SpringContracts

Design by Contract

Vertragsgrundlagen

Vertragsgestaltung

Beispiele

@Precondition

Analyse & Design

Framerules

commands

Queries

Motivation

Kollaboration von Akteuren

Klient nutzt Dienste eines Anbieters

Wie kann sich Klient auf die korrekte Abwicklung des Dienstes verlassen ?

Wie kann sich Anbieter auf die korrekte Inanspruchnahme
seines Dienstes verlassen ?

Auf welcher semantischen Grundlage interagieren Objekte miteinander ?

Semantik

... ausreichend durch Bezeichnung der Dienstleistung ?

(Klassennamen, Methodennamen, Argumentenbezeichner)

... ausreichend durch Einschränkung auf Typen ?

(Argumenttypen, Typ des Rückgabetyps)

... ausreichend durch Nutzungsbeispiele ?

(Unit-Tests)

Vertrag

Beschreibt die Dienstleistung des Anbieters

- Zusicherungen

Beschreibt die Gegenleistungen des Dienstenutzers

- (Vor-) Bedingungen

Beide Seiten müssen dem Vertrag zustimmen ...

... damit eine 'Geschäftsbeziehung' zustandekommt

... damit sich beide Seiten bei der Ausführung des Geschäfts vertrauen

Vorbedingungen

(vor Aufruf und Ausführung einer Methode)

specifies the circumstances under wich it is valid to call a feature

- beziehen sich auf aktuellen Zustand der Klasse (Anbieter)
- beziehen sich auf aktuellen Zustand der übergebenen Parameter (Klient)

```
@Precondition(
     condition =" not empty arg1
                 and
                 this.size < this.capacity" )

public void push( Object element );
```

Nachbedingungen

(nach Aufruf und Ausführung einer Methode)

specifies the effect of calling a feature

- beziehen sich auf den aktuellen Zustand der Klasse (Anbieter)
- beziehen sich auf Zustand der Klasse vor dem Methodenaufruf ('old')
- beziehen sich auf Zustand des Rückgabewertes
- beziehen sich auf aktuellen / alten Zustand der übergebenen Parameter

```
@Postcondition(
     condition = "  this.size == old:this.size + 1
                              and
                              this.top == arg1" )

public void push( Object element );
```

Invarianten

(gelten über den gesamten Lebenszeitraum des Anbieters)

specifies unchanging properties of an object of a class

- beziehen sich auf aktuellen Zustand der Klasse

```
@Invariant(
     condition = "  this.size >= 0
                              and
                              this.size <=
this.capacity" )

public interface S {...}
```

Contracs are Spec's n Check's

Spec's ...

- Contracts specify the expected behaviour

- Contracts document the 'WHAT' without the need of knowing 'HOW'

-> the 'documentation' accurately describes what the code actually does

... n' Checks

- Contracts can be checked at runtime

-> help test and debug the implementation

-> fail fast

Design by Contract für Java

- Annotierung der Vertragselemente über Annotations
(@Precondition, @Postcondition, @Invariant)

```
@Precondition(  condition = "not empty arg1" )

@Postcondition(
  message      = " Element hinzufuegen: {arg1}",
  condition = " this.size == old:this.size + 1 " +
                    " and  this.top == arg1" )

public void push( Object element );
```

- Formulierung der Conditions mittels Expression Language + Extensions

Beispiel:

```
Interface Stack{
     public void push( Object element );
     ...
}
```

Nachbedingungen eines erfolgreichen 'push' ???

