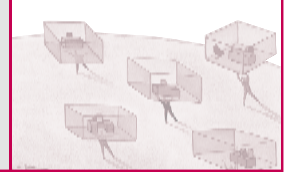


Des Kaisers neue Kleider

Aspektorientierte Softwareentwicklung mit CaesarJ

Iris Groher, Vaidas Gasiunas, Christa Schwanninger, Klaus Ostermann

In der Softwareentwickler-Community ist aspektorientierte Programmierung (AOP) mittlerweile weithin bekannt. Als praktische Herausforderungen, die man mit AOP bewältigen kann, gelten aber immer noch bloß die typischen Cross-Cutting Concerns wie Persistenz und Sicherheit, von Tracing ganz zu schweigen. Dabei bietet AOP viel mehr, wenn man über die etablierten Sprachen hinausgeht. Wir stellen Ihnen in diesem Artikel CaesarJ vor, einen Vertreter einer ganz neuen aspektorientierten Sprachgruppe. CaesarJ wurde speziell für die Entwicklung variabler Komponentenimplementierungen konzipiert.

Motivation

Die Entwicklung von Software-Systemfamilien bedeutet, für immer wieder ähnliche Systeme in ein und derselben Domäne einen Satz von Softwarekomponenten zu entwickeln, die so konfigurierbar sind, dass sie in mehreren Produkten verwendet werden können [SPL]. Einzelne Produkte unterscheiden sich in Anzahl und Variationen der enthaltenen Features, wobei ein Feature eine vom Benutzer als Mehrwert eingestufte Funktionalität ist, z. B. eine Rechtschreibprüfung in einem Editor.

Oft ist es nicht möglich ein Feature als modulares Add-on zu implementieren, da es sich häufig durch größere Teile der Implementierung eines Systems zieht, z. B. Benutzerverwaltung in einer Webapplikation. Eine große Herausforderung ist es, solche Features als optionale oder variable Systembestandteile zu implementieren. Diesen hohen Grad an Flexibilität hat bisher kaum eine Programmiersprache explizit unterstützt. Mit Mitteln der reinen Objektorientierung ist es notwendig, alle zu einem Feature gehörenden Klassen bzw. Teile von Klassen konsistent zu erweitern. Diese Menge an Klassenvarianten für ein konkretes Produkt richtig zusammenzufügen, ist oft nicht leicht. AO-Sprachen wie AspectJ [AspectJ] erlauben zwar, ein systemübergreifendes Feature modular zu implementieren, trotzdem bleibt diese Lösung ein „Patchen" einzelner bereits existierender Klassen. Außerdem wird für jede Variante ein neuer Aspekt notwendig.

Wir stellen hier CaesarJ [CaesarJ] vor, eine Java-basierte Programmiersprache für die Entwicklung von variablen Komponenten, die im Zusammenhang mit Software-Systemfamilien ein Feature ausmachen können. Eine CaesarJ-Komponente ist eine grobgranulare, in sich geschlossene Funktionseinheit, bestehend aus mehreren Klassen. Mit CaesarJ können Featureimplementierungen realisiert, in verschiedenen Ausprägungen angeboten und später zu einem fertigen Produkt kombiniert werden. Die Sprache verbindet die Konzepte der aspektorientierten Programmierung [AOP] mit Mechanismen für Verfeinerung und Komposition in „großem Stil" – das heißt, dass Kompositionsmechanismen wie Vererbung und Polymorphie auf Gruppen von Klassen, und nicht nur auf einzelne Klassen, angewendet werden. Zudem bietet CaesarJ Mechanismen, um neue Komponenten ohne Änderung des bestehenden Quelltextes in ein existierendes System zu integrieren. CaesarJ ist Java-kompatibel, Werkzeugunterstützung wird in Form eines Eclipse-Plug-Ins [Eclipse] angeboten.

Ein Beispiel

Stellen Sie sich eine Börsenapplikation vor, eine einfache Anwendung zur Verwaltung von Aktienkursen, die von Kunden abgefragt werden können. Abbildung 1 zeigt die Klassenstruktur der Applikation. Kunden stellen Anfragen (`StockInfoRequest`) an den `StockInformationBroker` und können dadurch die aktuellen Kurse bestimmter Aktien erfragen (`StockInfo`).

