Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

Zukunft in Bewegung

Prinzipen für einen guten Entwurf

Architektur- und Entwurfsmuster der Softwaretechnik

Prof. Dr. Bernd Hafenrichter

Prinzipien für den Entwurf einzelner Klassen

Prinzipien objektorientierten Designs sind Prinzipien, die zu gutem objektorientierten Design führen sollen.

Dadurch werden folgende Qualitätsmerkmale unterstütz:

- Wartbarkeit
- Analysierbarkeit
- Anpassbarkeit
- Verständlichkeit
- Flexibilität bei Erweiterungen

Prinzipien für den Entwurf einzelner Klassen

Die SOLID – Prinzipien

- **S**ingle Responsibility-Prinzip
- **O**pen-Closed-Prinzip
- **L**iskovsches Substitutionsprinzip
- **I**nterface Segregation-Prinzip
- **D**ependency-Inversion-Prinzip

Quellen:

- https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs
- http://prinzipien-der-softwaretechnik.blogspot.de

SOLID-Prinzipien: Single Responsibility Prinzip

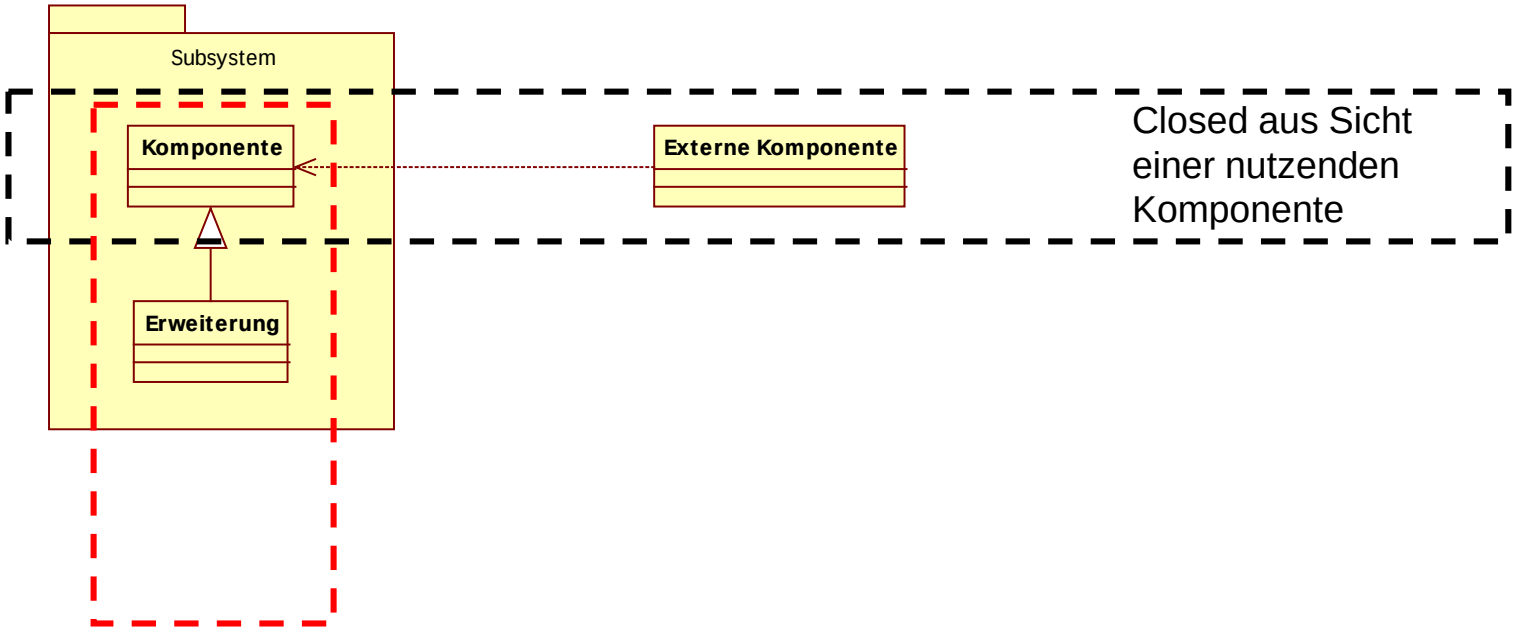
- **Single Responsibility Prinzip**: Jede Klasse soll nur eine einzige Verantwortung/Aufgabe haben.
  - Robert C. Martin: „There should never be more than one reason for a class to change”
  - In einer Klasse sollten lediglich Funktionen vorhanden sein, die direkt zur Erfüllung dieser Aufgabe beitragen.
  - Die Klasse wird nur verändert wenn sich die Aufgabe verändert
  - Die Änderung von unabhängigen Aufgaben haben geringeren Einfluss aufeinander

SOLID-Prinzipien: Open-Closed-Prinzip

- **Open-Closed-Prinzip:** Software-Einheiten sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein.
  - Eine Erweiterung in diesem Sinne verändert das vorhandene Verhalten der Einheit nicht, vielmehr erweitert es die Einheit um zusätzliche Funktionen oder Daten
  - Eine Modifikation hingegen, würde das bisherige Verhalten der Einheit ändern

SOLID-Prinzipien: Open-Closed-Prinzip

Open-Closed-Principle:

