachträglich entwickelt und in *PMD* eingefügt werden.

Als Regel ist alles denkbar, was auf Grund des *AST*s einer einzelnen Klasse entscheidbar ist. So lassen sich zum Beispiel Regeln schreiben, die überprüfen, ob der Java Quelltext bestimmten Normen entsprechend formatiert ist. Es kann auch eine Regel definiert werden, die überprüft, ob die Methode *hashcode* überschrieben wird, falls die Methode *equals* überschrieben wird. Da PMD immer nur eine Klasse für sich betrachtet, können allerdings keine allgemeingültigen Regeln geschrieben werden, die prüfen, ob eine Klasse entsprechend den Vorgaben Ihrer Oberklasse implementiert ist.

3.5.2 Bewertung

Wie das *Checkstyle* Eclipse Plug-in ist auch PMD nicht geeignet, die beschriebenen Fehlerquellen zu umgehen. PMD läßt sich zwar im Gegensatz zum *CheckStyle* Eclipse Plug-in durch selbstgeschriebene Regeln erweitern. Doch die Einschränkung, dass immer nur auf den *AST* einer einzelnen Klasse zugegriffen werden kann, verhindert, dass allgemeingültige Regeln für die Vererbungsschnittstelle geschrieben werden können. Dies liegt daran, dass bei der Vererbungsbeziehung mindestens zwei Klassen, nämlich die Unter- und die Oberklasse betroffen sind.

3.6 Zusammenfassung

Die vorgestellten Ansätze haben Wege aufgezeigt, wie bestimmte Fehlerquellen vermieden werden können, wenn zu Rahmenwerksklassen Unterklassen erzeugt werden sollen. Allerdings geht keiner der vorgestellten Ansätze diesen Weg zu Ende. Die zuerst vorgestellten, von der Theorie geprägten Ansätze wurden bisher nur in Prototypen umgesetzt und sind somit nicht in der Praxis einsetzbar und / oder sie beschäftigen sich überwiegend mit der Rahmenwerksversionierung. Die zuletzt vorgestellten Ansätze stammen alle aus dem *Open Source*-Bereich und werden kontinuierlich weiterentwickelt. Sie konzentrieren sich allerdings eher auf die Programmverifikation mittels des Vertragsmodells bzw. sind zu speziell ausgelegt, als dass die beschriebenen Fehlerursachen allgemein behoben werden könnten.

Die Tabelle 3.1 listet noch einmal die Fehlerquellen auf und verdeutlicht, inwieweit die vorgestellten Ansätze die Fehlerquellen vermeiden. Wie in diesem Kapitel gezeigt, ist keiner der vorgestellten Ansätze in der Lage, zwischen Überschreiben und Erweitern einer Methode zu unterscheiden. Mit *PMD* kann man spezielle Fälle von abhängigen bzw. zusammengehörige Methoden definieren, dies funktioniert aber nicht allgemeingültig und für jeden speziellen Fall müssen diese Werkzeuge extra angepaßt werden.

Tabelle 3.1: Vergleich bestehender Ansätze

Ansatz	Abhängige Methoden	Zusammengehörige Methoden	Erweitern vs Überschreiben	Methodenaufrufe aus Konstruktoren
Specialization interface	+	-	-	-
Reuse Contract	+	-	-	-
JML	+	-	-	-
OCL	-	-	-	-
CheckStyle	(+)	-	-	-
PMD	(+)	(+)	-	-

Checkstyle kann nur erkennen, ob auch die Methode *hashCode* überschrieben wird, falls die Methode equals in einer Klasse überschrieben wird.

Bei den theoretischen Ansätzen wurde gezeigt, dass diese insgesamt nicht dazu geeignet sind, die vorgestellten Fehlerquellen zu beheben. Zwar lassen sich zumindest abhängige Methoden beschreiben, doch selbst dazu sind teilweise schon Problemumgehungen notwendig, die der eigentlichen Intention der Ansätze wiedersprechen.

4 Extension Contracts

Nachdem im vorigen Kapitel gezeigt wurde, dass die bestehenden Lösungsansätze nicht geeignet sind, die vorgestellten möglichen Fehler zu verhindern, wird in diesem Kapitel ein neuer Lösungsansatz präsentiert. Dabei wird ein Zwischenweg zwischen den theoretischen Ansätzen und den pragmatischen Lösungen bestritten. Dieser Zwischenweg bezieht sich einerseits auf die fachliche Ebene, da die theoretischen Ansätze die Vererbungsbeziehung direkt kaum betrachten, sondern stattdessen überwiegend auf Probleme der Rahmenwerksversionierung abzielen und die pragmatischen Ansätze nicht mächtig genug sind, um die möglichen Fehler allgemeingültig erkennen zu können. Andererseits bezieht sich der Zwischenweg auch auf die technische Umsetzung. Der neue Ansatz kann zunächst genau die beschriebenen Fehlerquellen umgehen, damit er schlank genug ist, um im Gegensatz zu den theoretischen Lösungsansätzen im Javaumfeld eingesetzt werden zu können. Im Vergleich zu den bestehenden pragmatischen Lösungen ist er flexibler, so dass er bei Bedarf erweitert werden kann. Bei der Entwicklung dieses Lösungsansatzes ergibt sich somit eine agile, an *Extreme Programming* (vgl. [Bec99]) angelehnte Vorgehensweise .

4.1 Konzept

Es wird eine Beschreibungssprache entwickelt, mit der der Rahmenwerksentwickler die Erbenschnittstelle einer Klasse formaler als bisher beschreiben kann. Der Anwendungsentwickler kann nun anhand der formaleren Beschreibung der Erbenschnittstelle überprüfen, ob er seine Unterklassen korrekt implementiert hat. Der Rahmenwerksentwickler legt dabei einerseits fest, auf welche Weise die einzelnen Methoden redefiniert werden dürfen bzw. müssen. Andererseits garantiert er dafür, dass die vorgestellten Fehlerquellen nicht auftreten können, wenn sich der Anwendungsentwickler an die Vorgaben hält. Angelehnt an das Java-Schlüsselwort *extend*, das die Vererbungsbeziehung zwischen einer Unterklasse und deren Oberklasse definiert, ergibt sich somit für den hier vorgestellten Ansatz die Bezeichnung *Extension Contract*. Die Abbildung 4.1 verdeutlich dies noch einmal.

Die Abbildung zeigt, dass der Rahmenwerksentwickler zusätzlich zur Programmierung der Klasse nun die Aufgabe hat, den *Extension Contract* für diese Klasse zu definieren. Der Anwendungsentwickler muss zusätzlich zur Programmierung der Unterklasse noch den *Extension Contract* für die entsprechende Oberklasse beachten und ihn einhalten.

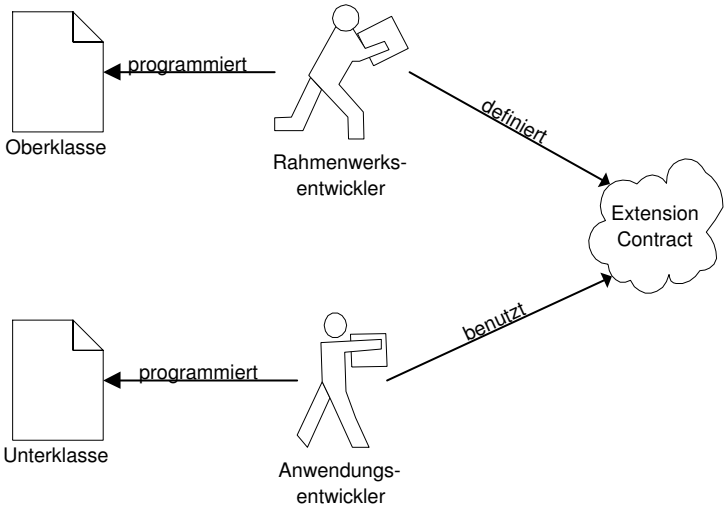

Abbildung 4.1: Grobkonzept der Extension Contracts

Die vorgestellten Ansätze sehen fast alle die Möglichkeit vor, die ergänzenden Beschreibungen im Quellcode zu definieren. Dieses *Single Source*-Prinzip (vgl. [RePo99], S.787) soll auch bei den *Extension Contracts* übernommen werden. Damit der Rahmenwerksentwickler nicht den Quellcode mitliefern muss, wenn er dem Anwendungsentwickler die *Extension Contracts* übergibt, ist für die Weitergabe der *Extension Contracts* eine entsprechende Datei (*Extension Contract*-Datei) vorgesehen. Die Erzeugung der *Extension Contract*-Datei anhand des Quelltextes wird durch ein Werkzeug (*Extension Contract Generator*) übernommen, so dass der Rahmenwerksentwickler weiterhin nur eine Datei, die *class*-Datei, bearbeitet. Einer der Hauptkritikpunkte am derzeitigen Status ist, dass nicht automatisch überprüft werden kann, ob der Anwendungsentwickler sich an den Vertrag der Erbenschnittstelle hält. Deshalb wird der Anwendungsentwickler durch ein Werkzeug (*Extension Contract Checker*) unterstützt, dass überprüft, ob die programmierten Unterklassen dem *Extension Contract* der jeweiligen Oberklasse entsprechen. Die Abbildung 4.2 zeigt dieses verfeinerte Konzept.

Im folgenden wird auf die einzelnen vorgestellten Bestandteile dieses neuen Ansatzes eingegangen. Dabei wird zunächst die Beschreibungssprache eingeführt, und deren Eigenschaften beschrieben. Darauf aufbauend werden dann die entwickelten Werkzeuge vorgestellt. Im Anschluss daran wird auf die *Extension Contract*-Datei eingegangen.