Die Aufgabe besteht nun darin, diese bereits vorhandene Applikation um das Feature *Pricing (Gebührenerhebung)* zu erweitern, welches flexible Mechanismen zur Verfügung stellen soll, um die Aktienkursabfrage kostenpflichtig zu machen.

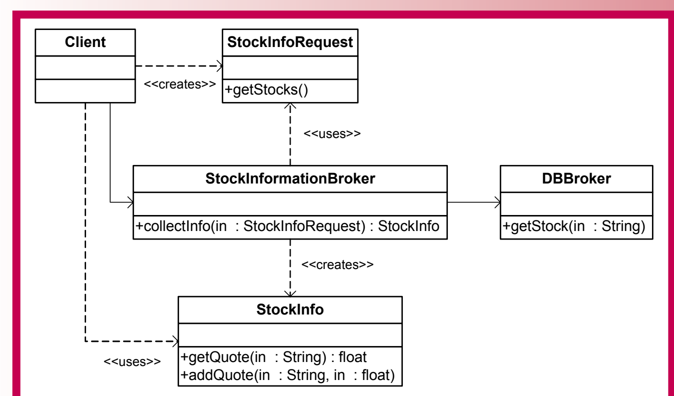


Abb.1: Klassenstruktur der Beispielapplikation

Unter flexibel verstehen wir hier, dass verschiedene Tarifmodelle für verschiedene Benutzer möglich sein sollen (wie etwa Bezahlung pro Anfrage oder pro Aktie) einschließlich diverser Rabattmodelle (zum Beispiel „3 zum Preis von 2").

Würden wir dieses Feature mit traditionellen Java-Mitteln implementieren, müssten wir die bestehende Applikation an vielen verschiedenen Stellen ändern. Wir müssten zumindest ein Attribut **balance** und eine Methode **charge()** zu **Client** hinzufügen, sowie einen Aufruf dieser Methode am Ende von **collectInfo()** der Klasse **StockInformationBroker**. Außerdem bräuchte die Methode **collectInfo()** für genau diesen Aufruf einen zusätzlichen Parameter vom Typ **Client**, um dem Kunden die Abfrage in Rechnung zu stellen.

In dieser Lösung ist das Pricing-Feature sehr eng mit der Implementierung der ursprünglichen Applikation gekoppelt. Daher muss jedes Mal, wenn das Tarifmodell geändert werden soll, auch die Applikation an vielen Stellen geändert werden. Noch schwieriger wird es, wenn verschiedene Tarifmodelle für verschiedene Benutzer zur Verfügung gestellt werden sollen – Nutzer des Systems könnten sich beispielsweise bei der Anmeldung individuell für ein Preismodell entscheiden. Für eine Familie von Börsenapplikationen bedeutet das, dass wir eine ganze Menge von Varianten der ursprünglichen Klassen für die verschiedenen Tarifmodelle implementieren müssten.

Im Folgenden zeigen wir, wie diese Anforderung (nicht-invasive Integration des Pricing-Features) mit den Konzepten von CaesarJ gelöst werden kann, ohne mit den soeben genannten Problemen kämpfen zu müssen.

Featuredefinition durch virtuelle Klassen

Eine Komponente in CaesarJ ist eine grobgranulare, in sich geschlossene Funktionseinheit, die mehrere Klassen beinhaltet, nämlich alle relevanten Abstraktionen, die für eine Komponente bzw. ein Feature notwendig sind. Wie sieht das genau aus? Alle Klassen, die eine Komponente ausmachen, sind in einer äußeren Klasse geschachtelt, ähnlich wie *inner classes* in Java. Allerdings ist dieses Schachtelungskonzept in CaesarJ sehr viel mächtiger: Innere Klassen können genau wie (virtuelle) Methoden in Ableitungen der umschließenden Klasse überschrieben bzw. erweitert werden – innere Klassen in CaesarJ sind daher *virtuelle Klassen*. Um CaesarJ-Klassen – sowohl innere als auch äußere Klassen – von konventionellen Java-Klassen zu unterscheiden, wird das Schlüsselwort **cclass** verwendet. Virtuelle Klassen können dazu benutzt werden, Gruppen von Klassen inkrementell zu verfeinern. Daher sind sie gut geeignet, um Features einer Produktlinie voneinander getrennt zu implementieren und zu variieren.