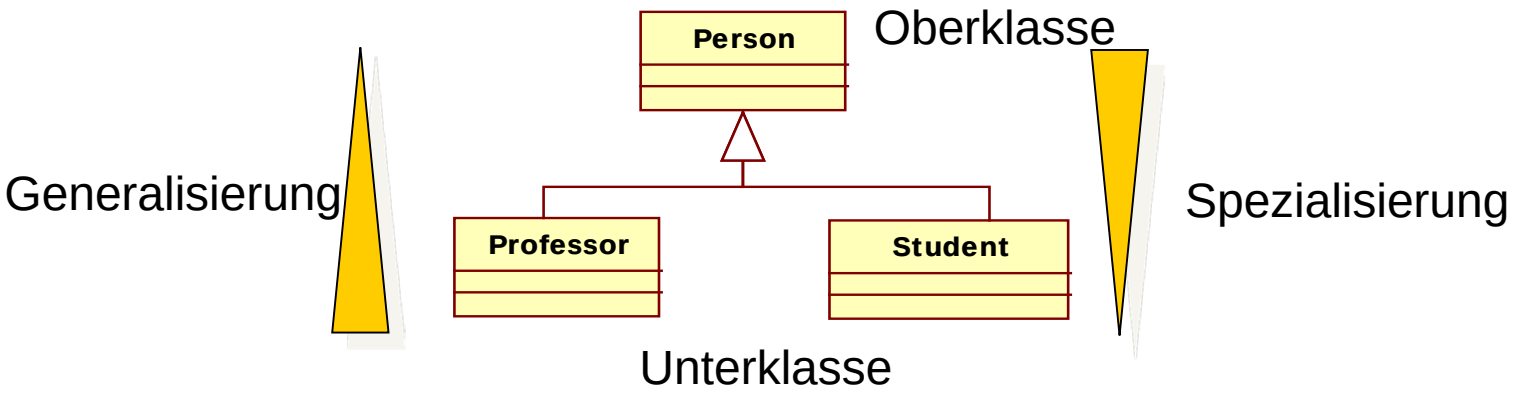


Erweiterbar aus ohne das „closed“ zu verletzen

SOLID-Prinzipien: Liskovsches Substitutionsprinzip

- Bei dem Design eines statischen Modells wird die Generalisierung/Spezialisierung verwendet um Entitäten mit gleichen Eigenschaften zusammenzufassen.
- Wie kann man prüfen dass die definierte Hierarchie korrekt ist ein keine Inkonsistenzen Zustände erlaubt
- Grundansatz: Substituierbarkeit ist gegeben, wenn in den Spezialisierung alle Eigenschaften gelten die auch in der Oberklasse gelten

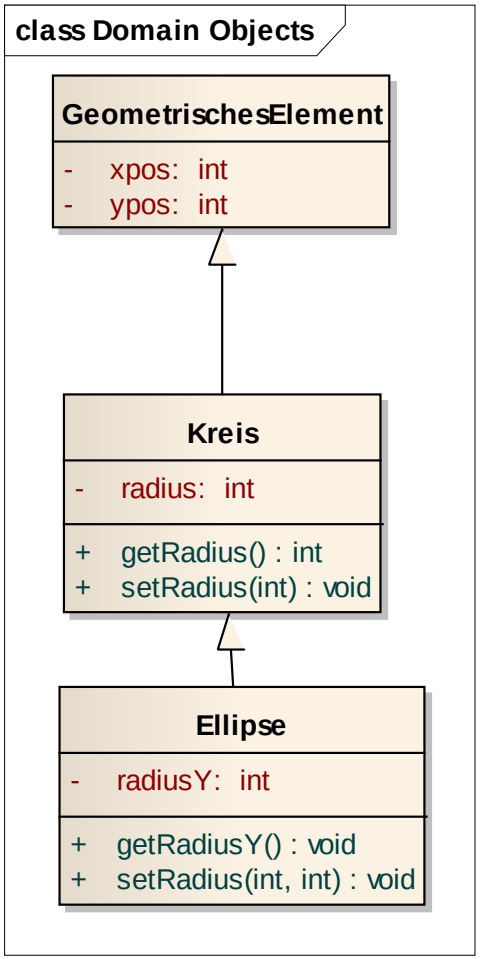
SOLID-Prinzipien: Liskovsches Substitutionsprinzip



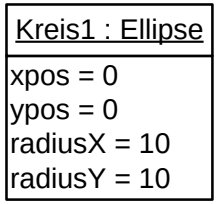
Taxonomische Beziehung zwischen einer spezialisierten Klasse und einer allgemeinen Klasse:

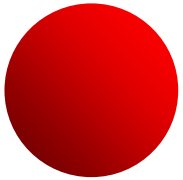
- Der spezialisierte erbt die Eigenschaften (Attribute, Assoziationen, Operationen) der allgemeinen Klasse
- Kann weitere Eigenschaften hinzufügen
- Eine Instanz der Unterklasse kann überall dort verwendet werden, wo eine Instanz der Oberklasse erlaubt ist (Substituierbarkeit)
- Mehrfachvererbung ist zulässig
- Spezialisierung kann auch auf Assoziationen, Akteure, … angewandt werden