- Formulierung der Conditions mittels Expression Language + Extensions

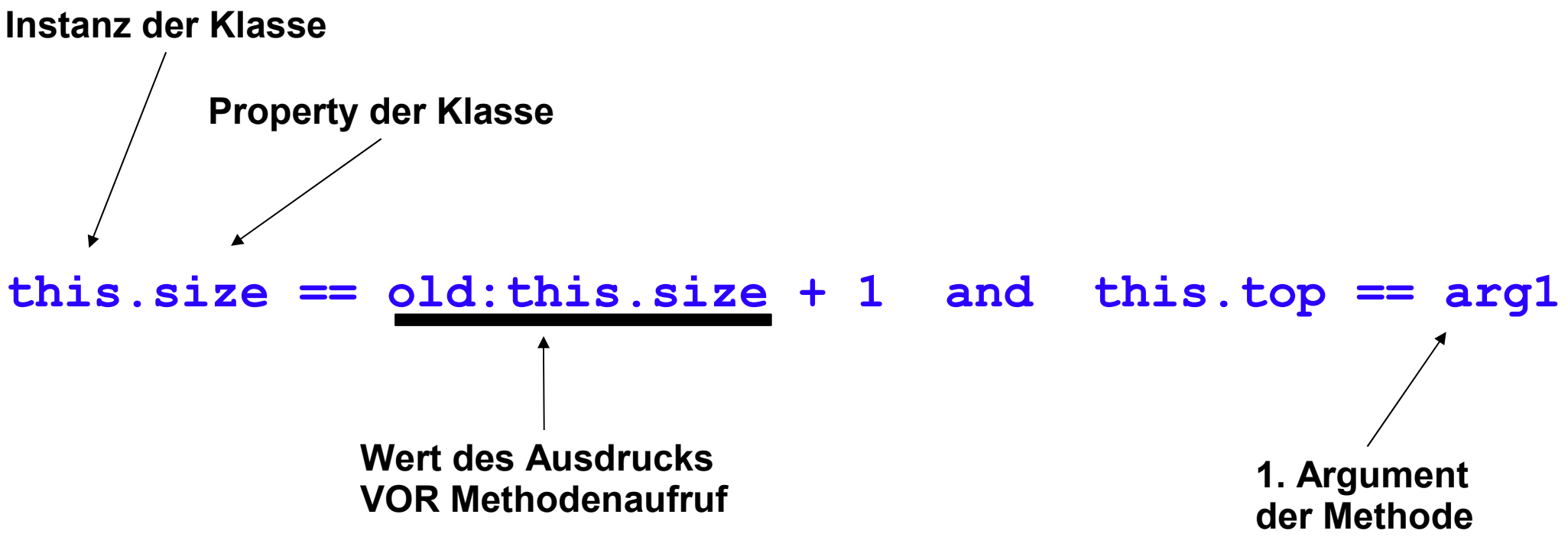
Beispiel:

```
Interface Stack{
     public void push( Object element );
     ...
}
```

- Anzahl der Elemente des Stacks erhöht sich um 1

- gepushtes Element ist oberstes Element des Stacks

- Formulierung der Conditions mittels Expression Language + Extensions



Beispiel:

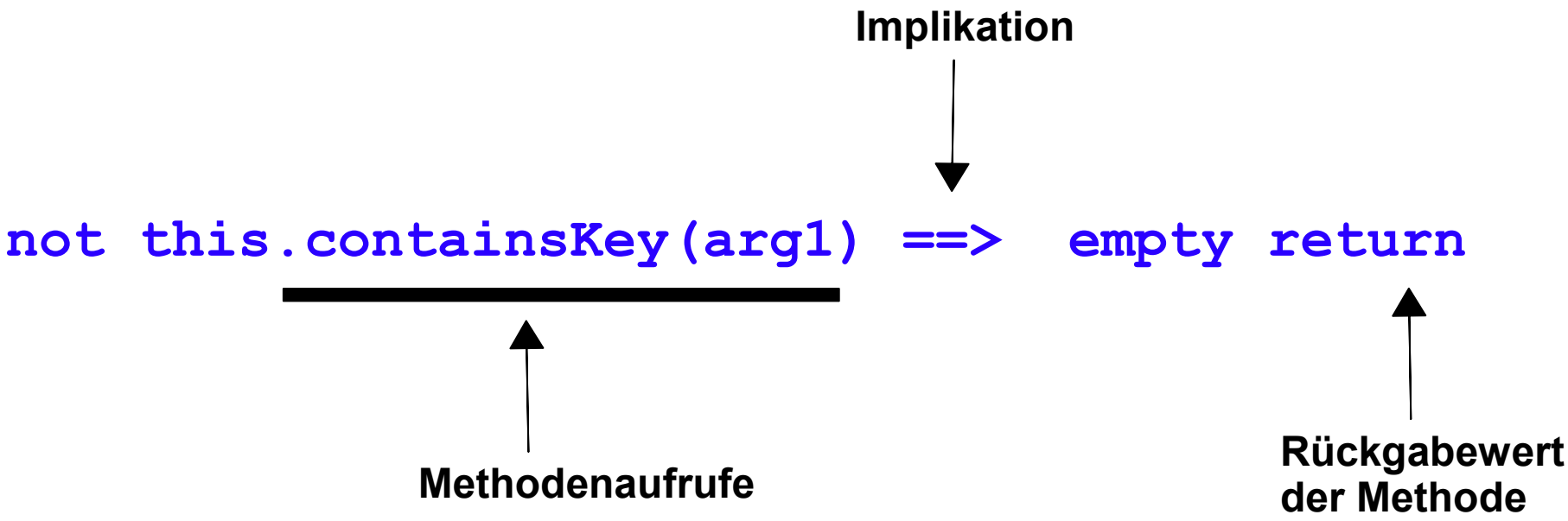
```
Interface Map{
     public Object get( Object key );
     ...
}
```

Nachbedingung get() :

WENN Map kein Key-Value-Paar mit dem übergebenen Key enthält (`public boolean containsKey( Object key )`)

DANN liefert Map **null**

SpringContracts – Conditions Forts.