Zuerst wollen wir beschreiben, wie das Feature Pricing mit einer Applikation interagiert, also die Schnittstelle des Features zu einer Anwendung. Listing 1 zeigt genau so eine Schnittstellenbeschreibung (in CaesarJ *Collaboration Interface* genannt) für unser Beispiel. **PricingCI** enthält eine virtuelle Klasse für jede für das Feature Pricing relevante Abstraktion, in diesem Fall **Customer** und **Item**. Deren Methoden sind unterteilt in *provided* Methoden, also solche, die die Funktionalität der Komponente ausmachen, und *expected* Methoden, die zur Abfrage der kontextspezifischen Informationen dienen. Letztere sind notwendig, da bestimmte Teile der Pricing-Funktionalität anwendungsspezifisch sind, wie zum Beispiel der Preis einer Aktienabfrage oder Informationen (z. B. Name) über einen Kunden. Diese anwendungsspezifischen Teile werden durch abstrakte Methoden wie **Item.getBasePrice()** oder **Customer.getCustInfo()** modelliert. In der Schnittstellenbeschreibung unterscheiden sich *expected* und *provided* Methoden nicht von einander.

```
abstract cclass PricingCI {
  abstract public cclass Customer {
    /* provided */
    abstract public double getBalance();
    abstract public void charge(Item it);
    abstract public Bill createBill();
    /* expected */
    abstract public String getCustInfo();
  }
  abstract public cclass Item {
    /* provided */
    public double getPrice();
    public double getTax();
    public BillLine createBillLine();
    /* expected */
    public double getBasePrice();
    public String getItemDescr();
  }
}
```

Listing 1: Interface für das Pricing-Feature

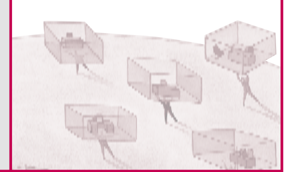
```
abstract cclass SimplePricing extends PricingCI {
  abstract public cclass Customer {
    private double balance;
    private List billLines;
    public double getBalance() { return balance; }
    public void charge(Item item) { balance -= item.getPrice();
      billlLines.add(item.createBillLine()); }
    public Bill createBill() {
      String header = "Bill for " + getCustInfo();
      ...
    }
  }
  abstract public cclass Item {
    public double getPrice() { return getBasePrice() + getTax(); }
    public BillLine createBillLine() {
      return new BillLine(getItemDescr(), getPrice(), getTax());
    }
    public double getTax() { ... }
  }
}
```

Listing 2: Einfache Implementierung des Pricing-Features

Die Komponente **SimplePricing** in Listing 2 implementiert eine mögliche Variante des Pricing-Features und zwar alle *provided* Methoden. Sie ist verantwortlich für die Verwaltung des Kontostandes, Kalkulation anfallender Steuern, Erzeugung von Rechnungen etc. Die *expected* Methoden können an dieser Stelle zwar verwendet werden, ihre Implementierung ist jedoch zu diesem Zeitpunkt noch nicht bekannt. Erst wenn die Komponente auf eine Applikation angewendet wird, ist die kontextspezifische Information verfügbar.

