Programming By Contract: Pre- und Postconditions in Java

Markus Völter, voelter@acm.org, www.voelter.de

Einleitung

Sowohl objektorientierte als auch komponentenbasierte Softwareentwicklung basiert auf dem Prinzip der Trennung von Verantwortlichkeiten. Eine Klasse oder Komponente (bzw. deren Instanzen) übernehmen für eine genau definierte Aufgabe die Verantwortung. Andere Objekte oder Komponenten (im Folgenden reden wir der Einfachheit halber nur noch von Objekten) delegieren die Ausführung von Teilen ihrer Aufgabe an Objekte, die für diese Aufgaben spezialisiert sind. Man spricht in diesem Zusammenhang von der Server- und der Clientklasse. Die Serverklasse bietet ihre Dienste über ein wohldefiniertes Interface an, die Clientklasse verwendet dieses Interface. Eine Klasse kann sowohl Client als auch Server (für eine andere Klasse) sein. Das Interface des Servers definiert dadurch einen Vertrag, genannt Client/Server-Vertrag.

Wenn man die üblichen Mitteln der Objektorientierung als "Werkzeugkasten" verwendet, wird ein derartiger Vertrag folgendermassen definiert: Eine Serverklasse bietet dem Client eine Menge von Operationen an. Jede dieser Operationen hat einen Namen und eine Liste von Parametern. Bei einigen Sprachen (wie z.B. Smalltalk) ist dies bereits alles. Statisch typisierte Sprachen wie Java oder C++ reichern diese Schnittstelle mit Typinformationen an. Sowohl die Operation an sich als auch alle Parameter haben einen Typ. Damit wird der Vertrag zwischen Server und ihren Klienten weiter verfeinert, weil der Client ein Stück genauer mitgeteilt bekommt, was der Server erwartet. Die meisten objektorientierten Sprachen und Systeme belassen es dabei. Aber lässt sich der Vertrag nicht noch genauer spezifizieren? Und ist dies überhaupt nötig? Die Antwort auf die zweite Frage ist eindeutig "ja", wie die folgenden Absätze verdeutlichen sollen.

Beispiele für zu ungenau spezifizierte Client/Server-Verträge

Oft gehen Implementierungen von Operationen von Voraussetzungen aus, die nicht im Interface spezifiziert sind weil sie nicht mittels Typinformationen beschreibbar sind. Eine übliche Annahme ist beispielsweise, dass eine Objektreferenz nicht *null* sein darf. Üblicherweise wird dies mittels entsprechender Sicherheitsabfragen realisiert:

```
class X {
  void doSomething( Y y ) throws Exception {
    if ( y == null ) throw (
       new Exception("y darf nicht null sein!") );
    //...
  }
}
```

Leider wird diese Annahme nicht in der Interface-Spezifikation deutlich, man verlässt sich also darauf, dass der Implementierer des Interfaces überprüft. Zum Beispiel mittels obiger Sicherheitsabfrage. Oder man erwartet, dass der Client dafür sorgt, dass `y` immer ungleich `null` ist. Leider gibt es keine formale Möglichkeit, dem Client(programmierer) dies mitzuteilen.

Ein anderes Beispiel: Eine Operation geht davon aus, dass das Objekt in einem bestimmten Zustand sein muss, wenn die Operation ausgeführt wird. Damit in Zusammenhang steht auch das Problem der Aufrufreihenfolge von Operationen. Oft erfordert das Aufrufen einer Operation, dass eine andere Operation vorher ausgeführt wurde, oder eine Operation darf nur einmal ausgeführt werden. All diese Aspekte lassen sich über eine Sicherheitsabfrage realisieren, gehen aber aus dem Interface nicht hervor.

Ein weiteres, typisches Problem ist die Semantik einer Operation, die üblicherweise durch den Namen der Operation deutlich wird, vor allem im Zusammenhang mit Unterklassen. Dazu folgendes Beispiel:

```
class Auto {
  protected int speed = 0;
  public void accelerate() throws Exception {
    speed++;
  }
}
```

Eine Unterklasse kann (und soll) eine Operation aber jederzeit überschreiben können, um unterklassenspezifisches Verhalten zu ermöglichen, wie das nächste (legale) Beispiel zeigt:

```
class LKW extends Auto {
  public void accelerate() throws Exception {
    speed++;
    if ( speed > 80 ) throw ( new Exception( "Zu schnell..." )
);
  }
}
```

Der folgende Fall ist aber problematisch, weil das Verhalten der Operation (verringern der Geschwindigkeit) der durch ihren Namen implizierten Bedeutung widerspricht, und sich Teile des Clients eventuell auf diese Bedeutung verlassen.

```
class AnotherAuto extends Auto {
  public void accelerate() throws Exception {
    speed--;
  }
}
```

Dabei gibt es zwei Betrachtungsweisen:

1. Wieso setzt der Client eine bestimmte Semantik voraus, obwohl ihm diese formal nicht zugesichert wurde? Man müsste also eine formale Spezifikation der Semantik definieren.