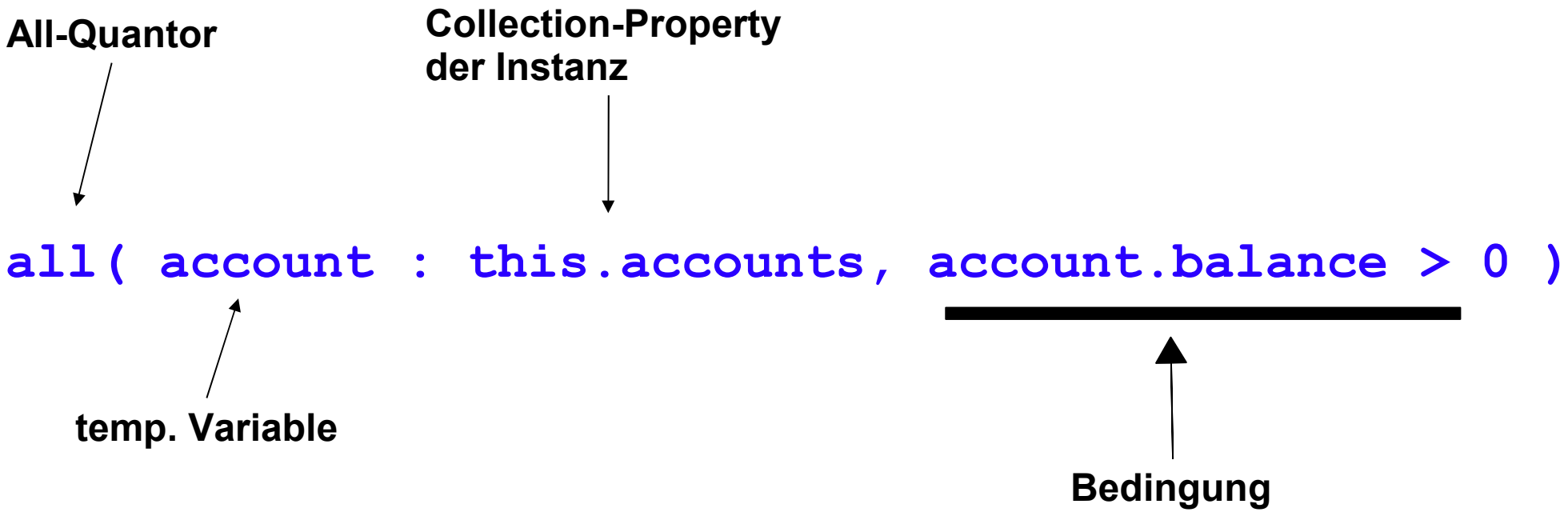
Beispiel:

```
Interface Account{
     public int getBalance;
     ...
}

Interface Bank{
     public List<Account> getAccounts();
     ...
}
```

Invariante Bank:

Alle Konten der Bank weisen jederzeit einen positiven Saldo auf

SpringContracts – Conditions Forts.

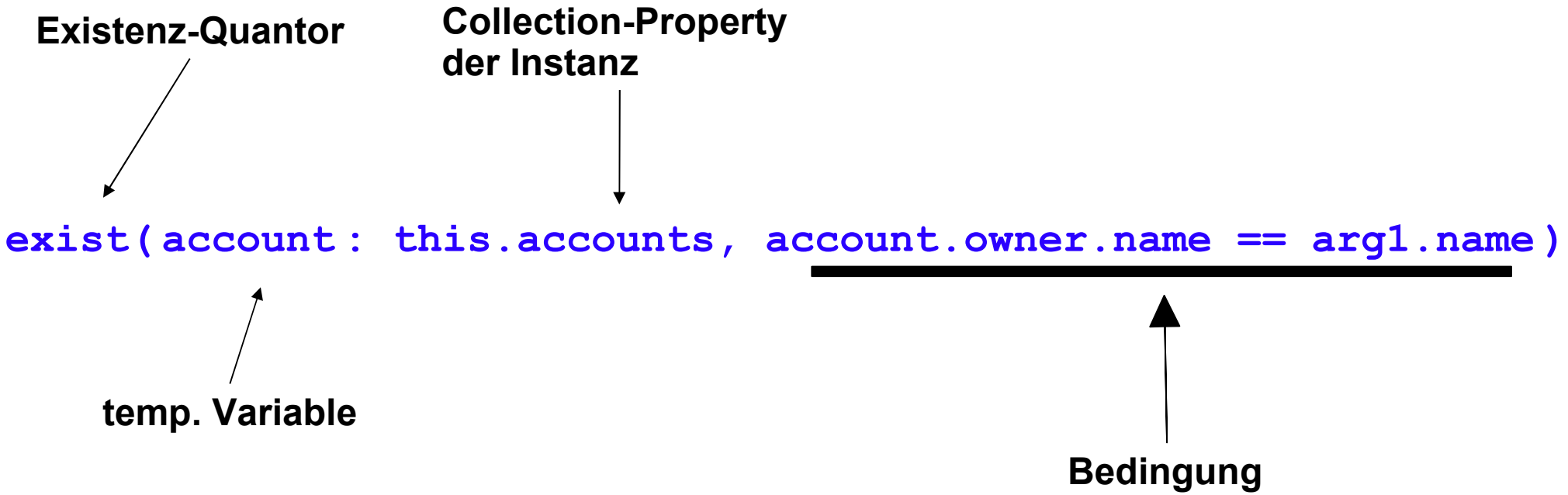


```
Interface Account{
     public Customer getOwner();
     ...
}

Interface Customer{
     public String getName();
     ...
}

Interface Bank{
     public ... getStatementSummary( Customer customer );
     ...
}
```