SOLID-Prinzipien: Liskovsches Substitutionsprinzip

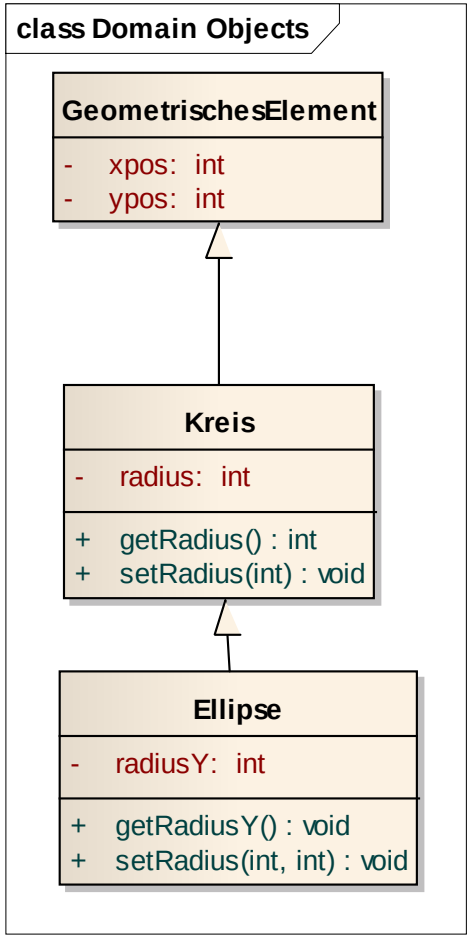


Welches „semantische“ Problem enthält die modellierte Klassenhierarchie ???

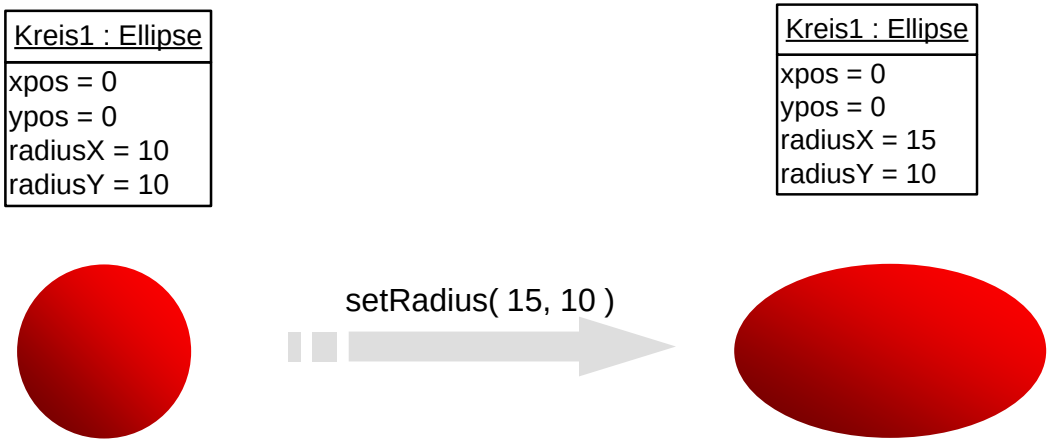




SOLID-Prinzipien: Liskovsches Substitutionsprinzip



Welches „semantische“ Problem enthält die modellierte Klassenhierarchie ???

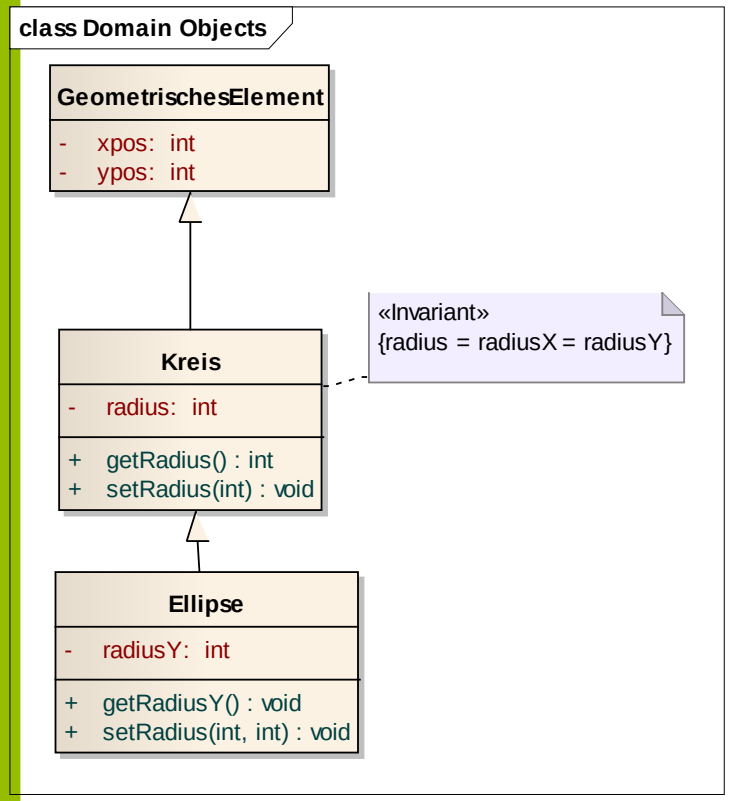


Problem:

Wird auf das Objekt Kreis1 die Methode setRadius(10, 15) angewendet entsteht eine Wiederspruch, da radiusX != radiusY.