2. Der Client soll diese Semantik voraussetzen, weil sie der Programmierer der Oberklasse so vorgesehen hat. Wie kann man den Programmierer einer Unterklasse daran hindern, diese Semantik zu modifizieren? Auch hier müsste eine formale Spezifikation der Semantik vorliegen, an die auch alle Unterklassen gebunden sind.

Lösungsansätze

Der übliche Lösungsansatz für derartige Probleme ist, den durch das Interface definierten Vertrag durch sogenannte Constraints zu präzisieren. **Constraints** sind in der Regel boolesche Ausdrücke, die wahr sein müssen, wenn das System in einem validen Zustand sein soll.

Üblicherweise gibt es drei Arten von Constraints. Preconditions, Postconditions und Invarianten. **Invarianten** definieren Constraints, die zu jeder Zeit während der Existenz eines Objektes wahr sein müssen (genauer: immer dann, wenn der Zustand eines Objektes „öffentlich sichtbar" ist, d.h. während der Ausführung einer (atomaren) Operation dürfen Invarianten verletzt werden). **Preconditions** sind Bedingungen, die vor Beginn der Ausführung einer Operation gelten müssen. Sie werden Operationen zugeordnet. Das Gegenstück ist die **Postcondition**. Dies sind Constraints die nach der Ausführung einer Operation gelten müssen. Auch Postconditions werden pro Operation definiert.

Ziel dieses Artikels ist es, solche Constraints in Java verfügbar zu machen, ohne neue Sprachfeatures oder spezielle Precompiler einzuführen. Zunächst jedoch einige Beispiele, wo Constraints bereits erfolgreich eingesetzt werden.

Hoare Triples

C.A.R. Hoare hat den Ansatz der Pre- und Postconditions bereits sehr früh gewählt und damit den Begriff des *Hoare Triples* geprägt. Ein solches Tripel besteht immer aus einer Precondition, einem Statement und eine Postcondition. Ein Beispiel wäre

```
{p: x>5} /*Precondition*/
x:=x+1;  /*Statement*/
{q: x>0} /*Postcondition*/
```

Weitere Informationen zu Hoare Triples finden sich unter [HT].

Constraints in UML

Beginnen wir mit Analyse und Design. Auch beim dafür zuständigen Standard UML (siehe [UML]) hat man festgestellt, dass das reine, durch die grafische Notation definierte Interface oft nicht ausreichend genau ist. Auch mittels Interaktionsdiagrammen lassen sich nicht alle semantischen Anforderungen darstellen. Um zu vermeiden, dass zusätzliche Semantik als Prosatext hinzugefügt wird, wurde die Object Constraint Language definiert. Die OCL (siehe [OCL]) ist eine deklarative, halbformale Sprache die es u.a. erlaubt, boolesche Ausdrücke zu definieren, die Notationselementen eines Diagrams zugeordnet werden können.

Abbildung 1 zeigt beispielhaft ein Klassendiagramm. Es werden zwei Klassen definiert, *Car* und *Person*, wobei das Auto eine Referenz auf eine Person besitzt: den Fahrer des Autos. Es werden dann zwei Constraints definiert. Die rechte definiert eine Invariante für die Klasse *Car* die besagt, dass das Alter einer *Person* die einem *Auto* als Fahrer zugeordnet wird größer als 18 Jahre sein muss. Die linke Constraint definiert eine Postcondition für die Operation *accelerate()*: Es wird verlangt, dass nach dem Aufruf der Operation die Geschwindigkeit größer oder gleich ist als vorher.

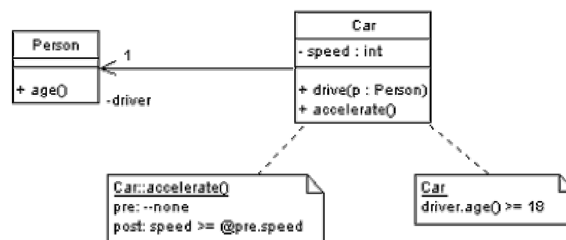

Abbildung 1: OCL-Constraints in UML: Die Abbildung zeigt eine Postcondition und eine Invariante.