Precondition getStatementSummary():

Die Bank führt mindestens ein Konto des Kunden

SpringContracts – Conditions Forts.



SpringContracts – Symbolische Namensbindung

bindArgs = “arg1 = customer“

Bindung des 1. Arguments an den symbolischen Namen 'customer'

Referenzierung des 1. Arguments über dessen symbolischen Namen

Existenz-Quantor

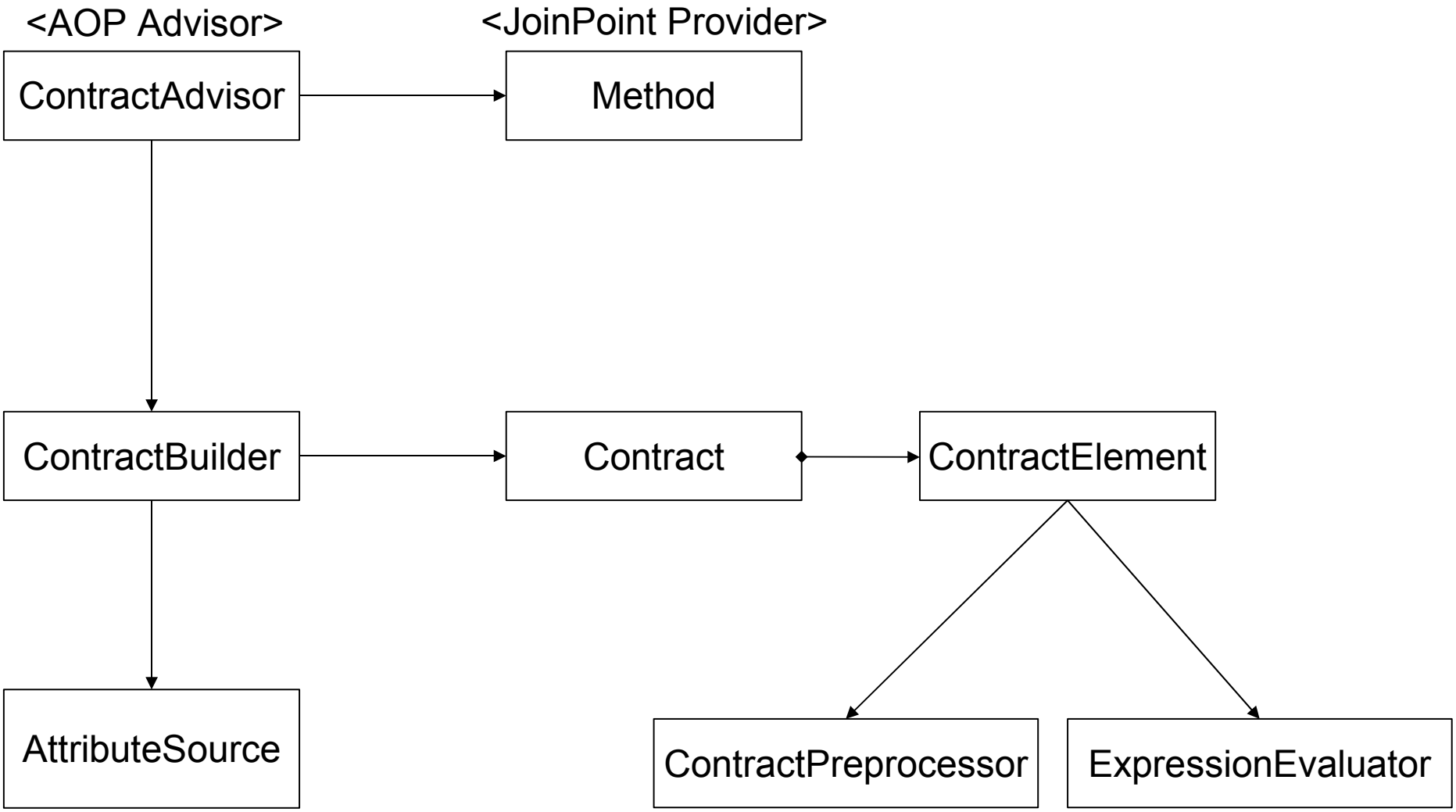
Collection-Property der Instanz

```
exist(account : this.accounts,account.owner.name == customer.name)
```

temp. Variable

Bedingung

Überblick Large-Scale Design SpringContracts



Card Game:

Es gibt 52 verschiedene Karten (Card)

Es gibt ein Spielfeld (Grid) mit einer festen Anzahl an Kartenpositionen (Position)

Regeln:

Jede Karte darf genau einmal auf dem Spielfeld platziert werden

Jede Karte darf beliebig oft auf dem Spielfeld verschoben werden

Jede Karte darf nur auf eine freie Position platziert bzw. verschoben werden

Eine platzierte Karte darf nicht mehr vom Spielfeld zurückgenommen werden

Das Spielfeld kann nicht mehr Karten aufnehmen als an Kartenpositionen (Kapazität) zur Verfügung stehen

Card

```
@invariant(
      condition = "this.number >= 1 and this.number <= 52" )

class Card{

      private  int number;

      public Card( int number ){
            this.number = number;
      }

      public int getNumber(){
            return number;
      }
}
```

Position

```
@invariant(
        condition = "this.placeNumber > 0" )

class Position{

        private  int placeNumber;

        public Position( int placeNumber ){
                this.placeNumber = placeNumber;
        }

        public int getPlaceNumber(){
                return placeNumber;
        }
        ...
}
```

Separate features into different categories:

Separate Commands and Queries

Queries:

- return a result but do not change the visible properties of the object
- provide a conceptional model of objects of the model

Commands

- might change the object, but do not return a result

Separate basic Queries from derived Queries

derived Queries can be specified in the terms of basic Queries

```
interface Grid{

        // Commands

        public void place( Card card, Position position );

        public void move( Position from, Position to );


        // Queries

        public int getCapacity();

        public int cardsOnGrit();

        public Card cardAt( Position position );

        public List<Position> getAllPositions();

}
```

Derived Queries Forts.

```
public boolean isAssigned( Position position );
```

-> cardAtPosition(position)

```
public boolean isFull();
```

-> getAllPositions() + cardAtPosition(position)

-> oder getCapacity() + getCardsOnGrit()

```
public boolean isAllreadyOnGrid( Card card );
```

-> getAllPositions() + cardAtPosition(position)

Writing appropriate Preconditions:

For every Query and Command decide on a suitable Precondition

constrain when clients may call queries and commands

Add physical constraints where appropriate

typically these will constraints that variables should not be void

```
public void place( Card card, Position position );
```

Argumente müssen gesetzt sein (physical constraint):

```
all( arg : args,  not empty arg )
```

Jede Karte darf genau einmal auf dem Spielfeld platziert werden :

```
not this.isAllreadyOnGrid( card )
```

Die Karte muss auf einer freien Position platziert werden :

```
not this.isAssigned( position )
```

Die Position muss innerhalb des gültigen Spielfelds liegen :

```
position.placeNumber <= this.capacity
```

```
@Precondition(
        bindArgs = " arg1 = card,  arg2 = position",
        condition = "all( arg : args,  not empty arg )
                                and
                                not this.isAllreadyOnGrid( card )
                                and
                                not this.isAssigned( position )
                                and
                                position.placeNumber <= this.capacity"
        message = "place card {card.number} on pos
{position.placeNumber}" )

public void place( Card card, Position position );
```

```
public void move( Position from, Position to );
```

? ? ?

```
public void move( Position from, Position to );
```

Auf der Ausgangsposition 'from' muss eine Karte platziert sein

```
this.isAssigned( from )
```

Die Karte muss auf eine freie Zielposition 'to' verschoben werden :

```
not this.isAssigned( to )
```

Die Zielposition 'to' muss innerhalb des gültigen Spielfelds liegen :

```
to.placeNumber <= this.capacity
```

Writing coherent Postconditions:

For each derived Query, write a Postcondition that specifies what result will be returned in terms of one ore more basic queries

-> then if we know the values of the basic queries, we also know the values of the derived queries

```
public boolean isAssigned( Position position );
```

Auf der Position ist eine Karte platziert :

```
        return == not empty this.cardAt( position )
```

```
public boolean isFull();
```

? ? ?

```
public boolean isAssigned( Position position );
```

Auf der Position ist eine Karte platziert :

```
return == not empty this.cardAt( position )
```

```
public boolean isFull();
```