D.h. der Kreis ist kein Kreis mehr

SOLID-Prinzipien: Liskovsches Substitutionsprinzip



Sei q(x) eine beweisbare Eigenschaft von Objekten x des Typs T. Dann soll q(y) für Objekte y des Typs S wahr sein, wobei S ein Untertyp von T ist.

Man kann ein Objekt y des Untertyps S von T so verwenden, als wäre es ein Objekt x vom Typ T.

Wichtige Eigenschaften nach Liskov:

- Alle Eigenschaften (Invarianten) der Oberklasse müssen auch/immer für alle Unterklassen gelten
- Beim Spezialisieren/Generalisieren muss dies Berücksichtigt werden
- Bedingungen in den Unterklassen dürfen Bedingungen der Oberklasse verfeinern, niemals aber erweitern
- Vermeidung schwerwiegender semantischer Probleme bei der Verwendung von Subtypen.

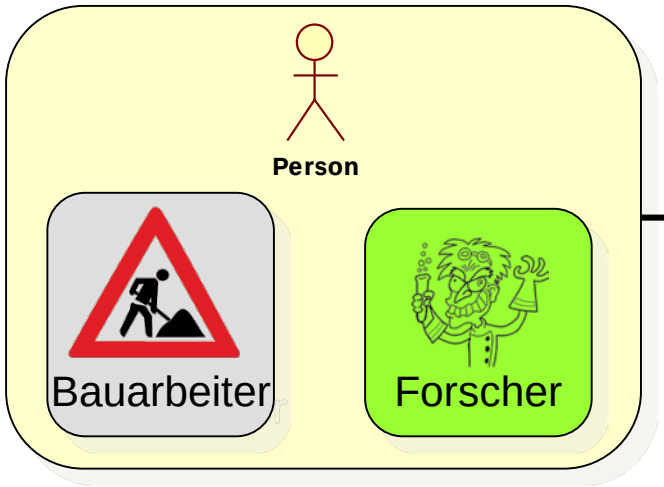
SOLID-Prinzipien: Liskovsches Substitutionsprinzip

Barbara Liskov:

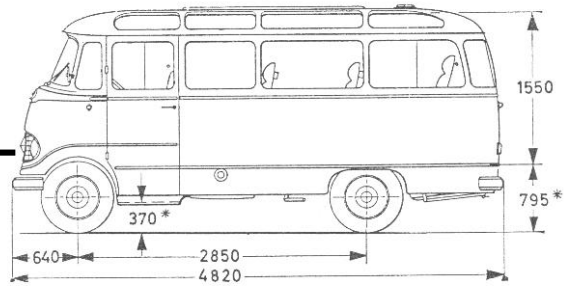
- Definierte das Liskovsche Substitutionsprinzip
- Professorin für Elektrotechnik und Informatik am MIT
- Mitglied der amerikanischen National Academy of Engineering
- 2004 erhielt sie die John-von-Neumann-Medaille für „fundamental contributions to programming languages, programming methodology, and distributed systems“

SOLID-Prinzipien: Liskovsches Substitutionsprinzip

- Substituierbarkeit bei Assoziationen



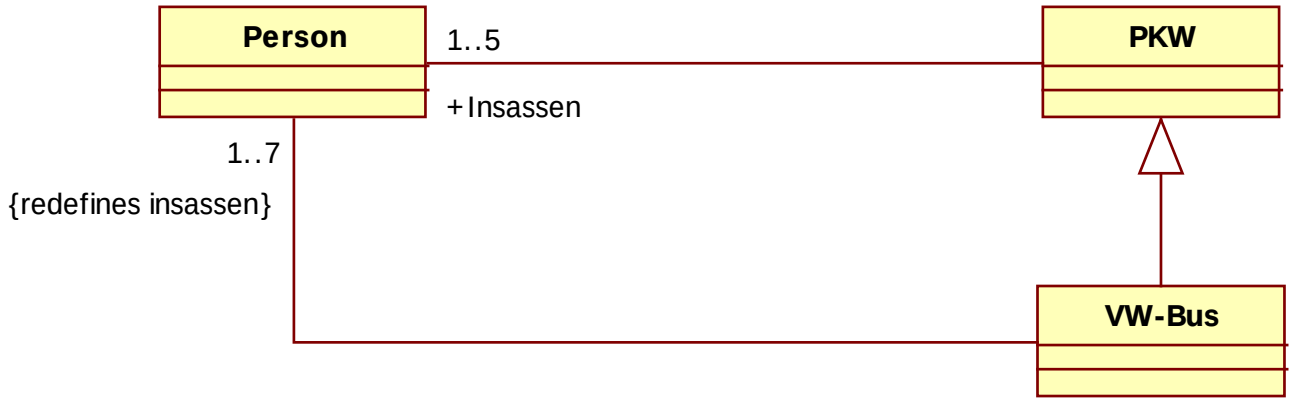
Beförderung



Jede beliebige Person kann durch den Bus Befördert werden

SOLID-Prinzipien: Liskovsches Substitutionsprinzip

- Substituierbarkeit bei Assoziationen



Problem:

- PKW dürfen maximal 5 Personen befördern
- Ein VW-Bus kann bis zu 7 Personen befördern. Hier ist ein Widerspruch, da ja ein PKW nur 5 Personen befördern kann
- D.h. VW-Bus verletzt Eigenschaften von PKW

SOLID-Prinzipien: Interface-Segregation-Prinzip

- **Interface-Segregation-Prinzip:** Ein Aufrufer sollte nur von Schnittstellen abhängig sein der er auch tatsächlich benötigt
  - Aufteilung eines großen Interfaces in kleinere Teile
  - Dadurch wird die Abhängigkeit von unnötigen Schnittstellen reduziert
  - Der Aufwand für die Implementierung einer Schnittstelle wird ebenfalls reduziert

  - Vorteil: Liefert für Interfaces eine Operationalisierung des Prinzips der hohen Kohäsion.

SOLID-Prinzipien: Interface-Segregation-Prinzip

```
interface ManagerService {

   public void sendMail( ... );

   public void createCustomer( ... );

   public void deleteCustomer( ... );

   public void sendBill( ... );

   public void checkPaiment( ... );

   public HTLM createHTMLViewForKonto( ...);

   public HTLM createPDFViewForKonto( ...);
}
```

SOLID-Prinzipien: Interface-Segregation-Prinzip

```
interface EMailService {

   public void sendMail( ... );
}
```

```
interface CustomerService {

   public void createCustomer( ... );

   public void deleteCustomer( ... );

}
```

```
interface InvoiceService {

   public void sendBill( ... );

   public void checkPaiment( ... );
}
```

```
interface ExportService {

   public HTLM createHTMLViewForKonto( ...);

   public PDF  createPDFViewForKonto( ...);
}
```

SOLID-Prinzipien: Dependency Inversion Principle

- **Dependency Inversion Principle**: „A. Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Interfaces abhängen. B. Interfaces sollten nicht von Details abhängen. Details sollten von Interfaces abhängen.“
  - Damit ist sichergestellt, dass die Abhängigkeitsbeziehungen immer in eine Richtung verlaufen, von den konkreten zu den abstrakten Modulen, von den abgeleiteten Klassen zu den Basisklassen
  - Damit werden die Abhängigkeiten zwischen den Modulen reduziert und insbesondere zyklische Abhängigkeiten vermieden.

SOLID-Prinzipien: Dependency Inversion Principle

- Vorteile:
  - In Kombination mit Dependency Injection Containern sehr einfach anwendbar.
  - Sehr universell, da auf praktisch allen Strukturierungs-Ebenen gültig. Dadurch breiter Anwendungsbereich.
- Nachteile:
  - erhöhter Entwurfs- und Umsetzungsaufwand
  - weniger konkret, dadurch weniger unmittelbar verständlich

Weiterführende Prinzipien Objektorientierten Designs

- Gesetz von Demeter
- Design by Contract
- Datenkapselung
- Abstraktion
- Linguistic-Modular-Units-Prinzip
- Self-Documentation Prinzip
- Uniform Access Prinzip
- Single-Choice-Prinzip
- Persistence-Closure-Prinzip

Weitführende Prinzipien: Gesetz von Demeter

- **Gesetz von Demeter**: „A supplier object to a method M is an object to which a message is sent in M. The preferred supplier objects to method M are: The immediate parts of self or the argument objects of M or the objects which are either objects created directly in M or objects in global variables or attributes. Every supplier object to a method must be a preferred supplier.“
  - Das Gesetz von Demeter (englisch: Law of Demeter, kurz: LoD) besagt im Wesentlichen, dass Objekte nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren sollen.
  - Dadurch soll die Kopplung in einem Softwaresystem verringert und dadurch die Wartbarkeit erhöht werden.

Weitführende Prinzipien: Gesetz von Demeter

- Vorteile:
  - Einfach und teils automatisiert erkenn- und anwendbar.
  - Verankert in zahlreichen weiteren Prizipien.
  - Empirisch belegt.
- Nachteile:
  - Bei Einhaltung des LoD entsteht systematisch einer von zwei unerwünschten Nebeneffekten:
    - eine größere Anzahl von Methoden, oft lediglich zusätzliche Vermittlermethoden (dadurch Schwächung der [Kohäsion] der Klasse)
    - längere Parameterlisten einzelner Methoden (dadurch erschwerte Verständlichkeit)
  - Das Laufzeitverhalten kann sich verschlechtern.

Weitführende Prinzipien: Gesetz von Demeter

```
package demeter;

class Motor {
    public void starten() {
        // den Motor starten.
    }
}

class Auto {
    public Motor motor;
    public Auto() {
        motor = new Motor();
    }
}

class Fahrer {
    public void fahren() {
        Auto zuFahrendesAuto = new Auto();
        zuFahrendesAuto.motor.starten(); //hier wird gegen das Gesetz verstoßen
    }
}
```

Weitführende Prinzipien: Gesetz von Demeter

```
class Auto {
    private Motor motor;
    public Auto() {
        motor = new Motor();
    }
    public void anlassen() {
        motor.starten();
    }
}

class Fahrer {

    public void fahren() {
        Auto zuFahrendesAuto = new Auto();
        zuFahrendesAuto.anlassen();
    }
}
```

Weitführende Prinzipien: Design by Contract

- Klassen stehen in Beziehungen zu anderen Klassen.
- Als Partner interagieren diese Klassen miteinander
- Damit das gesamte Programm korrekt funktioniert müssen sich die Partner an festgelegte Bedingungen (Veträge) halten.