4.2 Die Extension Contract Beschreibungssprache

Wie im vorherigen Unterkapitel geschrieben, soll die Beschreibung der Erbenschnittstelle in der Quelltext-Datei vorgenommen werden. Da der Ansatz der *Extension Contracts* für die vorliegende Arbeit für die Programmiersprache *Java* umgesetzt wurde,

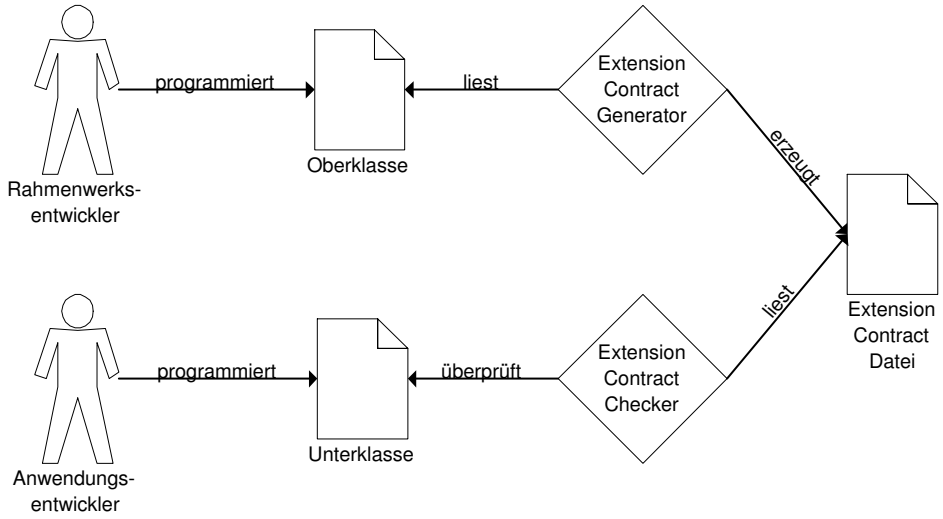

Abbildung 4.2: Verfeinertes Konzept der Extension Contracts

bietet es sich hier an, die Beschreibungssprache als *Javadoc*-Kommentar zu implementieren. Dadurch ergibt sich ohne zusätzlichen Aufwand die Möglichkeit, den *Extension Contract* einer Klasse ohne den Quellcode der Klasse weiterzugeben. Außerdem ist der *Javadoc*-Kommentar eine der wichtigsten Nachschlagewerke eines Anwendungsentwicklers, so dass sich im *Javadoc*-Kommentar notierte *Extension Contracts* nahtlos in die Arbeitsumgebung des Anwendungsentwicklers integrieren.

Der *Javadoc*-Kommentar einer Klasse oder Methode ist typischer Weise so aufgebaut, dass zuerst ein Fließtext das Verhalten der Klasse oder Methode beschreibt und danach mit einigen speziellen *Javadoc*-Tags dieses Verhalten noch näher definiert wird. Zum Beispiel kann mit den *Javadoc*-Tags *param* und *return* definiert werden, welche Werte die Parameter haben dürfen und welches die möglichen Rückgabewerte sind. Für die Methode *sqrt(y)*, die in Kapitel 3.3 vorgestellt wurde, könnte dieses wie folgt aussehen:

```
/**
 * Diese Methode berechnet die Quadratwurzel aus der gegebenen
 * positiven Zahl.
 *
 * @param y Positive Zahl, aus der die Wurzel berechnet wird.
 * @return Die Quadratwurzel der gegeben Zahl. Diese ist in
 * jedem Fall größer oder gleich Null.
 */

public int sqrt(y)
{
  ...
}
```

Der Nachteil der vorhandenen *Javadoc*-Tags ist, dass diese ohne zusätzliche Werkzeuge nicht maschinell überprüfbar sind. Da für den Einsatz der *Extension Contracts* aber in jedem Fall neue Werkzeuge für den Rahmenwerks- und Anwendungsentwickler entwickelt werden müssen, bietet es sich an, die *Extension Contracts* mit Hilfe von formal definierten *Javadoc*-Tags zu beschreiben. Dabei wird für die unterschiedlichen Fehlerquellen jeweils mindestens ein entsprechendes *Javadoc*-Tag definiert, auf dessen Grundlage dann automatisch überprüft werden kann, ob der Anwendungsentwickler die Vorgaben des Rahmenwerkentwicklers eingehalten hat.

Bei der Festlegung, dass der Ansatz der Extension Contracts auf Javadoc-Tags beruhen soll, stellt sich zunächst die Frage, welche Möglichkeiten dadurch modelliert werden können. So wird im Lösungsansatz von Gary Leavens (vgl. refjml) aus der Beschreibungssprache zunächst zusätzlicher Quelltext generiert. Für den Ansatz der der *Extension Contracts* wurde erwogen für die *Extension Contract Tags* ähnliches zu ermöglichen. Diese sogenannten *dynamischen Tags* hätte zur Laufzeit die Aufrufreihenfolge von Methoden innerhalb des Programmflusses kontrolliert werden können. Dieses läßt sich allerdings mit etwas mehr Aufwand für den Rahmenwerksentwickler auch mit Hilfe des Vertragsmodells implementieren. Deshalb wurde davon abgesehen, *dynamische Tags* in den neuen Ansatz aufzunehmen, um den Fokus auf die sogenannten *statischen Tags* zu setzen, mit denen die vorgestellten Fehlerquellen umgangen werden können. Die *statischen Tags* tragen ihre Bezeichnung, da ihre Einhaltung statisch anhand des Quellcodes überprüft werden kann. Es wird also kein neuer Quelltext erzeugt und damit auch nicht die spätere Programmausführung beeinflusst.

4.2.1 *Extension Contract*-Tags

Im Folgenden werden kurz die bisher implementierten *Extension Contract Tags* mit ihrem Haupteinsatzzweck vorgestellt. Eine komplette Sprachbeschreibung findet sich in Anhang A.

Extension Contract-Tags *extend* und *override*

In Kapitel 2.3 wurde das Problem beschrieben, dass ein Anwendungsentwickler nicht erkennen kann, ob er eine bestimmte Methode, die er redefinieren kann, überschreiben oder erweitern soll. Dieses Problem wird durch die beiden Tags *override* und *extend* umgangen. Mit Hilfe dieser Tags kann ein Rahmenwerksentwickler festlegen, wie der Anwendungsentwickler einzelne Methoden redefinieren kann. Beide Tags können auch mit der Option *must* benutzt werden. Diese Option bedeutet, dass die betreffende Methode in Unterklassen überschrieben bzw. erweitert werden muss. Dieses scheint zunächst für das Tag *override* keinen Sinn zu machen, da ja eine abstrakte Methode in einer Oberklasse auch dazu führen würde, dass diese Methode in einer Unterklasse überschrieben werden muss. Der Vorteil des Tags *override* liegt darin, dass die Oberklasse keine abstrakte Klasse sein muss, so dass gegebenenfalls Objekte dieser Klasse

erzeugt werden können. Dieses kann zum Beispiel zum Testen von Klassen bei der Verwendung von *Unit-Tests* (vgl. z. B. [Bec03], [Lin02]) eine Erleichterung bringen.

Soll eine Methode in einer Unterklasse erweitert werden, stellt sich für den Anwendungsentwickler die Frage, wann er die Implementierung der Oberklasse aufrufen muss. Es hängt von der konkreten Methode ab, ob die Implementierung der Oberklasse am Anfang der Methode der Unterklasse, an deren Ende oder an einer beliebigen Position aufgerufen werden muss.

Um dieses zu beschreiben, kann der Rahmenwerksentwickler das Tag *extend* mit einer der Optionen *pre* und *post* benutzen. Dabei bedeutet *extend pre*, dass die Implementierung der Methode der Unterklasse vor dem Aufruf der Implementierung der Oberklasse ausgeführt werden muss. Entsprechend bedeutet *extend post*, dass in der Implementierung der entsprechenden Methode der Unterklasse zuerst die Implementierung der Oberklasse aufgerufen werden muss. Wird keine der beiden genannten Optionen angegeben, so kann der Aufruf der Implementierung der Oberklasse an beliebiger Stelle innerhalb der Methode der Unterklasse erfolgen. Um sicherzustellen, dass der Aufruf der Methode der Oberklasse wirklich stattfindet, wird dabei die Einschränkung vorgenommen, dass dieser Aufruf auf oberster Ebene innerhalb der Methode stattfindet, also nicht in einer Programmstruktur wie zum Beispiel der Anweisung *if* verschachtelt ist. Im Folgenden ist ein kleines Beispiel gegeben, das den Einsatz des Tags *extend* demonstriert:

```
/**
 * @extend post // anschließend Subwerkzeuge initialisieren
 */

protected void equip ()
{
  _isEquipped = true;
}
```

Am Beispiel wird deutlich, dass das Setzen der Membervariablen *_isEquipped* vorgenommen werden muss, bevor in der Unterklasse in dieser Methode weitere Methoden aufgrufen werden dürfen.