Auf allen Positionen ist schon eine Karte platziert :

```
return == all( position : this.allPositions, not empty this.cardAt( position ) )
```

alt:

```
return == ( this.cardsOnGrit == this.capacity() )
```

```
public boolean isAllreadyOnGrid( Card card );
```

? ? ?

```
public boolean isAssigned( Position position );
```

Auf der Position ist eine Karte platziert :

```
        return == not empty this.cardAt( position )
```

```
public boolean isFull();
```

Auf allen Positionen ist schon eine Karte platziert :

```
        return == all( position : this.allPositions, not empty this.cardAt(
position ) )
```

```
public boolean isAllreadyOnGrid( Card card );
```

Es gibt eine Position, auf welcher die Karte platziert ist :

```
        return == exist( position : this.allPositions, this.cardAt( position ) ==
card )
```

Writing coherent Postconditions Forts:

For each Command write a Postcondition that specifies the value of every basic Query

-> taken together with principle 3 this means that we now know the total visible effect of each Command

```
public void place( Card card, Position position );
```

Position ist nach Platzierung der Karte belegt

```
not empty this.cardAt( position )
```

besser: die platzierte Karte liegt auf der Position

```
this.cardAt( position )    == card
```

Es liegt eine weitere Karte auf dem Spielfeld

```
this.cardOnGrit == old:this.cardsOnGrit + 1
```

Design Prinzipien - Postconditions

```
public void move( Position from, Position to );
```

? ? ?

Design Prinzipien - Postconditions

```
public void move( Position from, Position to );
```

Ausgangsposition 'from' ist nicht mehr belegt

```
empty this.cardAt( from )
```

Zielposition 'to' ist mit der verschobenen Karte belegt

```
this.cardAt( to ) == card
```

```
public void move( Position from, Position to );
```

Anzahl der Karten auf dem Spielfeld bleibt unverändert

```
this.cardOnGrit == old:this.cardsOnGrit
```

Frame-Rules

- beschreiben, was sich **NICHT** ändert

-> Beschreibung der begrenzten Auswirkung von Commands

-> Beschreibung der Nicht-Antastbarkeit von Argumenten

Writing Invariants:

Write Invariants to define unchanging properties of objects

concentrate on properties that help reader build on appropriate conceptual model of the abstraction that the class embodies

Constrain attributes using an invariant

when a derived query is implemented as an attribute it can be constrained to be consistend with other queries by an assertion in the class's invariant section

Design Prinzipien - Invariants

Die Anzahl der Karten überschreitet nie die Kapazität des Spielfelds

this.cardsOnGrid >= 0 and this.cardsOnGrit <= this.capacity

Alle Karten auf dem Spielfeld sind paarweise verschieden

? ? ?

Die Anzahl der Karten überschreitet nie die Kapazität des Spielfelds

```
this.cardsOnGrid >= 0    and   this.cardsOnGrit <= this.capacity
```

Alle Karten auf dem Spielfeld sind paarweise verschieden

```
not exist( position : this.positionen,
        all( otherPosition : this.positionen,
                (
                        not empty this.cardAt( position )
                        and
                        not empty this.cardAt( otherPosition )
                        and
                        position.placeNumber !=
otherPosition.placeNumber
                )
                ==>
                this.cardAt( position ) == this.cardAt( otherPosition )
        )
)
```

Redefinition

- common: Redefinition of a feature that is inherited from a superclass

- new : Redefinition of the Contract of that feature

Beispiel:

```
public interface Courier{

        @Precondition( condition = “package.weight < 3“ )

        @Postcondition( condition = “this.deliveryTime <= 5“ )

        public void subscribe( Package package );

        ...
}
```

You can weaken the precondition

```
public interface BikeCourier extends Courier{

        @Precondition( condition = “package.weight < 8“ )

        ...

        public void subscribe( Package package );

        ...
}
```

this.@Precondition OR super(s).@Precondition

You can strengthen the postcondition

```
public interface BikeCourier extends Courier{

        ...

        @Postcondition( condition = “this.deliveryTime <= 3“ )

        public void subscribe( Package package );

        ...
}
```

this.@Postcondition AND super(s).@Postcondition

Online – Auktion

Bei einem Online-Auktion-Shop können Artikel registriert und anschliessend versteigert werden

Eine Online-Auktion besitzt eine Menge von Kategorien, in welche die Artikel eingeteilt werden

Ein Artikel kann in bis zu drei Kategorien bei der Online-Auktion registriert werden

Ein Artikel kann nach der Registrierung (in mind. einer Kategorie) genau einmal zur Versteigerung gestartet werden

Bis zum Auktionsstart darf ein Artikel auch wieder aus der Online-Auktion entfernt werden (und damit aus allen derzeit registrierten Kategorien).