OCL-Constraints sind ein mächtiges Werkzeug, welches in der Praxis leider viel zu selten Verwendung findet. Auch die Unterstützung in gängigen Tools ist nicht besonders ausgereift. Die führenden Modellierungswerkzeuge unterstützen OCL entweder gar nicht explizit, oder nur in Form von Freitextfeldern. Eine Überprüfung der Constraints auf syntaktische Korrektheit bezüglich des Modells oder gar eine semantische Überprüfung ist nicht implementiert. Auch bei der Codegenerierung wird auf Constraints keine Rücksicht genommen, obwohl sich OCL- Constraints relativ leicht in Programmcode wandeln lassen.

Constraints in Eiffel

Die Programmiersprache Eiffel realisiert einen Ansatz der Softwareentwicklung, der als *Programming By Contract* bezeichnet wird. Dabei wird davon ausgegangen, dass ein Dienstanbieter (Serverklasse) und ein Dienstverwender (Clientklasse) einen Vertrag abschliessen. Dieser Vertrag wird wie üblich als Interface und zusätzlich mittels deklarativer Constraints definiert (siehe [EIFFEL]).

Dazu bietet Eiffel Pre- und Postconditions sowie Invarianten direkt auf Sprachlevel. Sie können als Teil von abstrakten Datentypen (*class interfaces*) definiert werden. Eiffel verwendet dabei eine leicht andere Nomenklatur. Preconditions werden im *require* Teil eines Features (ein Feature ist ein Attribut oder eine Operation) angegeben, Postconditions im *ensure* Teil. Diese Bezeichnung verdeutlicht die folgende Semantik:

- Der Server **verlangt**, dass der Client bei Aufruf einer Operationen dafür sorgt, dass die Preconditions eingehalten werden. Der Server überprüft dies. Die

Implementierung der Operation **kann also davon ausgehen**, dass die in den Preconditions gemachten Annahmen zutreffen.

- Im Gegenzug **versichert** der Server dem Client, dass nach Ausführung der Operation die Postconditions gelten. Es ist Aufgabe der Implementierung der Operation, dass unter Annahme zutreffender Preconditions die Postconditions eingehalten werden. Der Client **kann also davon ausgehen**, dass nach Aufruf der Operation die Postconditions zutreffen.

Dies stellt den erweiterten Vertrag zwischen Client- und Serverklasse dar. Der Vertrag wird ausschliesslich bezüglich der Serverklasse spezifiziert, da im Voraus nie bekannt sein kann, von welchen Clients eine Serverklasse verwendet wird. Im Folgenden ein Beispiel, bei dem die folgenden Constraints definiert sind:

- die Operation *stop* verlangt, dass die Geschwindigkeit vorher größer 0 ist, dafür sichert sie zu, dass das Auto nach Ausführung der Operation steht (Geschwindigkeit gleich 0).
- *accelerate* sichert zu, dass die Geschwindigkeit nach Aufruf der Operation größer ist als vorher.
- Die Invariante verlangt, dass die Geschwindigkeit nie kleiner 0 sein darf.

Das folgende Codestück zeigt das Interface *AUTO* inklusive der beschriebenen Constraints:

```
class interface
  AUTO

  feature
    stop
      -- anhalten eines AUTOs
    require
      speed > 0
    ensure
      speed = 0

  feature
    accelerate( amount: INTEGER )
      -- AUTO um amount km/h beschleunigen
    require
      amount > 0
    ensure
      speed > old(speed)

  feature
    speed : INTEGER

  invariant
```

```
      invalidSpeed: speed >= 0

  end
```

Bei Fehlschlagen einer Constraint erzeugt Eiffels Laufzeitsystem eine Exception. Um dabei eine genauere Angabe der fehlgeschlagenen Constraint zu ermöglichen, ist es möglich, den Conditions und Invarianten Namen zu geben (wie bei der Invariante zu sehen: `invalidSpeed`).

Wie demonstriert, lässt sich mit den bisher beschriebenen Mechanismen der Vertrag zwischen der Serverklasse und den Clienten genauer spezifizieren als mit reinen Interfaces. Man beachte dabei, dass diese Constraints immer noch nichts über die Implementierung der Klasse aussagen, da sie ausschliesslich bezüglich des Interfaces spezifiziert sind.

