Konstruktive Qualitätssicherung

- Software Richtlinien
- Typisierung
- Vertragsbasierte Programmierung
- Fehlertolerante Programmierung
- Portabilität
- Dokumentation

Literatur

- Design by Contract in Java with Google
  https://objectcomputing.com/resources/publications/sett/september-2011-design-by-contract-in-java-with-google

- Contracts for Java
  https://github.com/nhatminhle/cofoja

- R. Mitchell, J. McKim: *Design by contract by Example*, Addison Wesley, Boston, 2002
- Building bug-free O-O software: An Introduction to Design by Contract
  - https://www.eiffel.com/values/design-by-contract/introduction/

Idee von DbC

- Definition von formalen und messbaren Vereinbarungen für Schnittstellen zwischen Modulen.
- Überprüfung der Vereinbarungen
- **Design** by Contract: Vor der Implementierung werden die Vereinbarungen festgeschrieben.

Design by Contract

Gründer: Bertrand Meyer im Zusammenhang mit Entwicklung der Programmiersprache Eiffel.

Es wird ein **Vertrag** zwischen Aufrufer und Aufgerufenem vereinbart, der Vor- und Nachbedingungen sowie Invarianten festlegt.

DbC

The Design by Contract theory, then, suggests associating a specification with every software element. These specifications (or contracts) govern the interaction of the element with the rest of the world.

Quelle: https://www.eiffel.com/values/design-by-contract/introduction/

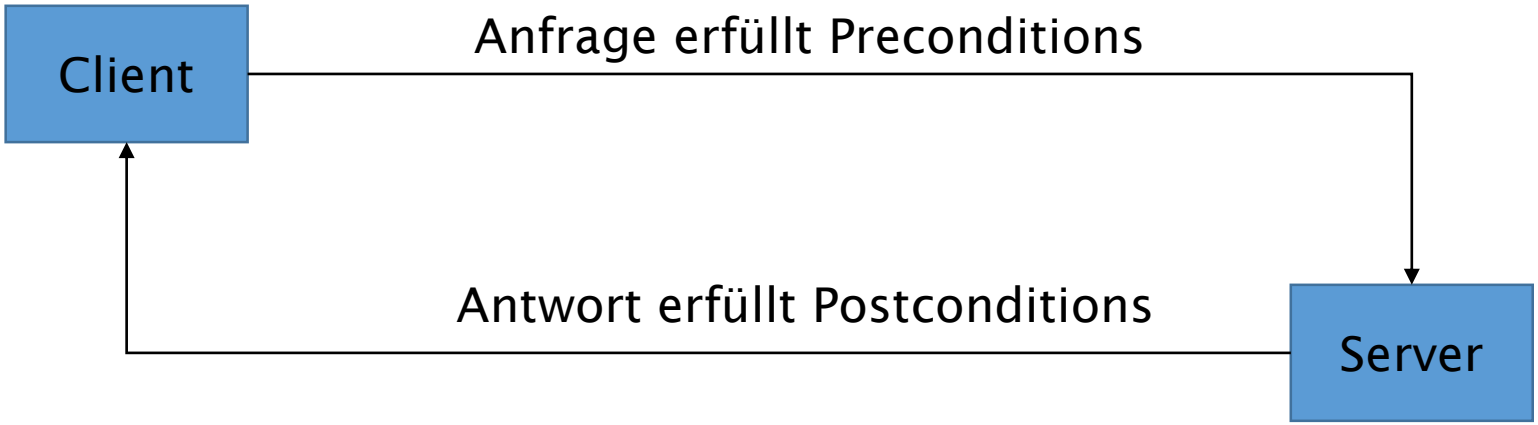
Grundprinzipien von DbC

Vorbedingungen:
Zusicherungen, die der Aufrufer zu beachten hat.

Nachbedingungen:
Zusicherungen, die der Aufgerufene zu beachten hat.

Invarianten:
Gesundheitszustand der Klasse. (logische Aussagen, die für alle Instanzen einer Klasse über den gesamten Objektlebenszyklus hinweg gelten)

DbC – Pre- und Postconditions



Verletzung der Constraints führt zu Programmabbruch.