- Ursprung
  - Betrand Meyer
  - Erstmals als Entwurfstechnik in der Programmiersprache Eiffel

Weitführende Prinzipien: Design by Contract

- Verträge werden durch Zusicherungen ausgedrückt
- Eine Zusicherung ist ein boolscher Ausdruck der immer true sein muss
- Für eine Klasse existieren folgende Zusicherung
  - Vorbedingungen
  - Nachbedingung
  - Invarianten

Weitführende Prinzipien: Design by Contract

- Der Vertrag einer Methode besteht aus Vor- und Nachbedingung
  - Vorbedingung:
    - Der Aufrufer hat die Pflicht die Vorbedingung der aufgerufenen Methode zu erfüllen.
    - Der Aufgerufen geht davon aus dass die Vorbedingung eingehalten wird
  - Nachbedingung:
    - Stellt den Zustand eines Objektes nach Aufruf der Methode dar
    - Der Aufgerufene ist Verpflichtet die Nachbedingung sicherzustellen

Weitführende Prinzipien: Design by Contract

- Der Vertrag einer Klasse besteht aus den Verträgen aller Methoden sowie zusätzlichen Invarianten.

- Invariante:
  - Ein Zusicherung welche sich auf die Eigenschaft der Klasse bezieht.
  - Eine Invariante gilt für alle Operation
  - Invariante müssen von allen öffentlichen Methoden vor und nach dem Methodenaufruf eingehalten werden.
  - Die Invarianten gelten während der gesamten Lebensdauer eines Objektes
  - Eine Invariante kann temporär verletzt (nichtöffentliche Methoden)

Weitführende Prinzipien: Design by Contract

- Idee:
  - Design By Contract wird durch die Programmiersprache unterstützt.
  - Zur Laufzeit werden die Zusicherungen automatisch überprüft.
  - Spezifikation und Prüfung sind Konsistenz

- Quellen für Design By Contract in Java

  - http://www.javaworld.com/article/2074956/learn-java/icontract--design-by-contract-in-java.html
  - *www.gruntz.ch/courses/sem/ws06/DBC.pdf*

Weitführende Prinzipien: Design by Contract

Spezifikation von Zusicherungen über OCL:

- Notation

  context Klassenname::Opname(param1 : Typ1, . . .) : Ergebnistyp

  pre: [Name‘:‘]OCLExpression

  post: [Name‘:‘] OCLExpression

  body**:** OCLExpression
- Schlüsselwort result für Ergebnis der Operation
- Zugriff auf Attributwerte vor Aufruf mit *attribute*@**pre**

```
context countDown()
pre: self.count > 0
post: result = self.count@pre - 1
```

Weitführende Prinzipien: Design by Contract

- Ein Invariante definiert eine allgemeingültige Zusicherung, welche für alle Instanzen des Kontext gültig ist (während der gesamten Lebenszeitzeit).
- Invarianten können verwendet werden um dass „Allgemeinwissen“ über einer Problemdomäne formal darzustellen.
- Notation

  context *Klassenname* **inv:** *OCLExpression*

  context c:*Klassenname* **inv:** *OCLExpression*

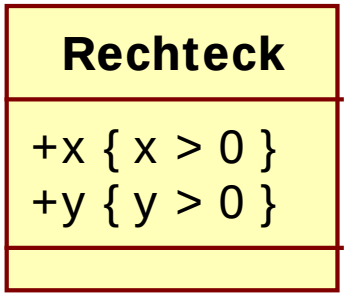
  context c:*Klassenname* **inv** *Name***:** *OCLExpression*

- Es existiert eine Variable „self“ welche sich immer auf eine spezifische Instanz von Kontext bezieht. D.h. self identifiziert das Objekt für welches die Zusicherung geprüft wird
- **Beispiel:**

  context Company inv: self.manager.isUnemployed=false

Weitführende Prinzipien: Design by Contract

- Zusicherung für Attribute können auch direkt in der Klasse selbst modelliert werden



- Dies ist eine Kurzschreibweise für:
  - context Rechteck inv: a > 0
  - Bedetung:
    - Für alle Instanzen von Rechteck muss das Attribut a immer größer 0 sein

Weitführende Prinzipien: Datenkapselung

- **Kapselung**: Der **kontrollierten** Zugriff auf Methoden bzw. Attribute von Klassen.
  - Klassen können den internen Zustand anderer Klassen nicht in unerwarteter Weise lesen oder ändern.
  - Eine Klasse hat eine Schnittstelle, die darüber bestimmt, auf welche Weise mit der Klasse interagiert werden kann.
  - Zugriff auf Attribute erfolgt nur über Methoden
  - Invarianten der Klasse können nicht umgangen werden

Weitführende Prinzipien: Datenkapselung

- **Vorteile der Kapselung**:
  - Leichtere Änderbarkeit
  - Bessere Übersichtlichkeit
  - Innere Struktur kann unabhängig verändert/angepasst werden
  - Verbesserte Testbarkeit, Stabilität und Änderbarkeit
  - Vereinfachte Fehlersuche