Komponentenintegration durch Bindings

Nun wollen wir unsere Komponente auf die Börsenapplikation anwenden. Die umschließenden Klassen, die wir bisher gesehen haben, sind abstrakt und können daher nicht instanziiert werden: Es fehlt noch die Implementierung der anwendungsspezifischen *expected* Methoden. Dieser Integrationscode ist komplett getrennt von den Implementierungen der Pricing-Komponente und ist daher ebenfalls wieder verwendbar. Solcher Integrationscode wird *Binding* genannt, weil er die Verbindung zwischen Features beschreibt.



```
abstract public cclass PerStockRequestBinding extends PricingCI {
  public cclass ClientCustomer extends Customer wraps Client {
    public String getCustInfo() {
      return wrappee.getClientName() + " " + wrappee.getClientId();
    }
  }
  public cclass StockItem extends Item wraps StockInfoRequest {
    public double getBasePrice() {
      return 5 + wrappee.getStocks().length * 0.2;
    }
    public String getItemDescr() {
      return "Stock req. "+ wrappee.getStocks().length;
    }
  }
  after(Client client, StockInfoRequest request) :
   (call(StockInfo StockInfoBroker.collectInfo(..)) &&
     this(c) && args(request)) {
     ClientCustomer(client).charge(StockItem(request));
  }
}
```

Listing 3: Integrationscode („Binding“) um Pricing mit der Stockbroker-Anwendung zu kombinieren

Ein Beispiel zeigt Listing 3. Die Klasse **PerStockRequestBinding** erweitert lediglich das Interface **PricingCI**, nicht aber eine konkrete Komponentenimplementierung wie **SimplePricing**. Betrachten wir eine mögliche Integration des Pricing-Features in die Stockbroker-Anwendung. Hierzu müssen wir zunächst mal die definierten Klassen zueinander in Beziehung setzen: Die Klasse **Client** hat zum Beispiel die Rolle des **Customer**, die Klasse **StockInfoRequest** die Rolle **Item**. Die Beziehungen zwischen den Klassen werden durch die **extends**- bzw. **wraps**-Klauseln spezifiziert. Die **extends**-Relation ist die bekannte Vererbungsrelation. Die **wraps**-Beziehung bedeutet, dass diese Klasse ein Wrapper für Instanzen der in der **wraps**-Klausel angegebenen Klasse ist, das entspricht genau der Idee des wohlbekannten Adapter-Musters [GOF]. Auf das adaptierte Objekt, welches bei der Objekterzeugung dem Wrapper-Objekt übergeben wird, kann über das Schlüsselwort **wrappee** zugegriffen werden.

Im Binding werden die *expected* Methoden implementiert. Es ist möglich, dass innerhalb von einem einzigen Binding eine Rolle (z. B. **Item**) an mehrere Anwendungsklassen gebunden wird, nämlich indem für jede Bindung eine eigene Subklasse definiert wird. So könnten wir die Rolle **Item** zusätzlich zur Klasse **StockInfoRequest** auch einer weiteren Klasse, z. B. **BondInfoRequest**, überstülpen. Die geschachtelten Klassen in Listing 3 implementieren nun die im jeweiligen Interface als *expected* markierten Methoden mit Hilfe der **wrappee**-Referenz.

Ein weiterer Bestandteil des Integrationscodes ist, dafür zu sorgen, dass die zu integrierende Komponente bei bestimmten Ereignissen benachrichtigt wird – in unserem Beispiel sollte bei einem Zugriff auf die Aktiendatenbank dem Kunden ein entsprechender Preis in Rechnung gestellt werden. Dies könnte durch Aufrufe der Pricing-Komponente innerhalb des Anwendungscodes geschehen, doch wir möchten dies nicht-invasiv integrieren, damit die Teile der Anwendung unabhängig voneinander bleiben. Für diesen Zweck benutzt CaesarJ die *Pointcut*- und *Advice*-Konstruktion aus der bekannten AOP-Sprache AspectJ [AspectJ]. Der **after**-Advice in Listing 3 reagiert auf Aufrufe der **StockInformationBroker.collectInfo()**-Methode und bindet die Information über den Aufrufer vom Typ **Client** und das Argument – in diesem Fall ein Objekt der Klasse **StockInfoRequest**. Im Advice selbst wird nun die **charge()**-Methode der Pricing-Komponente aufgerufen. Hierzu muss zunächst aus dem **Client**-Objekt **client** und dem **StockInfoRequest**-Objekt **request** ein **Customer**- bzw. **Item**-Objekt gemacht werden. Dafür bietet CaesarJ eine Art typsicheren Cast, der für jedes Objekt sein zugehöriges Wrapper-Objekt liefert.