Nutzen

- Steigerung der Qualität durch genaue Spezifikation der Schnittstellen.
- Contracts als Dokumentation (im Code)
- Vererbung wird unterstützt.
- Unterstützung beim Testen/Fehlerfinden.

Umsetzung von Design by Contract

In manchen Sprachen nativ verankert: Eiffel, D

Design by Contract in Java:

- Nicht in der Sprache verankert.
- Konzept anwenden: javadoc (oder Äquivalent) nutzen um Vor-, Nachbedingungen und Invarianten zu definieren.
- Rudimentär: Arbeiten mit assert.
- Möglichkeit mit frameworks das Konzept zu implementieren. Beispiele:
  - Bean Validation https://beanvalidation.org/
  - CoFoJa: https://github.com/nhatminhle/cofoja
  - Open JML https://www.openjml.org/
  - Contracts for Java (C4J): https://c4j-team.github.io/C4J/

Beispiel mit Contract for Java (https://github.com/nhatminhle/cofoja)

Annotationen :

Annotation	Für	
Invariant	Invariant	Check nach Konstruktor und zu Beginn und am Ende aller public und package-private Methoden
Requires	Precondition	Check zu Beginn der annotierten Methode
Ensures	Postcondition	Check am Ende der annotierten Methode
ThrowEnsures	Exceptional Postcondition	Check, wenn eine Exception geworfen wird.

Default: keine Auswirkung,
Contracts angeschalten: Spezifische Laufzeitexceptions bei Verletzungen.
(PreconditionError, PostConditionError, InvariantError)

Bsp: shopping cart

(aus https://objectcomputing.com/resources/publications/sett/september-2011-design-by-contract-in-java-with-google

Klasse Book: Invariant

```
package com.ociweb.sett.sep2011;

import com.google.java.contract.*;
import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;


@Invariant({"title != null && title.length() > 0", "price > 0"})
public class Book {
    private final String title;
    private int price;

    @Requires({"title != null && title.length() > 0", "price > 0"})
    public Book(String title, int price) {
        this.title = title;
        this.price = price;
    }

    public int getPrice() {
        return price;
    }

    @Requires("price > 0")
    public void setPrice(int price) {
        this.price = price;
    }
```

Invariant für die Klasse: es werden zu prüfende Bedingungen für die Member festgelegt

Bsp: shopping cart

Klasse Book: Preconditions

```
package com.ociweb.sett.sep2011;

import com.google.java.contract.*;
import org.apache.commons.lang3.builder.EqualsBuilder;
import org.apache.commons.lang3.builder.HashCodeBuilder;


@Invariant({"title != null && title.length() > 0", "price > 0"})
public class Book {
    private final String title;
    private int price;

    @Requires({"title != null && title.length() > 0", "price > 0"})
    public Book(String title, int price) {
        this.title = title;
        this.price = price;
    }

    public int getPrice() {
        return price;
    }

    @Requires("price > 0")
    public void setPrice(int price) {
        this.price = price;
    }
```

Preconditions. Prüfungen der Argumente.

Bsp: shopping cart

Klasse ShoppingCart: Postconditions

```
package com.ociweb.sett.sep2011;

import com.google.common.collect.*;
import com.google.java.contract.*;

@Invariant("books != null")
public class ShoppingCart {
    private final Multiset<Book> books = HashMultiset.create();

    @Requires("book != null")
    @Ensures("books.count(book) == old(books.count(book)) + copies")
    public void addBooks(Book book, int copies) {
        books.add(book, copies);
    }

    @Requires({"book != null", "books.count(book) >= copies"})
    @Ensures("books.count(book) == old(books.count(book)) - copies")
    public void removeBooks(Book book, int copies) {
        books.remove(book, copies);
    }
```

Postconditions. Prüfungen der Ergebnisse
Schlüsselwort „old“: Zugriff auf Ausdrücke vor dem Methodenaufruf.
Schlüsselwort „result“: Zugriff auf den Rückgabewert.