Eiffel löst zusätzlich aber auch das Problem, dass Unterklassen die Semantik einer Operation ändern könnten. Sehen wir uns dazu das folgende Beispiel an:

```
class interface
  LKW
  inherit
    AUTO

  redefine
    accelerate
  end

  feature
    accelerate ( amount: INTEGER )
  ensure then
    speed < 80

end
```

Hier wird zusätzlich zugesichert, dass nach einer Beschleunigung die Geschwindigkeit nie grösser als 80 sein darf. Bei Vererbung sorgt Eiffel also zunächst dafür, dass bei einer Unterklasse alle Conditions der Oberklasse gelten. Zusätzlich können die Constraints aber ergänzt werden, und zwar in einer genau definierten Weise:

- Preconditions werden mit Oder-Semantik ergänzt, d.h. bei Aufruf der Operation der Unterklasse müssen entweder die Preconditions der Oberklasse, oder die neu hinzugefügten Preconditions der Unterklasse zutreffen. Die Operation der Unterklasse kann daher mit einer **weniger genau** definierten Umgebung auskommen, sie erledigt also **mehr** Dienste als die Oberklasse. Dies wird durch die explizite Formulierung als *require else*-Klausel verdeutlicht.

- Postconditions werden mit Und-Semantik ergänzt. Eine Unterklasse muss also mindestens die Zusicherungen der Oberklasse erfüllen. Zusätzlich können weitere Zusicherungen gemacht werden. Notiert wird dies durch eine *ensure then*-Klausel.

Damit ist sichergestellt, dass eine Unterklasse auch semantisch ein Untertyp der Oberklasse ist.

Constraints und Java

In Java werden deklarative Constraints leider nicht direkt unterstützt (deshalb zeigt dieser Artikel, wie es trotzdem geht). Wie bei vielen Tools oder Features kann man aber auch einfach von dem Konzept profitieren. Abgesehen von dem unten ausführlich beschriebenen Ansatz kann man das Konzept der Constraints auch mit Javas Bordmitteln realisieren – mit mehr oder weniger großen Einschränkungen:

- Man kann am Beginn von Methoden systematisch die Preconditions prüfen, bzw. am Ende die Postconditions und Invarianten. Damit steht der Constraint-Code aber direkt im Anwendungscode, eine Trennung der verschiedenen Aspekte liegt also nicht vor. Er kann auch nicht einfach an- und abgeschaltet werden. Auch das Problem der semantischen Korrektheit von in Unterklassen überschriebenen Operationen lässt sich damit nicht lösen, da die Operation der Unterklasse die Operation in der Basisklasse nicht aufrufen muss. Diese Lösung ist also unpraktikabel.

- Eine Lösung mittels Precompiler liegt nahe. Die Constraints können in einer separaten Datei oder in speziellen Kommentaren angegeben und per Precompiler eingefügt werden. Ein Precompiler lässt sich allerdings recht schwer in eine bestehende Entwicklungsinfrastruktur integrieren. Beispiele für dieses Vorgehen (beide verwenden spezielle Kommentare) finden sich unter [WEI] oder [IC].

- Mittels Aspektorientierung (z.B. AspectJ, [AJ]) können Constraints als Aspekte implementiert werden. Abgesehen von der Tatsache, das auch der AspectJ-Weaver den Quellcode modifiziert und insofern einen Precompiler darstellt, ist dies sicherlich eine praktikable Lösung, auf die aber hier nicht weiter eingegangen werden soll.

Übrigens gab es auch bereits einen Vorschlag, Java auf Sprachlevel um Constraints zu erweitern. Nachdem Java ja schon das Konzept der Interfaces explizit unterstützt, wären Constraints eine gute Ergänzung.

Eine Lösung in Java

Die im Folgenden vorgestellte Lösung adressiert alle in der Einleitung erläuterten Anforderungen, und kommt ohne zusätzliche Tools aus, es wird also kein Precompiler verwendet:

Es können Preconditions, Postconditions und Invarianten definiert werden.

- In Preconditions ist Zugriff auf die Parameter der Operation möglich, in den Postcondition auf den Rückgabewert.
- Bei Unterklassen können Preconditions gelockert, und Postconditions verschärft werden, wie im Eiffel-Beispiel erläutert.
- Die Überprüfung von Constraints kann ein- und ausgeschaltet werden, sodass in einer Release-Version keine Performancenachteile auftreten.
- Die Constraints werden separat vom eigentlichen Implementierungscode notiert, und durch den Proxy-Mechanismus von Java 1.3 zum eigentlichen Code "gelinkt" (s.u.)

Die Lösung hat aber auch Unschönheiten:

- Man kann Pre- und Postconditions sowie Invarianten nur für Interfaces definieren, nicht direkt für Klassen. Das heisst, man muss die Funktionalität zunächst als Interface spezifizieren, die dann von Implementierungsklassen implementiert wird. Dies passt zwar zu dem Konzept, dass Interfaces mittels Constraints genauer definiert werden, ist aber manchmal etwas unpraktisch.
- Die Erzeugung der Zielobjekte muss mittels einer Factory geschehen, die gegebenenfalls die Proxies erzeugt und um die Zielobjekte "herumlegt" (Wrapper).
- Die Performanz ist aufgrund der exzessiven Verwendung der Java Reflection API nicht besonders gut – in der Release-Version müssen die Constraints daher möglicherweise abgeschaltet werden.