Featurekombination durch Klassenkomposition und Deployment

Um eine funktionsfähige Pricing-Komponente zu erhalten, muss ein Binding mit einer Komponentenimplementierung kombiniert werden, da jeder Seite allein die *expected* bzw. *provided* Methoden der anderen Seite fehlen. In unserem Beispiel können wir mit der folgenden Deklaration **SimplePricing** und **PerStockRequestBinding** miteinander kombinieren:

```
cclass SimplePricingPerStockRequest
  extends SimplePricing & PerStockRequestBinding { }
```

Hier deklarieren wir eine Klasse, die von den beiden anderen angegebenen Klassen erbt. CaesarJ unterstützt eine einfache Form der Mehrfachvererbung, wobei die Vererbungshierarchie linearisiert wird, um die typischen Probleme von Mehrfachvererbung zu vermeiden. Diese Regeln sollen hier nicht im Vordergrund stehen; wichtiger ist, dass sich die Klassenkomposition mit dem &-Operator rekursiv in allen geschachtelten Klassen fortsetzt, d. h. durch die Komposition der umschließenden Klassen werden unterschiedliche Erweiterungen derselben virtuellen Klassen ebenfalls miteinander kombiniert. Hierdurch wird erreicht, dass sich die *expected* und *provided* Facetten der virtuellen Klassen wie gewünscht miteinander kombinieren. Dieser Kombinationsvorgang wird in Abbildung 2 illustriert.

SimplePricingPerStockRequest ist eine konkrete Klasse, die ganz gewöhnlich instanziiert werden kann. Um tatsächlich eingebunden zu werden, müssen solche Objekte explizit mit Hilfe eines Deployment-Mechanismus aktiviert werden und können auch zur Laufzeit wieder deaktiviert werden. Es gibt eine Reihe unterschiedlicher Deployment-Mechanismen in CaesarJ, von denen hier nur einer gezeigt wird. Das Feature Pricing wird für den Block innerhalb der Klammern nach dem Schlüsselwort **deploy** aktiviert:

```
final SimplePricingPerStockRequest pricing =
  new SimplePricingPerStockRequest();
deploy (pricing) {
  StockInfoRequest request = new StockInfoRequest(stockList);
```

➔

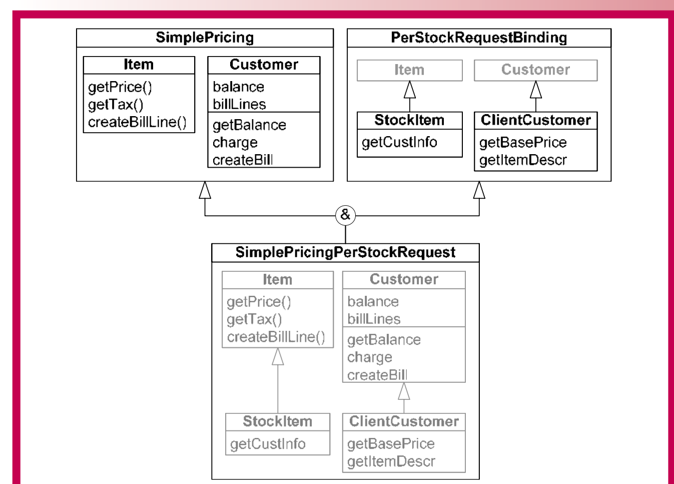


Abb. 2: Kombination von Komponentenimplementierung und Binding

```
    StockInfo si =
      StockInformationBroker.getInstance().collectInfo(request);
    // Advice ist während der Ausführung dieses Blocks aktiviert
  }
```

Erweiterbarkeit der Produktlinie

Wir haben gesehen, dass es möglich ist, eine Komponente wie Pricing nicht-invasiv zu einer bestehenden Anwendung hinzuzufügen. Wir haben auch gesehen, dass Komponenten unabhängig von ihrem Anwendungskontext implementiert werden können. Unsere Pricing-Komponente ist ohne weiteres in einer ganz anderen Anwendung wieder verwendbar, es muss nur ein neues Binding implementiert und mit einer der bestehenden Komponentenimplementierungen kombiniert werden. Es ist auch leicht möglich, Varianten einer bestehenden Komponente zu bilden.