- **Nachteile der Kapselung**
  - Geschwindigkeitseinbußen durch die Zugriffssfunktion
  - Zusätzlicher Programmieraufwand

Weitführende Prinzipien: Abstraktion

- **Abstraktion**: Die Schnittstelle/Signatur einer Methode abstrahiert das Verhalten eines Objektes.
  - Die öffentliche Schnittstelle eines Objektes ist wohldefiniert
  - Die Implementierung einer Methode ist nach außen nicht sichtbar nur die Schnittstelle
  - Die „Schnittstelle“ eines Objektes sind die öffentliche Methoden

Weitführende Prinzipien: Linguistic-Modular-Units-Prinzip

- **Linguistic-Modular-Units-Prinzip**: „Modules must correspond to syntactic units in the language used.“
  - Das Linguistic-Modular-Units-Prinzip (englisch für Prinzip linguistisch-modularer Einheiten) besagt, dass Module durch syntaktische Einheiten der verwendeten Sprache - sei es eine Programmier-, Design- oder Spezifikationssprache - repräsentiert werden müssen.
  - Im Falle einer Programmiersprache müssen Module durch separat voneinander kompilierbare Einheiten dieser gewählten Programmiersprache repräsentiert werden:

Weitführende Prinzipien: Self-Documentation Prinzip

- **Self-Documentation Prinzip**: „The designer of a module should strive to make all information about the module part of the module itself”
  - Das Self-Documentation-Prinzip (englisch für Selbstdokumentierungsprinzip) besagt, dass alle Informationen zu einem Modul in diesem selbst enthalten sein sollten.
  - Damit wird gefordert, dass der Code entweder selbst für sich spricht (d. h. keine weitere Dokumentation nötig hat) beziehungsweise die technische Dokumentation möglichst nahe beim Sourcecode (beispielsweise bei der Verwendung von Javadoc) liegt.
  - Dadurch wird einerseits gewährleistet, dass die technische Dokumentation dem Code entspricht und andererseits, dass die Komplexität des Codes so gering ist, dass keine komplexe Dokumentation nötig ist:

Weitführende Prinzipien: Uniform Access Prinzip

- **Uniform Access Prinzip**: All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.
  - Das Uniform-Access-Prinzip (englisch für Prinzip des gleichartigen Zugriffes) fordert, dass auf alle Services eines Modules mittels einer gleichartigen Notation zugegriffen werden kann - ohne dass dadurch preisgegeben wird, ob die Services dahinter auf Datenbestände zugreifen, oder Berechnungen durchführen.
  - Services sollten also nach außen nicht bekanntgeben, wie sie ihre Serviceleistung erbringen, sondern nur was ihre Serviceleistung ist:

Weitführende Prinzipien: Single-Choice-Prinzip

- **Single-Choice-Prinzip**: Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.
  - Das Single-Choice-Prinzip besagt, dass verschiedene Alternativen (beispielsweise von Daten oder Algorithmen) in einem Softwaresystem in nur einem einzigen Modul abgebildet werden sollten.
  - Beispielsweise unterstützt das Entwurfsmuster Abstrakte Fabrik dieses Prinzip, indem in der Fabrik einmal aus einem Set an Alternativen entschieden wird, welche Klassen und welche Algorithmen zu verwenden sind. Diese Entscheidung muss an keiner weiteren Stelle im Programm mehr getroffen werden:

Weitführende Prinzipien: Persistence-Closure-Prinzip

- **Persistence-Closure-Prinzip**: Whenever a storage mechanism stores an object, it must store with it the dependents of that object. Whenever a retrieval mechanism retrieves a previously stored object, it must also retrieve any dependent of that object that has not yet been retrieved.
  - Das Persistence-Closure-Prinzip besagt, dass Persistenzmechanismen Objekte mit all ihren Abhängigkeiten speichern und auch wieder laden müssen.
  - Damit ist sichergestellt, dass Objekte ihre Eigenschaften durch Speichern und darauffolgendes Laden nicht verändern.

Packaging-Prinzipien

- Reuse-Release-Equivalence-Prinzip
- Common-Closure-Prinzip
- Common-Reuse-Prinzip
- Acyclic-Dependencies-Prinzip
- Stable-Dependencies-Prinzip
- Stable-Abstractions-Principle

Packaging-Prinzipien: Reuse-Release-Equivalence-Prinzip

- **Reuse-Release-Equivalence-Prinzip**: The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package.“
  - Das Reuse-Release-Equivalence-Prinzip besagt, dass diejenigen Klassen, welche üblicherweise gemeinsam released werden, zu Modulen zusammengeführt werden sollten.
  - Damit wird gewährleistet, dass die Anzahl der zu verteilenden Module möglichst gering gehalten wird, was wiederum die Aufwände der Softwareverteilung reduziert.

Packaging-Prinzipien: Common-Closure-Prinzip

- **Common-Closure-Prinzip**: The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.
  - Das Common Closure Prinzip fordert, dass alle Klassen in einem Modul gemäß dem Open-Closed-Prinzip geschlossen gegenüber derselben Art von Veränderungen sein sollten.
  - Änderungen an den Anforderungen einer Software, welche Änderungen an einer Klasse eines Moduls benötigen, sollten auch die anderen Klassen des Moduls betreffen.
  - Die Einhaltung dieses Prinzips ermöglicht es die Software derart in Module zu zerlegen, dass (zukünftige) Änderungen in nur wenigen Modulen umgesetzt werden können.
  - Damit ist sichergestellt, dass Änderungen an der Software ohne große Seiteneffekte und somit relativ kostengünstig gemacht werden können.