Die Proxy-API

Die Gang-of-Four (siehe [GoF])beschreiben den Zweck des Proxymusters folgendermassen:

> *Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.*

Das bedeutet, dass ein Clientenobjekt welches mit einem Serverobjekt arbeiten will, in Wirklichkeit mit einem Proxy arbeitet, der dieselbe Schnittstelle wie das Serverobjekt bietet. Der Client merkt von diesem Versteckspiel nichts, d.h. aus Sicht des Clients ist nicht nur die Schnittstelle identisch, sondern auch das Verhalten. Allerdings steht es dem Proxy frei, **zusätzliche** Aufgaben zu erledigen. Dies ist die Stärke des Proxymusters. Abbildung 2 zeigt die Struktur des Proxymusters.

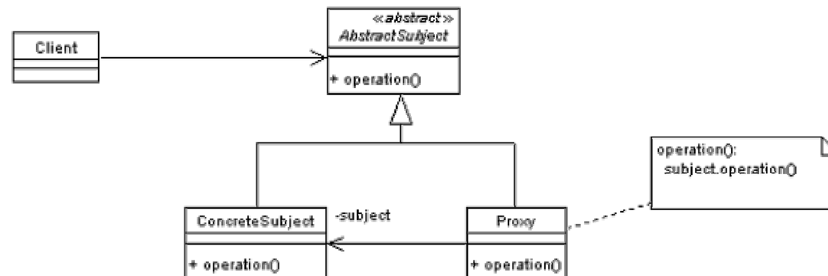

Abbildung 2: Struktur des Proxymusters

Der Client wird also gegen die Schnittstellen des *AbstractSubject* programmiert. Zur Laufzeit interagiert der Client mit dem *Proxy*, der den Aufruf gegebenenfalls an das *ConcreteSubject* weiterleitet. (Anmerkung: Das *AbstractSubject* kann in Java natürlich auch ein *Interface* sein.)

Beispielsweise kann der Proxy den Aufruf des Klienten in ein Byte-Array verpacken und dieses Byte-Array über ein Netzwerk transportieren. Auf der anderen Seite sitzt ein weiterer Proxy, der das Byte-Array einliest und beim Serverobjekt die gewünschte Operation aufruft. Man nennt ein derartiges Verhalten einen Remote-Proxy, es bildet die Basis für praktisch alle ORBs (Object Request Broker, Kommunikationsbackbone in verteilten, objektorientierten Systemem.)

Ein Proxy kann aber auch andere Aufgaben erledigen. Im Falle von Javas Proxy-API leitet der Proxy einen Methodenaufruf an ein anderes Objekt, einen sogenannten *InvocationHandler*, weiter. Dieser kann beliebige Aktionen durchführen, und dann gegebenenfalls die Operation beim eigentlichen Zielobjekt aufrufen. Das Besondere an der Proxy-API ist nun, dass das Java Runtime System einen solchen Proxy für Instanzen beliebiger Klassen automatisch generieren kann.

Prinzipielle Funktionsweise

Abbildung 3 zeigt ein Klassendiagramm des Systems. Das Java Laufzeitsystem erzeugt für das eigentliche Serverobjekt (*destObject*) automatisch einen Proxy, der alle Methodenaufrufe an den *PPCInvocationHandler* weiterleitet. Dieser kann aus den Namen der Interfaces die *destObject* implementiert durch anhängen von *Constraint* die Namen der zuständigen *ConstraintBase*-Subklassen herleiten. Diese enthält die Pre- und Postconditions sowie die Invarianten. Für ein Interface Vehicle würde der PPCInvocationHandler nach einer Klasse *VehicleConstraints* suchen. Der *PPCInvocationHandler* ist dann dafür verantwortlich, die Constraints zu überprüfen.