Nach dem Auktionsstart darf der Artikel nicht mehr weiteren Katgorien zugeordnet werden

Jeder Artikel hat einen Startpreis, welcher bis zum Auktionsstart verändert werden darf

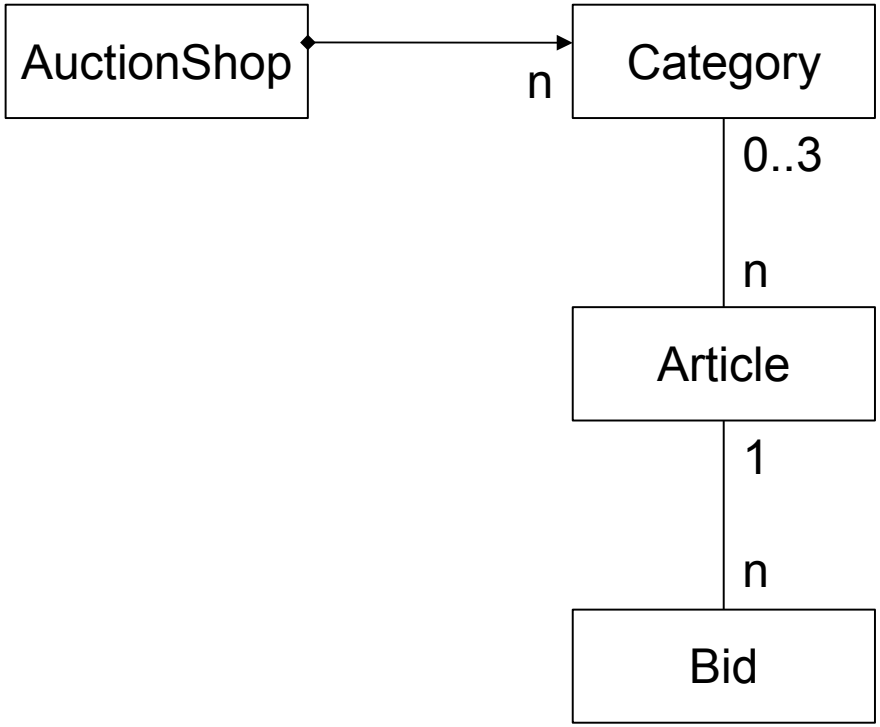
Für einen Artikel können nach Auktionsstart bis zum Auktionssende Gebote abgegeben werden

Jeder Artikel besitzt eine Gebotshistorie

Ein Gebot wird nicht berücksichtigt, wenn es niedriger als das derzeit höchste Gebot bzw. Startpreis ist.

Für jeden Artikel kann die Versteigerung genau einmal geschlossen werden**.**

Analyse - Typ-Modell



BID

Angebotshöhe muss grösser 0 sein

```
@Invariant( condition = “this.amount > 0“ )
public class Bid{

        int amount;

        @Precondition( condition = “arg1 > 0“ )
        public Bid( int amount ){
                this.amount = amount;
        }
}
```

Article (Queries)

```
public class Article{

        public int getInitialPrice(){...}

        public List<Bid> getBids(){...}

        public Bid getHighestBid(){...}

        public int countBids(){...}

        public boolean isAuctionOpened(){...}

        public boolean isAuctionClosed(){...}
        ...
}
```

Article (Queries Forts.)

public Bid getHighestBid(){...}

Wert aller anderen Gebote ist kleiner als das gelieferte höchste Gebote

? **?** **?**

Article (Queries Forts.)

public Bid getHighestBid(){...}

Wert aller anderen Gebote ist kleiner als das gelieferte höchste Gebote

```
@Postcondition(
        condition = "all( bid : this.bids,
                                ( bid != return ) ==> ( bid.amount <
return.amount ) )" )
public Bid getHighestBid(){...}
```

Article (Invarianten)

Alle Gebote liegen über dem Startpreis

```
@Invariant(
        condition = “all( bid : this.bids,
                                bid.amount > this.initialPrice“ )
public class Article{
        ...
}
```

Article (Invarianten)

Es gibt keine zwei Gebote mit dem gleichen Gebotswert

```
not exist( bid : this.bids,
        all( otherBid : this.bids,
                bid != otherBid  ==>  bid.amount != otherBid.amount
        )
)
```

Article (Commands)

public void raise(Bid newBid)

Vorbedingungen

Neues Gebot muss höher sein als Startwert bzw. alle anderen bereits abgegebenen Gebote

```
newBid > this.initialPrice
and
all( existingBid : this.bids, newBid.amount > existingBid.amount )
```

Article (Commands)

public void raise(Bid newBid)

Vorbedingungen

Auktion muss schon geöffnet sein und darf noch nicht geschlossen sein

this.auctionOpened and not this.auctionClosed

Article (Commands)

public void raise(Bid newBid)