Extension Contract-Tags *coupled_with* und *redefinde_with*

Ein weiteres Problem für den Anwendungsentwickler sind von einander abhängige Methoden (vgl. Kapitel 2.1 und 2.2). Dabei ist einerseits zwischen Methoden zu unterscheiden, die wechselseitig voneinander abhängen und andererseits Methoden, bei denen die Abhängigkeit nur in einer Richtung besteht, wie zum Beispiel bei den Methoden *equals* und *hashcode*. Der Fall, dass die Methoden wechselseitig voneinander abhängen, kann mit dem Tag *coupled_with* abgebildet werden. Dieses Tag muss im *Javadoc*-Kommentar jeder betreffenden Methode notiert werden. Dabei müssen die jeweils anderen Methoden aufgeführt werden. Alternativ hierzu würde es reichen, wenn

die voneinander abhängigen Methoden an einer Stelle, z. B. im *Javadoc*-Kommentar der Klasse notiert werden. Dieses Vorgehen hätte jedoch zum einen den Nachteil, dass dieser Kommentar nicht automatisch im Fokus der Arbeit ist, wenn der Rahmenwerksentwickler Änderungen an den betreffenden Methoden vornimmt. Zum anderen wäre es auch nicht möglich, eine Konsistenzprüfung der beschriebenen Abhängigkeiten vorzunehmen, wie dies bei einem Kommentar zu jeder betreffenden Methode möglich ist. Im Folgenden ist zu sehen, wie der Einsatz des Tags *coupled_with* aussehen kann:

```
/**
 * Notice: You have to redefine canCreateFromString(),
 * value(String) and isValid(String) always together
 * to ensure consistency.
 * @override
 * @coupled_with valueImpl(), isValidImpl()
 */
public boolean canCreateFromString () {
  return false;
}

/**
 * [...]
 * @override
 * @coupled_with canCreateFromString(), isValidImpl(String)
 */
protected DomainValue valueImpl (String s) {
  return null;
}

/**
 * [...]
 * @override
 * @coupled_with canCreateFromString, valueImpl(String)
 */
protected boolean isValidImpl (String s) {
  return false;
}
```

Das Beispiel zeigt, wie der Rahmenwerksentwickler für die drei Methoden *canCreateFromString*, *valueImpl* und *isValidImpl* definieren kann, dass sie alle überschrieben werden müssen, sobald dies für eine einzige von ihnen der Fall ist. Dazu wird zunächst mit Hilfe des Tags *override* zu jeder einzelnen Methode definiert, dass diese überschrieben werden soll. Dieses ergibt sich aus dem konkreten Beispiel, in anderen Fällen ist es möglich, dass einige Methoden überschrieben und andere erweitert werden sollen. Anschließend wird definiert, dass jede einzelne Methode jeweils von den anderen beiden abhängt.

Das Tag *redefine_with* beschreibt, dass beim Redefinieren einer Methode eine bestimmte andere Methode auch redefiniert werden muss. Im Beispiel der Methoden *equals* und *hashcode* bedeutet dies, dass nur im *Javadoc*-Kommentar der Methode *equals* das Tag *redefine_with* wie folgt notiert würde:

```
/**
 * @redefine_with hashCode()
 */
public boolean equals(Object obj) {
...
}
```

Extension Contract-Tags *redefine_restricted* und *calls*

Auch das zuletzt vorgestellte Problem der polymorphen Methodenaufrufe aus einem Konstruktor heraus läßt sich mit dem Ansatz der *Extension Contracts* beheben. Dazu kann der Rahmenwerksentwickler Methoden, die vom Konstruktor aus aufgerufen werden, gesondert zum Überschreiben durch den Anwendungsentwickler freigeben. Dies geschieht mit dem Tag *redefine_restricted*. Dieses Tag bedeutet für den Anwendungsentwickler, dass er die entsprechenden Methoden nur mit Einschränkungen redefinieren darf. So ist es zum Beispiel verboten, auf Membervariablen der Unterklasse zuzugreifen, da diese noch nicht initialisiert sind.

Wird eine Methode aus einem Konstruktor heraus polymorph aufgerufen und führt der Rahmenwerksentwickler diese Methode nicht explizit mit dem Tag *redefine_restricted* auf, wird automatisch ein *calls*-Tag erzeugt und in den *Extension Contract* aufgenommen. Dieses signalisiert, dass die betreffende Methode nicht redefiniert werden darf.

4.3 Eigenschaften von Extension Contracts

Im Folgenden wird auf die Eigenschaften von *Extension Contracts* eingegangen. Die Eigenschaften lassen sich dabei überwiegend aus dem Einsatzzweck der *Extension Contracts* und ihrer Beziehung zur beschriebenen Methode ableiten.

Extension Contracts sind nicht auf Interfaces anwendbar

Wie in diesem Kapitel bisher gezeigt wurde, beschreiben *Extension Contracts* zu einer Methode, wie sich die Implementation einer redefinierenden Methode in einer Unterklasse in Bezug auf die Implementierung der Oberklasse verhalten soll. Daraus folgt, dass *Extension Contracts* weder für Interfaces noch für abstrakte Methoden einen Sinn haben.

Extension Contracts werden vererbt

Da sich Extension Contracts auf die Implementierung einer konkreten Methode beziehen, werden sie auch zusammen mit dieser Methode weitervererbt, solange bis diese Methode in einer Unterklasse redefiniert wird.

Redefinieren von Extension Contracts

Wird eine Methode in einer Unterklasse redefiniert, wird automatisch der *Extension Contract* vor weiteren Unterklassen verborgen, da ja auch die Implementierung, zu der der *Extension Contract* gehört, für diese verborgen ist. Falls der Entwickler der Unterklasse diesen *Extension Contract* trotzdem aufrechterhalten möchte, oder einen neuen definieren möchte, so kann er dies einfach dadurch tun, indem er diesen in den Javadoc-Kommentar der redefinierenden Methode schreibt.

Extension Contracts müssen auch von abstrakten Klassen eingehalten werden

Dieses bedeutet, dass sich auch abstrakte Klassen an den vorgestellten *Extension Contract* der Klasse *Object* halten müssen, falls sie direkte Unterklassen sind. Das heißt, falls sie die Methode *equals* redefinieren, müssen sie auch die Methode *hashcode* redefinieren.

Es könnte jetzt argumentiert werden, dass die Methode *hashcode* nicht in der abstrakten Klasse selbst redefiniert werden muss, da von dieser Klasse keine Objekte erzeugt werden können. Ausserdem könnte die Methode ja noch in Unterklassen dieser abstrakten Klasse definiert werden, so dass dann der *Extension Contract* der Klasse *Object* wieder eingehalten wäre.

Diese Argumentation läßt allerdings einen wichtigen Punkt außer acht. *Extension Contract* beziehen sich immer auf eine bestimmte Implementierung und gerade bei Methoden, die zusammen redefiniert werden sollen, müssen dann auch die entsprechenden Implementierungen zusammen passen. Doch woher soll der Entwickler der weiteren Unterklasse wissen, wie er die Methode *hashcode* überschreiben muss, damit sie zur Methode *equals* der abstrakten Klasse passt? Eventuell ist er nicht mal in der Lage, eine solche zu implementieren, da ihm der Zugriff auf entscheidende (*private*-) Teile der abstrakten Klasse fehlt.

4.4 Der Extension Contract Generator

Der *Extension Contract Generator* ist ein Programm, das Java-Quellcodes nach den beschriebenen JavaDoc-Tags durchsucht und entsprechende *Extension Contract*-Dateien erzeugt. Dabei wird zu jeder Quellcode-Datei eine *Extension Contract*-Datei erzeugt. Der Generator kann sowohl zu einzelnen Quellcode-Dateien die *Extension Contract*-Datei erzeugen, als auch für ganze Packages und Unterpackages. Der Generator wird im Standardfall vom Rahmenwerksentwickler benutzt, der mit der Hilfe des *Extension Contract Generator* die *Extension Contract*-Dateien erzeugt. Diese können dann losgelöst vom Java Quellcode weitergeben werden.

Als Eingabeparameter benötigt der *Extension Contract Generator* die Java Quellcode-Datei, zu der der *Extension Contract* erzeugt werden soll. Wahlweise kann der *Extension Contract Generator* auch auf Verzeichnissen arbeiten. Dann wird zu jeder Java

Quellcode-Datei in dem gegebenen Verzeichnis und in allen Unterverzeichnissen eine *Extension Contract*-Datei erzeugt. Ein weiterer benötigter Eingabeparameter ist das Verzeichnis, in das die *Extension Contract*-Dateien abgespeichert werden sollen. Dabei erzeugt der *Extension Contract Generator* eine Unterverzeichnisstruktur entsprechend den in den Java Quellcode-Dateien angegebenen Packages.

Ein Auruf, um zu allen Java Quellcode-Dateien im Verzeichnis *d:\java_beispiel* die entsprechenden *Extension Contract*-Dateien im Verzeichnis *d:\contracts* zu erzeugen, könnte dementsprechend wie folgt aussehen (hierbei wird vorrausgesetzt, dass sich alle benötigten *Jar*-Dateien im Klassenpfad befinden):

```
java de.extensioncontract.ECGenerator d:\java_beispiel d:\contracts
```

Von der Funktionsweise her lässt sich der *Extension Contract Generator* in drei einzelne Teile zerlegen. Die erste und grundlegende Funktion ist die eines Parsers. Wie beschrieben werden die einzelnen Quellcodes nach den *Extension Contract* Tags durchsucht. Außerdem werden die einzelnen Methoden-Signaturen ermittelt. Die zweite Funktion ist die semantische Überprüfung bestimmter Tags. Zum Beispiel wird geprüft, ob die *coupled_with*-Tags konsistent zueinander sind. Ausserdem wird überprüft, ob jeder relevante Methodenaufruf innerhalb eines Konstruktors durch ein *redefine_with*-Tag bekannt gemacht wird. Falls dieses nicht der Fall ist, wird automatisch ein entsprechendes *calls*-Tag erzeugt. Die dritte Funktion ist die Abspeicherung der erkannten, geprüften und gegebenenfalls erzeugten *Extension Contracts* in die *Extension Contract*-Dateien.

4.5 Der Extension Contract Checker

Der *Extension Contract Checker* dient dazu, Java Sourcecodes gegen bestehende *Extension Contract*-Dateien zu prüfen. Dabei wird zu jeder Javaklasse geprüft, ob es zur entsprechenden Oberklasse eine *Extension Contract*-Datei gibt. Ist dies nicht der Fall, versucht der Checker die zugehörige Source-Datei, in der potenziell noch *Extension Contracts* definiert sind, oder als letztes die *class*-Datei zu öffnen. Dies ist nötig, um die Klassenhierarchie bis zur Klasse *Object* zurückzuverfolgen und eventuell von Klassen, die in der Hierarchie höher angesiedelt sind, den *Extension Contract* überprüfen zu können.