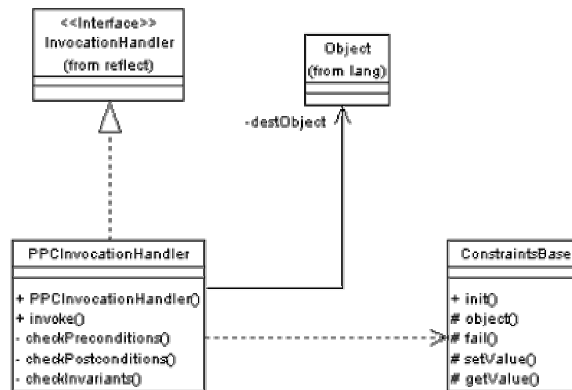

Abbildung 3: Klassendiagramm des Systems

Abbildung 4 zeigt den Ablauf eines Methodenaufrufs mit Constraint-Überprüfung. Der Client beabsichtigt, eine Operation auf dem Zielobjekt aufzurufen. Dieser Aufruf erreicht aber statt dessen den automatisch generierten Proxy. Dieser leitet den Aufruf an den dem Proxy zugeordneten *InvocationHandler* weiter, indem er dessen Operation *invoke()* aufruft. Diese Operation überprüft dann die Preconditions, ruft die eigentliche Operation auf dem Zielobjekt auf, und prüft dann alle Postconditions und die Invarianten der Klasse. Grob sieht die *invoke()*-Operation also folgendermassen aus:

```
public class PPCInvocationHandler implements InvocationHandler {
  private Object destObject;
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable {
    try {
      checkPreconditions( method, args );
      Object result = method.invoke( destObject, args );
      checkPostconditions( method, result );
      checkInvariants();
      return result;
    } catch ( Failed s ) {
      // handle...
    }
    return null;
  }
}
```

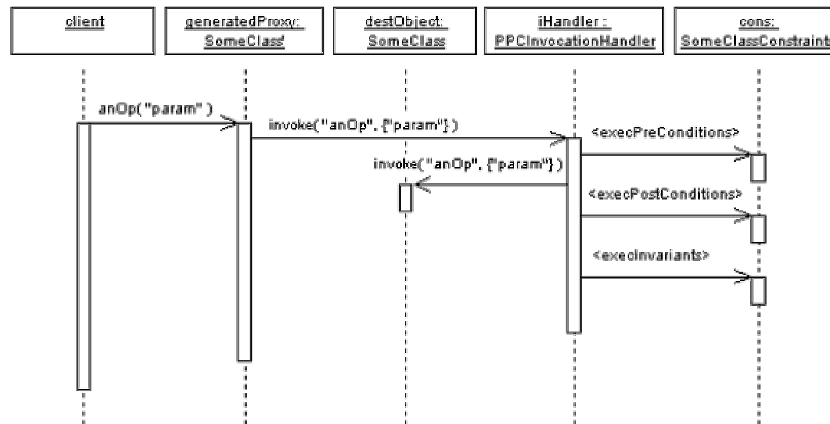

Abbildung 4: Aufruf einer Operation incl. Überprüfen der Constraints

Fragt sich nun noch, wie die Pre- und Postconditions sowie die Invarianten definiert werden. Dies ist sehr einfach. Wie bereits erwähnt, muss die Constraint-Klasse den Namen des Interfaces plus *Constraint* besitzen. Desweiteren muss die Klasse von *ConstraintBase* abgeleitet sein. Die Namenskonvention für die Constraints ist dann die folgende:

- Für jede Operation des Interfaces *{visibility} {RetType} op(params)* heisst die Precondition *public void pre_op(params)*. Die Parametersignatur ist identisch wie bei der Operation des Interfaces, jedoch ist der Rückgabetyp immer *void* und die Sichtbarkeit *public*.
- Für jede Operation des Interfaces *{visibility} {RetType} op(params)* heisst die Precondition *public void post_op(Object retVal)*. Auch hier ist die Sichtbarkeit immer *public*, als Parameter bekommt die Operation jedoch immer ein *Object*, welches den Rückgabewert der Operation des Interfaces enthält.
- Die Invariante der Klasse muss in der Operation *invariant()* überprüft werden.

Für alle drei Methodenarten gilt: Soll die Constraint fehlschlagen, so muss die Operation *fail(„why it failed...")* aufgerufen werden. Diese wirft dann eine *Failed* Exception. Deshalb müssen die Methoden müssen die *Failed* Exception werfen können (...*throws Failed*). Zugriff auf das Zielobjekt ist innerhalb der Constraintklasse mittels *object()* möglich. Beispiele folgen unten.

Damit ist die prinzipielle Funktionsweise beschrieben. Bleibt noch die Frage wie der Proxy erzeugt wird. Denn Java kann dies zwar automatisch, aber man muss Java schon dazu anweisen. Zur Kapselung des Erzeugungsprozesses wird eine Fabrik verwendet. Für jedes Interface muss eine Erzeugungsoperation definiert werden. Je nachdem, wie die Fabrik konfiguriert ist, erzeugt sie entweder nur ein Implementierungsobjekt für das Interface, oder, wie in Abbildung 5 dargestellt, ein Implementierungsobjekt mit vorgelagertem Proxy.