Nachbedingungen

Neues Gebot ist das höchste Gebot
Anzahl der abgegebenen Gebote hat sich um 1 erhöht

this.highestBid == newBid and this.countBids() == old:this.countBids() + 1

Article (Commands)

protected void startAuction() **// darf nur über AuctionShop gestartet werden // wg. Regel 'mind. in einer Category registriert'**

Vorbedingung

Auction ist noch nicht eröffnet

not this.auctionOpened

Nachbedingung

Auction ist geöffnet

this.auctionOpened

Article (Commands)

public void closeAuction()

Vorbedingung

Auction ist schon geöffnet, aber noch nicht geschlossen

this.auctionOpened and not this.auctionClosed

Nachbedingung

Auction '*ist nicht mehr geöffnet*', dafür aber geschlossen

not this.auctionOpened and this.auctionClosed

Category

```
@Invariant(
        condition = “all( article : this.Articles,
                                this.offersArticle( article ) == true )“ )
interface Category{

        public List<Article> getArticles();

@Postcondition(
        condition = “return == true
                            ==>
                            exist( existingArticle : this.articles,
                                existingArticle == searchedArticle )“ )
        public boolean offersArticle( Article searchedArticle );
}
```

AuctionShop (Queries + Invariant)

```
@Invariant(
        condition = “all( category : this.categories,
                                all( article : category.articles,
                                        this.countCategoriesFor( article ) <= 3
 ) )“ )
public class AuctionShop{

        public Set<Category> getCategories(){...}

        public int countCategoriesFor( Article article ){...}

}
```

AuctionShop (Commands)

```
@Precondition(
        condition = “       not category.offersArticle( article )
                            and
                            this.countCategoriesFor( article ) < 3
                            and
                            not article.auctionOpened“ )
@Postcondition(
        condition = “       category.offersArticle( article ) == true
                            and
                            this.countCategoriesFor( article ) ==

                            old:this.countCategoriesFor( article ) + 1
                            and
                            category.articles.size =
old:category.articles.size + 1“ )
public void registerArticle( Category category, Article article )
```

AuctionShop (Commands)

```
@Precondition(
        condition = “       not article.auctionOpened“ )
@Postcondition(
        condition = “       this.countCategoriesFor( article ) == 0“ )
public void removeArticle( Article article )
```

```
@Precondition(
        condition = “       this.countCategoriesFor( article ) >= 1“ )
@Postcondition(
        condition = “       article.auctionOpened“ )
public void openAuction( Article article )
```

Benefits

Better Design

More systematic Design

Programmers are encouraged to think about such matters as preconditions, ...
the approach makes the concept explicit

Clearer Design

Obligations and benefits are shared between client and supplier and are clearly stated

Simpler Designs

Limitations on the use of a routine are clearly expressed

Programmers are encouraged **NOT** to build routines that are too general, but to design classes with small, single-purpose routines

Benefits

Improved Reliability

Better understood

You say the same thing twice in different ways – helps you understand more clearly what the routine does

Better tested

Assertions are checked at runtime, thereby testing that routines fulfill their stated

contracts

... and anything that helps testing will lead to code with fewer bugs

Benefits

Better Documentation

Clearer Documentation

Contracts for part of the public (client) view of a class

Reliable Documentation

Assertions are checked at runtime, thereby testing that the stated contracts are consistent with what the routine actually do (maintained with the code)

Support for precise Specification

Contracts provide a means to gain some of the benefits of precisespecifications while allowing programmers to work with their familiar operational intuitions

Benefits

Support for Maintenance

if programms are delivered with assertion checking switched on, customers can provide
developers with more accurate information on the circumstances surrounding a failure.

Supports for Reuse

Good documentation for library users – contracts clearly explain what routines in library
classes do and what constraints are on using them

well written preconditions give client programmers an accurate analysis when calling a library class under the wrong circumstances

Costs and Limitations

Takes practice

writing good contracts is a skill.

False sense of security

contracts cannot express all the desirable properties of programms

Quality is not always the primary goal

for some developments the most important goal is an early release ...

Best for sequential programming

in a distributed system you want routines that support 'shoot first, ask questions later' (instead of clients that 'look before they leap')

Quellen

Meyer, B.
Object-oriented software construction, 2nd ed.
Prentice Hall, 1997

Mitchell, R. And McKim, J.
Design by Contract, by Example
Addison Wesley, 2002

Building bug-free O-O software: An introduction to Design by Contract
http://archive.eiffel.com/doc/manuals/technology/contract/

Design by Contract
A Conversation with Bertrand Meyer, Part II
http://www.artima.com/intv/contracts.html

Design by Contract: The Lessons of Ariane
http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html