Typischer Weise wird der *Extension Contract Checker* von Anwendungsentwicklern benutzt, die überprüfen wollen, ob die von ihnen programmierten Unterklassen den gegebenen *Extension Contracts* der entsprechenden Rahmenwerksklassen entsprechen. Als Eingabeparameter benötigt der *Extension Contract Checker* den Pfad der zu prüfenden Datei bzw. des zu prüfenden Verzeichnisses. Die *Extension Contract*-Dateien müssen sich im Klassenpfad befinden. Um also zu überprüfen, ob die Java Quellcodes im Verzeichnis *d:\sub_beispiel* den *Extension Contracts* in den im letzten Unterkapitel erzeugten *Extension Contracts*-Dateien entsprechen, sieht der Aufruf wie folgt aus:

```
java -cp % classpath %;d:\_contracts
de.extensioncontract.checker.ExtensionContractChecker d:\sub_beispiel
```

Auch der *Extension Contract Checker* vereint mehrere Funktionseinheiten in sich. Zuerst stellt er, wie der *Extension Contract Generator* einen Parser dar, der den Java-Quellcode analysiert und die Methodensignaturen ermittelt. Weiterhin werden auch die Methodenaufrufe innerhalb der einzelnen Methoden ermittelt, da diese bei der späteren Überprüfung der *Extension Contract* Tags eine wesentliche Rolle spielen.

Im nächsten Schritt wird zu einer Java-Quellcode-Datei die Oberklasse bzw. rekursiv die Oberklassen ermittelt und daraus ein einziger *Extension Contract* für diese Java-Quellcode-Datei erzeugt. Dabei werden, wie im Kapitel 4.3 dargestellt, die *Extension Contract*-Tags von einer Oberklasse zur Unterklasse weitervererbt, bis die entsprechenden Methoden redefiniert werden.

Als letztes folgt nach diesen Vorbereitungen die Prüfung, ob der Java-Quellcode dem *Extension Contract* entspricht. Dazu wird zu jeder Methode überprüft, ob sie von einem der ermittelten *Extension Contract*-Tags betroffen ist, und ob dieses Tag dann korrekt umgesetzt wurde. Der *Extension Contract Checker* gibt zum Schluß zu jeder Java Quellcode-Datei eine entsprechende Meldung aus. Im Fehlerfall beschreibt die Meldung, welche Methode betroffen ist, und welches Tag nicht korrekt umgesetzt wurde.

4.6 Die Extension Contract-Datei

Die *Extension Contracts* werden in einer Datei einer DTD (vgl. Anhang C) entsprechend im XML-Format abgespeichert. Dies hat viele positive Effekte:

Erstens können die *Extension Contracts* ohne den Quellcode weitergegeben werden, was bei einem kommerziellen Projekt in vielen Fällen nicht möglich wäre. Zweitens braucht ein Anwendungsentwickler nicht den ganzen Quellcode zu analysieren, wenn er sich den *Extension Contract* einer Methode ansehen möchte.

Ein anderer Aspekt ist, dass diese Dateien auch von Anwendungsentwicklern nachträglich erstellt oder verändert werden können. Dies ist z. B. dann sinnvoll, wenn ein bestehendes Rahmenwerk, zu dem kein Quellcode vorhanden ist, verwendet werden soll. Die Anwendungsentwickler, die bestimmte Fehlermöglichkeiten beim Beerben von Klassen aus diesem Rahmenwerk feststellen, können per Hand eine dementsprechende *Extension Contract*-Datei erstellen. Danach können neu erstellte Unterklassen gegen diese *Extension Contracts* geprüft werden. Z.B. kann auf diese Weise eine *Extension Contract*-Datei zur Klasse *Object* wie folgt geschrieben werden. Sie sorgt dafür, dass die Methode *hashcode* überschrieben werden muss, falls die Methode *equals* überschrieben wird:

```
<class name="Object" scope="public">
  <method name="equals" scope="public" type="boolean">
    <params><type>java.lang.Object</type></params>
    <comment>
      <doctag type="redefine_with">
        <tagattribute>hashCode()</tagattribute>
      </doctag>
    </comment>
  </method>
</class>
```

Das XML-Format sorgt dafür, dass die Dateien von vielen Programmen sinnvoll angezeigt und bearbeitet werden können, indem die entsprechenden Programme die Struktur der Datei hervorheben und bestimmte Teile der Datei ein- und ausblenden können.

4.7 Vergleich mit den bestehenden Ansätzen

Mit dem Ansatz der *Extension Contracts* wurde in diesem Kapitel gezeigt, dass es möglich ist, die vorgestellten Fehlerquellen zu erkennen und zu umgehen. Dabei wurde für jede vorgestellte Fehlerquelle mindestens ein *Extension Contract* Tag eingeführt, mit dessen Hilfe diese Fehlerquelle erkannt werden kann. Im Vergleich zu den vorgestellten bestehenden Ansätzen bildet der Ansatz der *Extension Contracts* eine pragmatische Lösung zwischen den eher theoretischen Ansätzen und den sehr speziellen Anwendungen.

Im Vergleich zu den theoretischen Ansätzen bieten die *Extension Contracts* eine einfach umzusetzende Lösung, um es Anwendungsentwicklern zu ermöglichen, funktionierende Unterklassen zu vorhandenen Rahmenwerksklassen zu schreiben. Wie im Kapitel 3 gezeigt wurde, befinden sich die Tools zu diesen Ansätzen, soweit es sie gibt, einerseits noch im Entwicklungsstadium (vgl. Kapitel 3.1, 3.2 und 3.3). Andererseits konzentrieren sich diese Ansätze auf die Programmverifikation (vgl. Kapitel 3.3) oder auf Versionierung von Rahmenwerken (vgl. Kapitel 3.3 und 3.2), und sind somit auch theoretisch nur in der Lage, einen kleinen Teil der aufgezeigten Fehlerquellen zu erkennen.

Betrachtet man die bestehenden Anwendungen, so sieht man, dass sie sich einerseits auf das Konzept des Vertragsmodells konzentrieren, oder andererseits zu speziell sind. So kann mit dem CheckStyle Plugin zwar die Beziehung der Methoden *equals* und *hashcode* zueinander überprüft werden, doch ist dies nicht auf andere Methoden übertragbar. Zwar lassen sich die vorgestellten Anwendungen zum Teil durch eigenen Java-Code (vgl. Kapitel 3.5) erweitern, doch existiert hier nicht das Konzept der unterschiedlichen Fehlerquellen, wie es im Ansatz der *Extension Contracts* durch die unterschiedlichen Tags verdeutlicht wird. Wie im Kapitel 5 noch gezeigt wird, kann der Ansatz der *Extension Contracts* auch noch um neue Funktionen erweitert werden.

Tabelle 4.1: Vergleich bestehender Ansätze mit Extension Contracts

Ansatz	Abhängige Methoden	Zusammengehörige Methoden	Erweitern vs Überschreiben	Methodenaufrufe aus Konstruktoren
Specialization interface	+	-	-	-
Reuse Contract	+	-	-	-
JML	+	-	-	-
OCL	-	-	-	-
CheckStyle	(+)	-	-	-
PMD	(+)	(+)	-	-
Extension Contracts	+	+	+	+

In der Tabelle 4.1 wurde zusätzlich zu den vorhandenen Ansätzen der Ansatz der *Extension Contracts* zum Vergleich mit aufgenommen. Auf den ersten Blick ist ersichtlich, dass der Ansatz der *Extension Contracts* der einzige ist, der überhaupt zwischen überschreiben und erweitern einer Methode unterscheidet und kontrolliert, dass keine Methode aus einem Konstruktor heraus polymorph aufgerufen wird.

Zusammengefasst ergibt sich, dass der Ansatz der *Extension Contracts* der einzige Ansatz ist, der einsetzbar ist; es gibt entsprechende Werkzeuge für den Rahmenwerks- und den Anwendungsentwickler und diese Werkzeuge sind einfach benutzbar. Er ist der einzige Ansatz, mit dem alle beschriebenen Fehlerquellen erkannt und umgangen werden können. Und schlußendlich ist der Ansatz der *Extension Contracts* allgemein genug, um bestimmte Fehlerquellen generell ausschliessen zu können und nicht nur einzelne vorher bestimmte Ausprägungen.

5 Umsetzung der Extension Contracts

In diesem Kapitel werden einige Implementierungsdetails des vorgestellten Ansatzes beschrieben. Dabei werden einerseits die wesentlichen Klassen der beiden Werkzeuge vorgestellt. Andererseits werden die Klassen vorgestellt, die wichtig für die Erweiterung des vorgestellten Ansatzes sind.

5.1 Der Extension Contract Generator

Der *Extension Contract Generator* besteht aus drei Teilen. Den wesentlichen Teil stellt die Parserkomponente dar. Sie basiert auf dem Parsergenerator ANTLR in der Version 2.7.1 [Par03]. Bei diesem wird eine Grammatik für die Sprache Java der Version 1.3 mitgeliefert. Auf Grundlage dieser Grammatik wurde die Grammatik für die *Extension Contracts* entwickelt. Die von ANTLR erzeugten Klassen liefern als Ergebnis einen Abstrakten Syntax Baum (AST), der zur Analyse der vorhandenen Methoden sehr unhandlich ist. Deshalb überführt die Parserkomponente des *Extension Contract Generators* diesen in ein Objektgeflecht, wobei zu den Java-Konstrukten *class*, *constructor* und *method* jeweils eine entsprechende Klasse existiert.