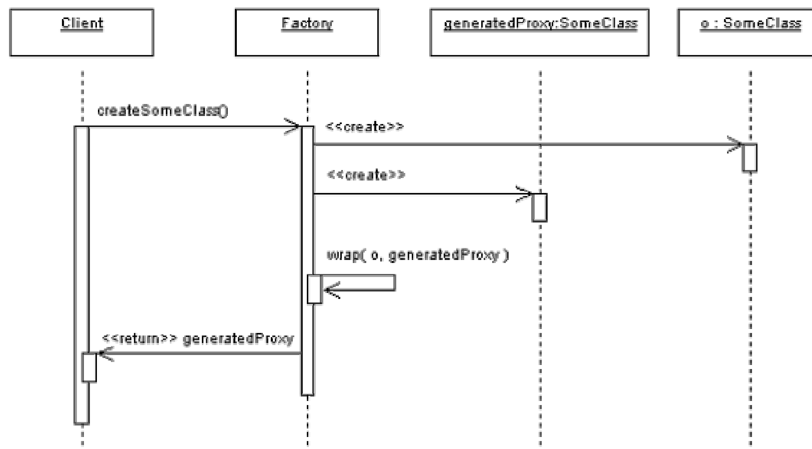

Abbildung 5: Erzeugung eines Objektes welches von einem Proxy umgeben ist.

Verwendung

Im folgenden Abschnitt soll die Verwendung der Pre- und Postconditions an Beispielen gezeigt werden. Beginnen wir mit einem Interface *Vehicle:*

```
public interface Vehicle {
  public void accelerate( int delta );      // beschl. um delta
  public int getSpeed();                    // Geschwindigkeit
  public void decelerate();                 // abbremsen
  public void stop();                       // anhalten
}
```

Als nächstes definieren wir eine entsprechende Implementierungsklasse. Die Implementierung ist hier so einfach wie möglich gehalten.

```
public class VehicleImpl implements Vehicle {

  private int speed = 0;

  public VehicleImpl() {
    speed = 0;
  }

  public void stop() {
    speed = 0;
  }

  public void accelerate( int delta ) {
    speed += delta;
  }

  public void decelerate() // ...
```

```
    public int getSpeed() // ...
}
```

Nun wollen wir für das Interface *Vehicle* die Constraints definieren. Man beachte, dass die Constraints wirklich für das Interface definiert werden, sie werden später automatisch für alle Implementierungsklassen überprüft. Wie oben beschrieben, heisst die Constraint-Klasse für das Interface *Vehicle* gemäß der Konvention *VehicleConstraints.*

```
public class VehicleConstraints extends ConstraintsBase {
    public void pre_accelerate( int delta ) throws Failed {
        if ( delta < 0 )
              fail( "Beschleunigung <0 nicht erlaubt!" );
    }
}
```

Die oben definierte Precondition wird der Operation *accelerate(int)* zugeordnet. Die Precondition überprüft, ob das Argument kleiner 0 ist. Wenn ja, soll die Precondition mit der Fehlermeldung *Beschleunigung <0 nicht erlaubt!* fehlschlagen.

Die folgenden Constraints stellen sicher, dass die Geschwindigkeit nach Aufruf von *decelerate()* kleiner ist als vor dem Aufruf. Natürlich gibt es in Java kein *old* Schlüsselwort wie in Eiffel, d.h. der Zugriff auf den Objektzustand vor der Ausführung der Operation ist nicht ohne weiteres möglich. Statt, was prinzipiell auch möglich wäre, das gesamte Objekt zu klonen und zwischenzuspeichern, wird der alte Geschwindigkeitswert von der Precondition mittels *setValue()* in einer Map abgelegt. In der Postcondition kann dieser Wert mit *getValue()* wieder ausgelesen werden.

```
public class VehicleConstraints extends ConstraintsBase {
    public void pre_decelerate() throws Failed {
        setValue( "speed",
              new Integer(((Vehicle)object()).getSpeed()) );
    }
    public void post_decelerate( Object retVal )
                                    throws Failed {
        int oldSpeed = ((Integer)getValue( "speed" )).
                            intValue();
        int newSpeed = ((Vehicle)object()).getSpeed();
        if ( oldSpeed <= newSpeed )
          fail( "Vehicle muss nach decelerate langs. sein! " );
    }
}
```

Als letztes Beispiel sei nun noch die Vererbung bei Interfaces demonstriert. Das Interface *Truck* erweitert *Vehicle*, wobei es keine weiteren Operationen hinzufügt. Auch die Implementierungsklasse *TruckImpl* erweitert *VehicleImpl* ohne deren Funktionalität zu verändern. Allerdings werden neue Constraints definiert, hier die Invariante, die besagt, dass ein LKW nie schneller als 80 fahren darf:

```
public class TruckConstraints extends ConstraintsBase {
    private Truck truck() { return (Truck)object(); }
```

```
        public void invariant() throws Failed {
            if ( truck().getSpeed() > 80 )
              fail( "LKW darf nicht schneller als 80 fahren!" );
        }
    }
```

Da das Interface *Truck* von *Vehicle* erbt, werden bei allen Implementierungen von *Truck* sowohl die Constraints in *TruckCostraints* als auch die in *VehicleConstraints* angewandt – mit der oben beschriebenen Semantik.