Betrachten wir als Beispiel eine Variante der SimplePricing-Komponente. Ähnlich wie bei virtuellen Methoden ermöglichen virtuelle Klassen es, solche Erweiterungen inkrementell zu spezifizieren, indem nur die für die Änderung relevanten virtuellen Klassen erweitert werden, siehe Listing 4.

```
abstract cclass DiscountPricing extends SimplePricing {
  abstract public cclass Customer {
    protected int discountState = 4;
    public void charge(Item item) {
      if (discountState == 0) discountState = 4;
      else {
        super.charge(item); discountState--;
      }
    }
  }
}
```

Listing 4: Definition einer Variante der Pricing-Komponente

Es ist kein zusätzlicher Code erforderlich, um diese Variante in die Stockbroker-Anwendung zu integrieren. Wir können das bestehende Binding wieder verwenden und einfach mit der erweiterten Pricing-Komponente kombinieren:

```
cclass DiscountPricingPerStockRequest
  extends DiscountPricing & PerStockRequestBinding { }
```

Fazit

Aspektorientierte Programmierung wie in AspectJ ermöglicht es, den Code für bestimmte Cross-Cutting Concerns an einer Stelle zu lokalisieren. CaesarJ bietet die Möglichkeit, Aspekte nicht nur zu lokalisieren, sondern auch wieder verwendbar und erweiterbar zu machen, und ist somit eine sehr gute Plattform für Produktlinienarchitekturen. CaesarJ unterstützt durch seine elegante Kapselung von Variabilität die einfache Konfiguration vollständiger Produkte aus einzelnen Komponenten.

Weitere Informationen, Tutorials, Beispiele, Werkzeuge und natürlich der CaesarJ-Compiler selbst können auf der Webseite [CaesarJ] heruntergeladen werden.

Literatur und Links

[AOP] G. Kiczales u.a., Aspect-Oriented Programming, ECOOP 1997, LNCS Vol. 1241, Springer, 1997
[AspectJ] AspectJ Webseite, `http://www.eclipse.org/aspectj/`
[CaesarJ] CaesarJ Webseite, `http://www.caesarj.org`
[Eclipse] Eclipse Webseite, `http://www.eclipse.org`
[GOF] E. Gamma u.a., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
[SPL] P. Clements u.a., Software Product Lines: Practices and Patterns, Addison-Wesley, 2001

Iris Groher ist Doktorandin im Fachbereich Informatik an der TU Darmstadt. Ihre Arbeit wird von der Corporate Technology Abteilung der Siemens AG unterstützt. Iris Groher beschäftigt sich mit aspektorientierter Softwareentwicklung im Bereich von Software-Produktlinien.
E-Mail: groher@informatik.tu-darmstadt.de.

Vaidas Gasiunas ist wissenschaftlicher Mitarbeiter an der TU Darmstadt. Er beschäftigt sich mit Sprachmechanismen für eine bessere Modularisierung von Querschnittsaspekten und ist einer der Hauptentwickler von CaesarJ.
E-Mail: gasiunas@informatik.tu-darmstadt.de.

Christa Schwanninger ist Senior Research Scientist bei der Corporate Technology der Siemens AG in München. Sie beschäftigt sich mit Softwarearchitekturen, Entwurfsmustern und aspektorientierter Softwareentwicklung (AOSD). Seit beinahe drei Jahren leitet sie ein Forschungsprojekt, welches das Potenzial von AOSD für die Softwareentwicklung in Siemens evaluiert und für den Transfer von reifen AO-Technologien in die Praxis sorgt.
E-Mail: christa.schwanninger@siemens.com.

Klaus Ostermann ist Juniorprofessor im Fachbereich Informatik der TU Darmstadt. Er beschäftigt sich in seiner Forschung mit Konzepten zur Organisation und Modularisierung von großen Softwaresystemen.
E-Mail: klaus@caesarj.org.