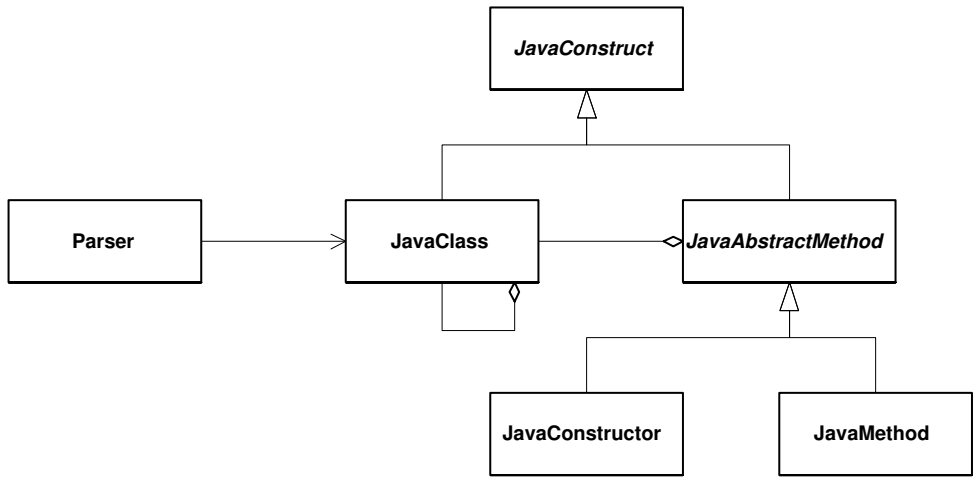

Abbildung 5.1: Klassendiagramm der wichtigsten Klassen der Parserkomponente

Die Abbildung 5.1 zeigt die Klasse *Parser* und die Klassen, die den zu parsenden Java-Konstrukten entsprechen. Es ist zu sehen, dass Klassen genau wie in Java üblich weitere Klassen (*inner classes*) als auch Methoden und Konstruktoren enthalten. Die Klasse *JavaConstruct* enthält Methoden für den Umgang mit den gemeinsamen Eigenschaften, die alle aufgeführten Java Konstrukte haben. Alle haben einen Namen, eine bestimmte Sichtbarkeit und zu allen kann es einen Javadoc-Kommentar geben. Daraus resultieren dann entsprechende Methoden an der Klasse *JavaConstruct*, wie in Abbildung 5.2 zu sehen ist.

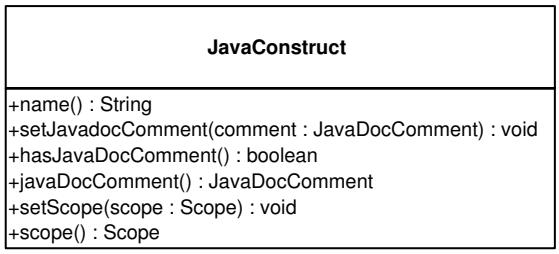

Abbildung 5.2: Die Klasse JavaConstruct

In der Abbildung ist weiterhin zu sehen, dass es keine Methode *hasScope* gibt, die man z. B. analog zur Methode *hasJavaDocComment* erwarten könnte. Dies liegt daran, dass die Sichtbarkeit von Klassen und Methoden immer genau definiert ist, auch wenn sie nicht vom Entwickler festgelegt wird. Hingegen ist es durchaus möglich, dass zu einer Klasse oder einer Methode kein Javadoc-Kommentar existiert.

Da Methoden und Konstruktoren einige gemeinsame Eigenschaften haben, wurde für diese beiden noch eine abstrakte Klasse *AbstractMethod* eingefügt (vgl. Abbildung 5.1), die diese Eigenschaften repräsentiert. Zu den gemeinsamen Eigenschaften zählen zum Beispiel, dass sowohl aus Methoden als auch aus Konstruktoren andere Methoden aufgerufen werden können. Die Methodenaufrufe werden mit den Methoden *addMethodCall*, *hasMethodCall* und *methodCalls* verwaltet (vgl. Abbildung 5.3). Ein Unterschied ist, dass Methoden im Gegensatz zu Konstruktoren einen Rückgabetyp haben, auf den mit den Methoden *setType* und *type* zugegriffen werden kann.

Zur Laufzeit erzeugt der Parserteil des *Extension Contract Generators* ein Objekt der Klasse *JavaClass*, welches auch alle Konstruktoren, Methoden und *inner classes* enthält. Dabei werden für die Konstruktoren und Methoden auch alle Methodenaufrufe, die in ihnen vorkommen, über die Methode *addMethodCall* berücksichtig. Anschließend wird überprüft, ob die angegebenen *Extension Contract Tags* semantische Wiedersprüche enthalten. Ist dies nicht der Fall wird das *JavaClass*-Objekt als XML-Datei abgespeichert. Dabei werden die XML-Dateien in eine den Quelltext- und Code-Dateien entsprechende Verzeichnisstruktur erzeugt.

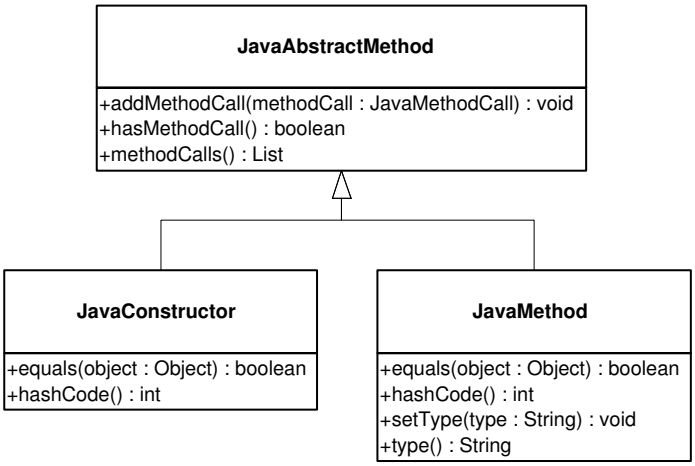

Abbildung 5.3: Klassenhierarchie zur Klasse *AbstractMethod*

5.2 Extension Contract Tags

Extension Contract Tags werden durch die Klasse *ExtensionContractTag* repräsentiert. Die Abbildung 5.4 zeigt die Schnittstelle dieser Klasse. Sie enthält die Methoden *addAttribute* und *attributes* für die Verwaltung aller optionalen Angaben der Tags. Zum Beispiel können bei dem Tag *coupled_with* beliebig viele Methoden aufgeführt werden. Diese Schnittstelle ist generisch gehalten, damit der Parser ohne Kenntnis eines speziellen Tags in der Lage ist, die zugehörigen Objekte zu erzeugen.

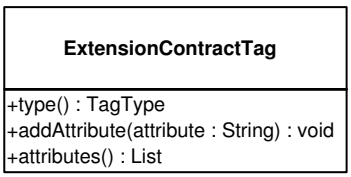

Abbildung 5.4: Die Klasse ExtensionContractTag

Um die einzelnen *Extension Contract Tags* unterscheiden zu können, enthält die Klasse die Methode *type()* die den Typ des Tags zurückliefert. Der Typ kann nur einer der vorgestellten *Extension Contract Tags* sein. Deshalb wurde die Klasse *TagType* als Aufzählungstyp im Wesentlichen nach dem von Joshua Bloch vorgeschlagenem *typesafe enum* Muster (vgl. [Blo01], S. 104ff) entwickelt. Abbildung 5.5 zeigt die Schnittstelle der Klasse *TagType*.

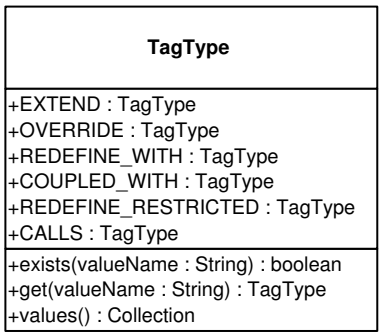

Abbildung 5.5: Der Aufzählungstyp TagType

Die Klasse enthält für jeden *Extension Contract Tag*-Typ ein konstantes Klassenattribut mit entsprechendem Namen. Zusätzlich enthält die Klasse im Gegensatz zum von Bloch vorgeschlagenen Muster drei statische Methoden. Diese verwalten zur Laufzeit alle vorhanden Konstanten und ermöglichen über die Methode *exists* die Abfrage, ob der als *String* übergebene Bezeichner ein *Extension Contract Tag* ist. In der Schnittstelle gibt es keine Methode zum Hinzufügen von *TagType*-Objekten, da diese nach dem verwendeten Muster nur über einen privaten Konstruktor erzeugt werden können, der direkten Zugriff auf die Klassenattribute hat. Somit wird eine entsprechene Methode nicht benötigt. Zur Laufzeit werden also bei der Initialisierung die konstanten Werte erzeugt. Dabei sammelt der Konstruktor die Wertebezeichner in einem Behälter, auf dem dann die drei statischen Methoden arbeiten.

5.3 Tag Checker

Zu jedem *Extension Contract Tag* gibt es eine Klasse, die das Einhalten dieses Tags überprüft. Es wird zwischen zwei Arten von *Tag Checkern* unterschieden. Die einen überprüfen die Tags, die Methoden betreffen, die anderen überprüfen die Tags, die sich auf Konstruktoren beziehen. Dementsprechend gibt es zwei Oberklassen für die beiden unterschiedlichen Tag-Typen. Die Klasse *ExtensionContractTagChecker* ist die Oberklasse für die *Checker* der Tags *override*, *extend*, *redefine_with* und *coupled_with*. Für die *Checker*, die die Tags *redefine_restricted* und *calls* überprüfen, bildet die Klasse *ExtensionContractConstructorTagChecker* die Basisklasse.