Nun fehlt noch ein einziger Schritt – die Factory um die Objekte samt Proxies zu erzeugen. Im Beispiel funktioniert das folgendermassen: Zunächst muss eine Factory definiert werden, diese muss von *ObjectFactory* erben.

```
public class VehicleFactory extends ObjectFactory {
  public VehicleFactory( boolean checkConstraints ) {
    super( checkConstraints );
  }
  public Vehicle createVehicle() {
    return (Vehicle)createProxy( new VehicleImpl() );
  }
  public Truck createTruck() {
    return (Truck)createProxy( new TruckImpl() );
  }
}
```

In dieser Factory wird für jede Klasse die instanziiert werden soll eine Operation angelegt. Diese Operation ruft *createProxy()* mit einem neuen Objekt des verlangten Typs auf. Die Factory erzeugt einen Proxy oder auch nicht – je nachdem, ob sie mit *true* oder *false* initialisiert wurde. Selbstverständlich können auch mehrere Erzeugungsoperationen mit unterschiedlicher Signatur angegeben werden, um Konstruktoren mit Initialisierungswerten versorgen zu können. Hier der Code für die *ObjectFactory*:

```
import java.lang.reflect.*;
import java.util.*;

public abstract class ObjectFactory
{
    boolean wrap = false;

    public ObjectFactory(boolean w) {
        wrap = w;
    }

    protected Object createProxy(Object o) {
        if ( wrap ) {
            Class c = o.getClass();
            return Proxy.newProxyInstance(
                    c.getClassLoader(), c.getInterfaces(),
                    new PPCInvocationHandler( o ) );
```

```
            } else return o;
        }
    }
```

Am Ziel

Zu guter Letzt wollen wir die Constraints dann natütrlich im Einsatz sehen. Die eigentliche Anwendung sieht aus wie gewohnt, der einzige Unterschied ist die Verwendung einer Factory um Instanzen der Klassen zu erzeugen, die Constraints besitzen sollen:

```
public class Test {
  public Test() {
    VehicleFactory f = new VehicleFactory( true );
    Vehicle v = f.createVehicle();
    v.accelerate(10);
    v.accelerate(-5);
    v.stop();
    Truck t = f.createTruck();
    // ...
  }
}
```

Das System überprüft nun – da die Factory mit *true* initialisiert wurde – die Constraints. Wenn eine Constraint nicht zutrifft, wird eine Exception geworfen, beispielsweise bei Übergabe eines negativen Betrages bei *accelerate()*:

```
constraints.Failed: negative beschleunigung ist nicht erlaubt!
  at constraints.ConstraintsBase.fail(ConstraintsBase.java:23)
  at constraints.ConstraintsBase.fail(ConstraintsBase.java:19)
  at constraints.VehicleConstraints.
                  pre_accelerate(VehicleConstraints.java:10)
```

Zusammenfassung

Wenn man mit den oben genannten Einschränkungen leben kann – Interfaces, Performance und die Factory – stellt die vorgestellte Technik eine nützliche Erweiterung des Vertrages zwischen Client- und Serverklassen dar. Sie betont vor allem die *Spezifikation* eines Interfaces und geht nicht auf Implementierungsdetails ein. Wie systematisches Testen auch, können die geschilderten Constraints die Qualität eines zu entwickelnden Systems deutlich verbessern.

Literatur

[AJ] Xerox PARC, *Aspect J Homepage*, http://aspectj.org

[EIFFEL] Thomas, Weedon; *Object-Oriented Programming in Eiffel*, Addison-Wesley 1997

[GOF] Gamma, Helm, Johnson, Vlissides, *Entwurfsmuster*, Addison-Wesley 1995

[HT] Middlebury College, *Learning Assertions*, http://www.middlebury.edu/classes/fall97/learn_assertions/hoaretriple.html

[IC] Reliable Systems, *iContract*, http://www.reliable-systems.com/tools/iContract/iContract.htm

[OCL] Object Management Group, *Object Constraint Language*, in [UML]

[UML] Object Management Group, *UML Specification*, http://www.omg.org

[WEI] Jim Weirich, *Design by Contract for Java* http://w3.one.net/~jweirich/java/javadbc/java-dbc.html