Die Abbildung 5.6 zeigt die wesentlichen Methoden der Schnittstelle der Basisklassen *ExtensionContractTagChecker* und *ExtensionContractConstructorTagChecker*. Die Methodennamen der Klassen sind identisch und sollen auch jeweils ein identisches Verhalten aufweisen. Die Methoden *check* und *doCheck* weisen jedoch für die beiden unterschiedlichen Klassen auch unterschiedliche Signaturen auf.

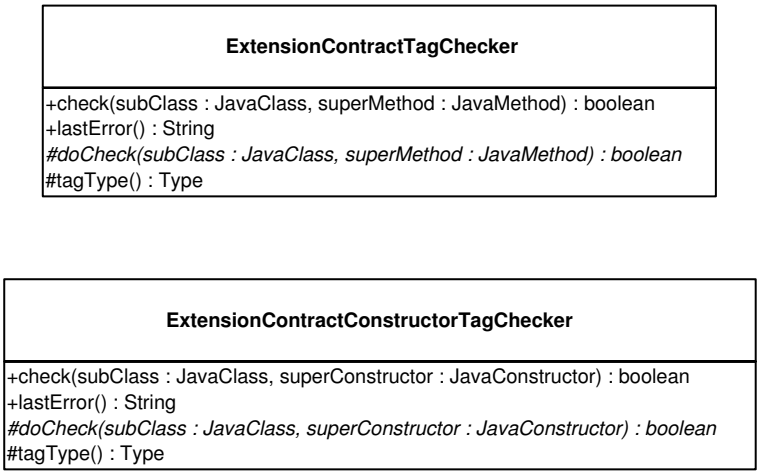

Abbildung 5.6: Die beiden Oberklassen für *Tag Checker*

Die beiden einzigen *public* Methoden sind die Methoden *check* und *lastError*. Die Methode *lastError* liefert in einem Fehlerfall den zuletzt gefundenen Fehler. Die Methode *check* prüft zuerst anhand des Parameters *superMethod* bzw. *superConstructor*, ob dieser *Tag Checker* die Unterklasse für diese Methode bzw. diesen Konstruktor überprüfen muß. Dazu wird geprüft, ob die der *Tag*-Typ, der von der Methode *type* geliefert wird, im Kommentar der *superMethod* bzw. des *superConstructor* enthalten ist. Falls dieser *Tag Checker* in Aktion treten muss, so wird die Methode *doCheck* aufgerufen, in der der eigentliche Test, ob die Unterklasse gemäß des *Extension Contract Tags* implementiert wurde, stattfindet.

Dieser Programablauf soll anhand der Klasse *OverrideTagChecker* verdeutlicht werden. In der Methode *check* dieser Klasse wird zunächst überprüft, ob für die Methode der Oberklasse das Tag *override* definiert wurde. Ist dies der Fall, wird die Methode *doCheck* aufgerufen. Diese prüft zunächst, ob die Methode in der Unterklasse auch vorhanden ist, also redefiniert wurde. Ist dies der Fall muss noch geprüft werden, ob die Methode überschrieben oder erweitert wurde. Dazu wird anhand der Methodenaufrufe aus der Methode der Unterklasse geprüft, ob darunter auch der Aufruf der Methode der Oberklasse ist. Ist dies der Fall, erfüllt die Unterklasse nicht die Bedingungen für das Tag *override* und es wird der Wert *false* zurückgeliefert. Gleiches gilt für den Fall, dass die Methode in der Unterklasse nicht redefiniert wurde und das entsprechende Tag *override* mit dem Zusatz *must* versehen wurde.

5.4 Der Extension Contract Checker

Die wesentliche Klasse des *Extension Contract Checkers*, die auch die einzelnen *Tag Checker* aufruft, bildet die gleichnamige Klasse *ExtensionContractChecker*. Sie enthält alle vorhandenen *Tag Checker* und *Constructor Tag Checker*, mit deren Hilfe der *Extension Contract* zu einer Klasse überprüft wird (vgl. Abbildung 5.7).

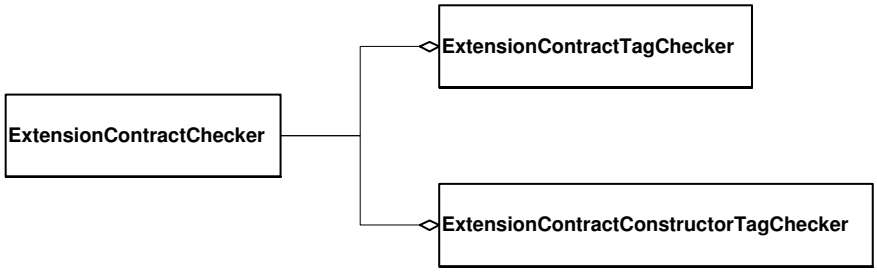

Abbildung 5.7: Klassendiagramm der Klasse ExtensionContractChecker

Die Schnittstelle dieser Klasse enthält nur zwei Methoden, wie die Abbildung 5.8 zeigt. Die Methode *checkFiles* erwartet als Parameter eine Java-Quelltext-Datei oder ein Verzeichnis, das selbst oder in Unterverzeichnissen Java-Quelltext-Dateien enthält. Die Methode check überprüft für eine einzelne Unterklasse, ob sie den *Extension Contract* der Oberklasse einhält.

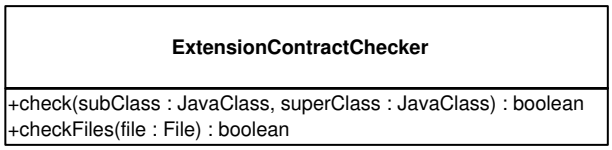

Abbildung 5.8: Die Klasse ExtensionContractChecker

Zur Laufzeit versucht die Methode *checkFiles*, die übergebene Java-Quelltext-Datei bzw. rekursiv alle Dateien aus einem Verzeichnis und dessen Unterverzeichnissen mit Hilfe des *Parsers* zu laden. Zu jeder Quelltext-Datei wird anhand der Oberklasse versucht, die entsprechende *Extension Contract*-Datei zu laden. Gelingt dies nicht, wird versucht, die Quelltext-Datei oder die *class*-Datei der Oberklasse zu laden. Dies geschieht, da in jedem Fall noch versucht wird, rekursiv die *Extension Contract*-Datei aller Oberklassen bis hin zur Klasse *Object* zu laden. Aus den geladenen *Extension Contract*-Dateien wird nach den in Kapitel 4.3 diskutierten Vererbungsregeln für *Extension Contracts* ein gesamter *ExtensionContract* für die gerade geladene Quelltext-Datei erzeugt. Dies geschieht, indem für die Überprüfung sämtliche sichtbaren Methoden, die also nicht in Unterklassen überschrieben wurden, aus den indirekten Oberklassen in die direkte

Oberklasse eingefügt werden. Mit dem vom *Parser* gelieferten Objekt und dem erzeugten Oberklassenobjekt wird nun die Methode *check* aufgerufen.

Die Methode *check* prüft für die gegebene Unter- und Oberklasse, ob die Unterklasse den *Extension Contract* der Oberklasse einhält. Ausgangspunkt für die Überprüfung sind die Konstruktoren und Methoden der Oberklasse und deren *inner classes*. Für jeden Konstruktor und jede Methode wird jeweils an allen vorhanden *Constructor Tag Checkern* bzw. *Tag Checkern* die Methode *check* aufgerufen. Sobald einer dieser Aufrufe den Wert *false* zurückliefert, ist der *Extension Contract* von der jeweiligen Unterklasse verletzt worden.

5.5 Erweiterungsmöglichkeiten

Der Ansatz der *Extension Contracts* läßt sich leicht um weitere Tags erweitern, solange diese statisch überprüfbar sind. Dazu muß zunächst in der Klasse *TagType* eine entsprechende Konstante eingefügt werden. Anschließend muß je nachdem, ob das Tag einem Konstruktor oder einer Methode zugeordnet ist, eine Unterklasse zu einer der Klassen *ExtensionContractTagChecker* oder *ExtensionContractConstructorTagChecker* programmiert werden. Dafür müssen die beiden Methoden *tagType* und *doCheck* überschrieben werden. Die Methode *tagType* liefert einfach als Ergebnis die neu angelegte Konstante der Klasse *TagType* zurück. Die entscheidende Arbeit besteht darin, die Methode *doCheck* zu implementieren. Hier muß anhand des Konstruktors bzw. der Methode der Oberklasse und der gesamten Unterklasse entschieden werden, ob die Unterklasse dem Tag entsprechend programmiert wurde. Falls dies der Fall ist, muss der Wert *true* zurückgeliefert werden, anderenfalls *falls*. Der letzte Schritt besteht darin, ein Objekt des neuen *Tag Checkers* im Konstruktor der Klasse *ExtensionContractChecker* zu erzeugen.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde gezeigt, dass viele Fehlerquellen beim Beerben von Klassen aus Rahmenwerken bestehen, die von vorhandenen Ansätzen nicht erkannt werden. Deshalb wurde in dieser Arbeit ein neuer Ansatz, die *Extension Contracts*, vorgestellt, mit dem die Fehlerquellen effektiv behoben werden. Um diesen Ansatz effizient bei der Entwicklung von Programmen einsetzen zu können, wurden Werkzeuge eingeführt, die sowohl die Rahmenwerks- als auch die Anwendungsentwickler unterstützen.

Gleichzeitig ermöglicht es der Ansatz der *Extension Contracts* auch dem Anwendungsentwickler nachträglich einen Vertrag zu einer bestehenden Rahmenwerksklasse zu definieren. Dadurch kann in einer Projektgruppe das Wissen, wie von bestimmten Klassen geerbt werden muss, gezielt weitergegeben werden, so dass neu einzuarbeitende Projektmitglieder nicht die schon gemachten Fehler wiederholen.

Weitere Entwicklungsmöglichkeiten für *Extension Contracts* Werkzeuge könnten sein:

- Extension Contracts als Eclipse-Plugin

  Die bisherige Implementierung der *Extension Contracts* ermöglicht es Anwendungsentwicklern, auf Knopfdruck die eigenen Quelltexte auf ihre Gültigkeit hin zu prüfen. Eine weitere Vereinfachung wäre die Integration der *Extension Contracts* als Plugin in die Eclipse-Umgebung. Dadurch könnten die Quelltextdateien automatisch beim Speichern geprüft werden.

- Extension Contract Editor

  Die *Extension Contract* Dateien können aufgrund ihrer XML-Struktur mit etlichen Programmen angezeigt und bearbeitet werden. Da diese Programme für die Bearbeitung allgemeiner XML-Dateien ausgelegt sind, unterstützen sie den Anwendungsentwickler nicht optimal bei der Interpretation und Erstellung der *Extension Contract*-Dateien. Ein Editor, der eine *class*-Datei als Eingabe bekommt, könnte den Anwendungsentwickler effektiv bei der Erstellung einer *Extension Contract*-Datei unterstützen.

Im konzeptionellen Umfeld sind sicherlich auch noch Erweiterungen möglich, die im Rahmen diser Arbeit nicht behandelt werden konnten. Hier zeigen die vorgestellten Ansätze aus dem akademischen Umfeld noch Richtungen auf, in die sich die *Extension Contracts* bewegen können.

- Weitere Extension Contract Tags

  Es stellt sich die Frage, ob noch weitere allgemeine Fehlerquellen mit Hilfe dieses Ansatzes behoben werden können.

- Extension Contracts für Rahmenwerksversionierung

  Extension Contracts bilden zur Zeit eine Beschreibung, wie zu einer gegebenen Implementierung einer Klasse Unterklassen gebildet werden können. Es stellt sich die Frage, wie sich diese Extension Contracts bei einer Veränderung der Implementierung der Oberklasse verhalten.

Die möglichen beschriebenen Erweiterungen sind eine weitere Stärke des Ansatzes der *Extension Contracts*. Er ist zur Zeit schon effektiv von den Entwicklern einsetzbar, die aktuellste Version ist über die *Extension Contract Homepage* (`http://www.tyzuk.de/ec/index.html`) zu erreichen. Die vorgeschlagenen Erweiterungen würden zu einer noch effizienteren Unterstützung der Entwickler führen.

Anhang A: Extension Contract Sprachdefinition

Extension Contract Tags

Im Folgenden werden die bisher implementierten Extension Contract Tags aufgeführt und ihre unterschiedlichen Verwendungsmöglichkeiten erläutert.

override

Mit dem Tag *override* wird angegeben, dass die gekennzeichnete Methode überschrieben werden soll, falls sie redefiniert wird. Das Tag hat folgende Syntax:

> *@override [must] [// Kommentar]* Diese Methode kann (muss) in Unterklassen überschrieben werden, die Implementierung der jeweiligen Oberklasse darf nicht aufgerufen werden.

Mit dem Tag override kann für eine Methode festgelegt werden, dass die gegebene Implementierung nicht aus Unterklassen heraus aufgerufen werden soll, falls diese Methode redefiniert wird. Dies ist z. B. dann sinnvoll, wenn die Klasse keine abstrakte Klasse sein soll und deshalb für diese Methode eine Dummy-Implementierung programmiert wurde. Eine weitere Einsatzmöglichkeit bietet sich bei leer implementierten Methoden, die von Unterklassen überschrieben werden können, wie z. B. die schon vorgestellte Methode *doUseMaterial(Thing material)*. Der Einsatz des Tags könnte dann wie folgt aussehen:

```
/**
 * @override // fuer spezifische Materialien entsprechend
 *   ueberschreiben}
 */
protected void doUseMaterial (Thing material)
{
  // this operation has to be redefined to act in a specific way
}
```

Für die Funktion einer Unterklasse, in der die Methode *doUseMaterial(Thing material)* redefiniert wird, ist es sicherlich unerheblich, ob die leere Implementation der Oberklasse

aufgerufen wird oder nicht. Allerdings würde dieses der Intention des Programmierers der Oberklasse entgegenlaufen und für einen Programmierer einer Unterklasse macht es auch nur Sinn, die Implementierung der Oberklasse aufzurufen, falls dadurch potenziell das Verhalten der Software verändert wird.

Eine weitere Einsatzvariante dieses Tags, nämlich mit der Option *must*, wirkt sicherlich zunächst seltsam. So bedeutet *override must* schließlich, dass eine so gekennzeichnete Methode in allen Unterklassen überschrieben werden soll, und zwar ohne die Implementierung der Oberklasse aufzurufen. Dies ist auf den ersten Blick das gleiche Verhalten, wie es bei abstrakten Methoden der Fall ist. Doch es gibt einen wichtigen Unterschied, denn wiederum führt eine abstrakte Methode dazu, dass die ganze Klasse abstrakt ist und somit keine Instanzen dieser Klasse erzeugt werden können. Mit Hilfe des Tags ist es aber möglich, das Überschreiben gewisser Methoden in einer Unterklasse von Klassen, die nicht abstrakt sind, zu erzwingen. Dadurch lassen sich auch zu solchen Klassen leicht Unit-Tests (vgl. [Oes99]) implementieren.

extend

Das Tag extend beschreibt, das die zugehörige Methode in Unterklassen erweitert werden muss, falls diese redefiniert wird. Die Syntax dieses Tags sieht wie folgt aus:

> *@extend [pre | post] [must] [// Kommentar]*
>
> Diese Methode kann (muss) in Unterklassen erweitert werden. Dabei soll die Implementierung der Unterklasse vor (pre) oder nach (post) der Implementierung der Oberklasse oder an beliebiger Stelle ausgeführt werden.

Für Methoden ohne Rückgabewert bedeutet *extend post*, dass diese Methode wie ein Konstruktor redefiniert werden soll, dass heißt, wenn diese Methode in einer Unterklasse redefiniert wird, soll als erstes die Implementierung der Oberklasse aufgerufen werden, z.B.:

```
/**
 * @extend post // hiernach Subwerkzeuge initialisieren
 */
protected void equip ()
{
  _isEquipped = true;
}
```

extend pre stellt das Gegenteil zu *extend post* dar und bedeutet dementsprechend, dass die Implementierung der Oberklasse als Letztes in der redefinierenden Methode aufgerufen werden soll. Diese beiden gegensätzliche Optionen lassen sich unter anderem bei Methoden gegensätzlicher Bedeutung einsetzen, z. B. bei Methoden wie *connect* und *disconnect*.

Auch das Tag *extend* kann mit der Option *must* versehen werden. Dies ist sinnvoll, da es einerseits eine sehr enge Verwandtschaft zum Tag *override* hat und dementsprechend auch ähnliche Eigenschaften haben soll, um einen intuitiven Umgang mit den Tags zu erleichtern. Andererseits wird bei den folgenden Tags deutlich werden, dass ein entsprechendes Verhalten auch durch andere Tags nachgebildet werden kann.

coupled_with

Wie in den Beispielen gesehen, kommt es häufig vor, dass Gruppen von Methoden zusammen redefiniert werden. Diese Abhängigkeit kann mit dem Tag *coupled_with* abgebildet werden:

@coupled_with method [, method] [// Kommentar]

Dieses Tag unterstützt sowohl den Entwickler der Unter- als auch den der Oberklasse. Für den Programmierer der Unterklasse gibt das Tag an, welche Methoden er zusätzlich redefinieren muss, falls er bestimmte Methoden redefiniert. Für den Programmierer der Oberklasse bietet sich der Vorteil, dass anhand der Tags geprüft werden kann, ob seine Kommentare konsistent sind. Die Benutzung dieses Tags sieht wie folgt aus:

```
/**
 * Notice: You have to redefine canCreateFromString(),
 * value(String) and isValid(String) always together
 * to ensure consistency.
 * @override
 * @coupled_with valueImpl(), isValidImpl()
 */
public boolean canCreateFromString () {
  return false;
}

/** [...]
 * @override
 * @coupled_with canCreateFromString(), isValid(String)
 */
protected DomainValue valueImpl (String s) {
  return null;
}

/** [...]
 * @override
 * @coupled_with canCreateFromString, valueImpl(String)
 */
protected boolean isValidImpl (String s) {
  return false;
}
```

In dem Beispiel sollen also die drei gegebenen Methoden in einer Unterklasse alle gemeinsam überschrieben werden, oder keine von ihnen. Das Beispiel zeigt gleichzeitig,

dass die *Extension Contract Tags* auch kombiniert verwendet werden können, indem zusätzlich definiert wird, dass die einzelnen Methoden gegebenenfalls überschrieben werden sollen.

Nun kann man sich vorstellen, dass es für eine der Methoden notwendig wäre, dass diese in Unterklassen überschrieben wird, zu ihr also *override must* definiert worden wäre. Dies würde automatisch dazu führen, dass auch die anderen beiden Methoden überschrieben werden müssten. Wenn jetzt eine der anderen Methoden nicht überschrieben sondern erweitert werden sollte, würde dies für diese Methode einem *extend must* gleichkommen.

redefine_with

Gilt die Abhängigkeit von Methoden nur in einer Richtung, so wird dies mit dem Tag redefine_with verdeutlicht:

> *@redefine_with method [, method] [// Kommentar]*
>
> Wenn diese Methode redefiniert wird, müssen die angegebenen Methoden auch redefiniert werden.

Dieses Tag könnte also eingesetzt werden, um die Abhängigkeit der Methode *hashCode* von der Methode *equals* zu beschreiben. Dies könnte wie folgt aussehen:

```
/**
 * @redefine_with hashcode() // vgl. Javadoc-Kommentar von Sun
 */
public boolean equals (Object object) {
  [...]
}
```

redefine_restricted

Das Tag *redefine_restricted* und das im Anschluss folgende Tag *calls* beziehen sich jeweils auf Konstruktoren, und geben an, welche Methoden aus dem gegebenen Konstruktor heraus aufgerufen werden:

> *@redefine_restricted method [, method] [// Kommentar]*
>
> Die angegebenen Methoden dürfen in Unterklassen redefiniert werden, dabei muss aber beachtet werden, dass diese polymorph aus dem Konstruktor der Oberklasse aufgerufen werden.

Als Methoden kommen in diesem Fall Methoden in Frage, die weder *private* noch *static* noch *final* sind. Der Entwickler der Klasse weist mit diesem Tag darauf hin, dass die angegebenen Methoden in Unterklassen nur mit Einschränkungen überschrieben werden dürfen. Dies liegt daran, dass die Methoden vom Konstruktor der Oberklasse aufgerufen werden, bevor die entsprechenden Membervariablen initialisiert wurden.

calls

Das Tag *calls* wird im gleichen Zusammenhang wie das eben vorgestellte Tag benutzt:

> *@calls method [, method] [// Kommentar]*
>
> Die angegebenen Methoden dürfen in Unterklassen nicht redefiniert werden, da diese polymorph aus dem Konstruktor der Oberklasse aufgerufen werden und sie nicht explizit zum Überschreiben freigegeben sind.

Auf den ersten Blick scheint dieses Tag überflüssig, denn wenn es verboten sein soll, die entsprechenden Methoden zu überschreiben, so können diese z. B. *final* definiert werden. In der Tat nimmt dieses Tag eine Sonderstellung ein, da es nicht für den Gebrauch durch den Entwickler der betreffenden Klasse vorgesehen ist. Vielmehr wird dieses Tag automatisch beim Erzeugen der *Extension Contract*-Datei vom *Extension Contract*-Generator (vgl. Kapitel 4.4) eingefügt.

Anhang B: Programminstallation und Benutzung

Installation

Um Extensioncontracts einsetzen zu können, müssen auf Ihrem Computer folgende Systemvoraussetzungen erfüllt sein:

Java ist installiert (Version 1.3 oder höher)

Die Umgebungsvariable JAVA HOME ist gesetzt.

Sind die Voraussetzungen bei Ihnen erfüllt, können Sie das Extension Contract Package installieren, indem Sie das heruntergeladene Archiv entpacken, z. B. nach `d:\`.

In den Systempfad nehmen Sie bitte folgendes auf: d: extensioncontract bin Als letzten muss noch folgende Umgebungsvariable gesetzt werden:

EC_HOME = d:\extensioncontract

Benutzung

Zu Ihren Sourcecode-Dateien können Sie wie folgt Extension Contracts erzeugen:

generate source dest [classpath]

z.B.:

generate d:\extensioncontract\source\de\extensioncontract\samples
d:\mycontract

Wenn Sie Ihre Sourcecode-Dateien im Hinblick auf vorhandene Extension Contracts überprüfen wollen, können Sie dieses mit folgendem Befehl tun:

check source [classpath]

z.B.:

check d:\extensioncontract\source\de\extensioncontract\samples
d:\contract;d:\extensioncontract\mycontract

Anhang C: Extension Contract DTD

```
<!-- Die Vererbungsschnittstelle wird fuer eine Klasse
     definiert -->

<!-- Eine Klasse enthält einen Javadoc-Kommentar und eine
     beliebige Anzahl von Methoden und/oder inneren Klassen
-->
<!ELEMENT class (comment?, (method|class)*)>
<!ATTLIST class
   scope (package-private|private|protected|public) #REQUIRED
   name CDATA #REQUIRED
   package CDATA #REQUIRED
>

<!-- Eine Methode kann eine Parameterliste haben und beliebig
     viele Javadoc-Tags -->
<!ELEMENT method (params?, comment?, methodcall*)>
<!ATTLIST method
   scope (package-private|private|protected|public) #REQUIRED
   type CDATA #REQUIRED
   name CDATA #REQUIRED
>

<!-- Die Parameterliste besteht aus unterschiedlichen Typen
-->
<!ELEMENT params (type+)>
<!ELEMENT type (#PCDATA)>

<!-- Ein Kommentar besteht aus beliebig vielen Javadoc-Tags
-->
<!ELEMENT comment (doctag+)>

<!-- Ein Javadoc-Tag ist von einem bestimmten Typ und kann
     beliebig viele Attribute haben -->
<!ELEMENT doctag (tagattribute*)>
<!ATTLIST doctag
```

```
   type (override|extend|coupled_with|redefine_with|
       redefine_restricted|calls) #REQUIRED
>

<!-- Ein Tagattribut kann ein beliebiger Text sein -->
<!ELEMENT tagattribute (#PCDATA)>

<!-- Ein Methodenaufruf besteht aus dem Methodennamen und der
     Klasse des zugehörigen Objektes -->
<!ELEMENT methodcall (params?)>
<!ATTLIST methodcall
    destination CDATA #REQUIRED
    methodname CDATA #REQUIRED
>
```

Literaturverzeichnis

[Bec99] BECK, KENT: *Extreme programming explained: embrace change*, Addison Wesley Longman, Inc., Reading, MA, 1999.

[Bec03] BECK, KENT: *Test-driven development: by example*, Addison-Wesley, Boston, MA, 2003.

[Blo01] Bloch, Joshua: *Effective Java, Programming Language Guide*, Addison Wesley, Boston, 2001.

[Bur03] Burn, Oliver: *Checkstyle*, `http://checkstyle.sourceforge.net`, Februar 2004.

[DeGa87] DeMichiel, L. G., Gabriel, R. P.: *The Common Lips Object System: An Overview*, Proc. ECOOP '87, Paris, France; in Lecture Notes in Computer Science 276, Springer-Verlag, 1987.

[J2SDK1.3] *JavaTM 2 Platform, Standard Edition, v 1.3.1 API Specification*, `http://java.sun.com/j2se/1.3/docs/api/index.html`, Februar 2004.

[J2SDK1.4] *JavaTM 2 Platform, Standard Edition, v 1.4.0 API Specification*, `http://java.sun.com/j2se/1.4/docs/api/index.html`, Februar 2004.

[JavaCC] *Java Compiler Compiler, v 3.2.* `https://javacc.dev.java.net`, Februar 2004

[JWAM03] *JWAM: Java Framework for the Tools and Materials Approach*, Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 2003. `http://www.jwam.de`, April 2003.

[Kee89] Keene, S. E.: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS* Addison-Wesley, Reading, MA, 1989.

[KiLa92] Kiczales, G.; Lamping, J.: *Issues in the Design and Specification of Class Libraries* Proceedings of OOPSLA 92, Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices Vol. 27, Nr. 10, S. 435-451, ACM Press, 1992.

[Lam93] Lamping, J.: *Typing the specialization interface*, Proceedings, OOPSLA93, Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices Vol. 28, Nr. 10, S. 201-215. ACM Press, Reading, MA, 1993.

[LBR99] Leavens, Gary T.; Baker, A.L.; Ruby, Clyde;: *JML: A Notation for Detailed Design*, `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmlkluwer.pdf`, 1999.

[Lin02] LINK, JOHANNES: *Unit tests mit Java: Der Test-first-Ansatz*. dpunkt-Verlag, Heidelberg, 2002.

[LSM97] Lucas, C; Steyaert, P; Mens, K.: *Managing Software Evolution through Reuse Contracts*, Techreport vub-prog-tr-97-01, Universität Brüssel, `http://prog.vub.ac.be/research/rcs`, 1997.

[Mey92] Meyer, B.: *Eiffel: the Language* Prentice-Hall, New York, 1992

[Mey97] Meyer, Bertrand: *Object-oriented software construction*, Prentice-Hall, New York, London, Second Edition, 1997.

[Par72] Parnas, David L.: *On the Criteria To Be Used in Decomposing Systems into Modules*, Communications of the ACM Vol. 15, Nr. 12, S. 1053 - 1058. ACM Press, Reading, MA, 1972.

[Par03] Terence Parr *Another Tool for Language Recognition*, v. 2.7.1. `Tool for Language Recognition`, 2003

[Oes99] Oestereich, Bernd: *Objektorientierte Softwareentwicklung, Analyse und Design mit der Unified Modeling Language* Oldenbourg Verlag, München, Wien, 4. aktual. Auflage 1. korrigierter Nachdruck, 1999

[RePo99] Rechenberger, Peter; Pomberger, Gustav: *Informatik Handbuch* Carl Hanser Verlag, München, Wien, 1997.

[RuLe00] Ruby, Clyde; Leavens, Gary T.: *Safely Creating Correct Subclasses without Seeing Superclass Code*, Proceedings, OOPSLA93, Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices Vol. 35, Nr. 10, S. 208-228. ACM Press, Reading, MA, 2000.

[SLMH96] Steyaert, P.; Lucas, C.; Mens, K.; D'Hondt, T.: *Reuse Contracts: Managing the Evolution of Reusable Assets*, Proceedings, OOPSLA96, Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices Vol. 31, Nr. 10, S. 268-285. ACM Press, Reading, MA, 1996.

[Sny86] Snyder, Alan: *Encapsulation and Inheritance in Object-Oriented Programming Languages*, Proceedings, OOPSLA93, Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM SIGPLAN Notices Vol. 21, Nr. 9, S. 38-45. ACM Press, Reading, MA, 1986. .

[Szy98] Szyperski, Clemens: *Component Software, Beyond Object-Oriented Programming*, Addison Wesley, 1998.

[Züllighoven98] Züllighoven, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*, dpunkt-Verlag, Heidelberg, 1998.

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig durchgeführt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Hamburg, den 29.02.2004

Rolf Tyzuk