Packaging-Prinzipien: Common-Reuse-Prinzip

- **Common-Reuse-Prinzip**: The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.
  - Das Common-Reuse-Prinzip beschäftigt sich mit der Verwendung von Klassen. Es besagt, dass diejenigen Klassen, welche gemeinsam verwendet werden, auch gemeinsam zu einem Modul zusammengefasst werden sollten.
  - Durch die Einhaltung dieses Prinzips wird eine Unterteilung der Software in fachlich bzw. technisch zusammengehörende Einheiten sichergestellt.

Packaging-Prinzipien: Acyclic-Dependencies-Prinzip

- **Acyclic-Dependencies-Prinzip**: The dependency structure between packages must be a directed acyclic graph (DAG). That is, there must be no cycles in the dependency structure.
  - Das Acyclic-Dependencies-Prinzip fordert, dass die Abhängigkeiten zwischen Modulen zyklenfrei sein müssen.
  - D. h. wenn Klassen in einem Modul von anderen Klassen in einem anderen Modul beispielsweise durch Vererbungs- oder Relationsbeziehungen abhängen, dann dürfen keine Klassen des anderen Moduls direkt oder indirekt von Klassen des ersteren Moduls abhängen.

Packaging-Prinzipien: Stable-Dependencies-Prinzip

- **Stable-Dependencies-Prinzip**: The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is.
  - Das Stable-Dependencies-Prinzip besagt, dass die Abhängigkeiten zwischen Modulen in Richtung der größeren Stabilität der Module gerichtet sein sollten.
  - Ein Modul sollte also nur von Modulen abhängig sein, welche stabiler sind als es selbst.

Packaging-Prinzipien: Stable-Dependencies-Prinzip

- **Stable-Dependencies-Prinzip**: The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable that it is.

$$I = \frac{Aa}{(Ae + Aa)}$$

I – Instabilität eines Modules
Ae … eingehende Abhängigkeiten (engl. *afferent couplings*).
Aa … ausgehende Abhängigkeiten (engl. *efferent couplings*).

Die Instabilität liegt im Bereich von 0 bis 1, eine Instabilität von 0 weist auf ein maximal stabiles Modul hin, eine von 1 auf ein maximal instabiles Modul.

Packaging-Prinzipien: Stable-Abstractions-Principle

- **Stable-Abstractions-Principle**: Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstraction of a package should be in proportion to its stability.“
  - Das Stable-Abstractions-Prinzip fordert, dass die Abstraktheit eines Moduls direkt proportional zu seiner Stabilität sein muss.

Packaging-Prinzipien: Stable-Abstractions-Principle

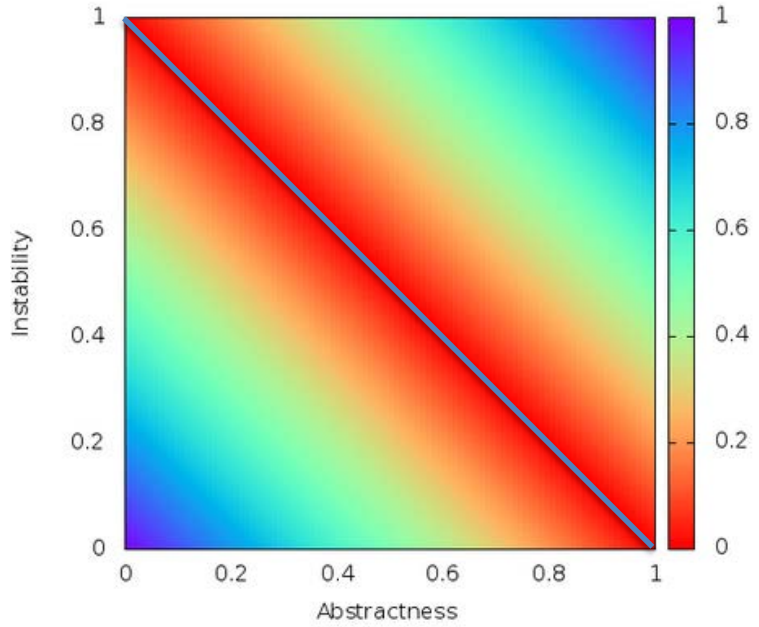
- **Stable-Abstractions-Principle**:

$$A = \frac{Ka}{K}$$

A – Abstraktheit des Modules
Ka - Anzahl abstrakter Klassen eines Moduls
K - Gesamtanzahl der Klassen eines Moduls

Packaging-Prinzipien: Stable-Abstractions-Principle

- **Stable-Abstractions-Principle**:
- Für jedes Modul lässt sich die Distanz zur idealen Linie - engl. Main Sequence genannt - zwischen maximaler Stabilität und Abstraktheit und maximaler Instabilität und Konkretheit errechnen.



$$D = \frac{A + I - 1}{\sqrt{2}}$$

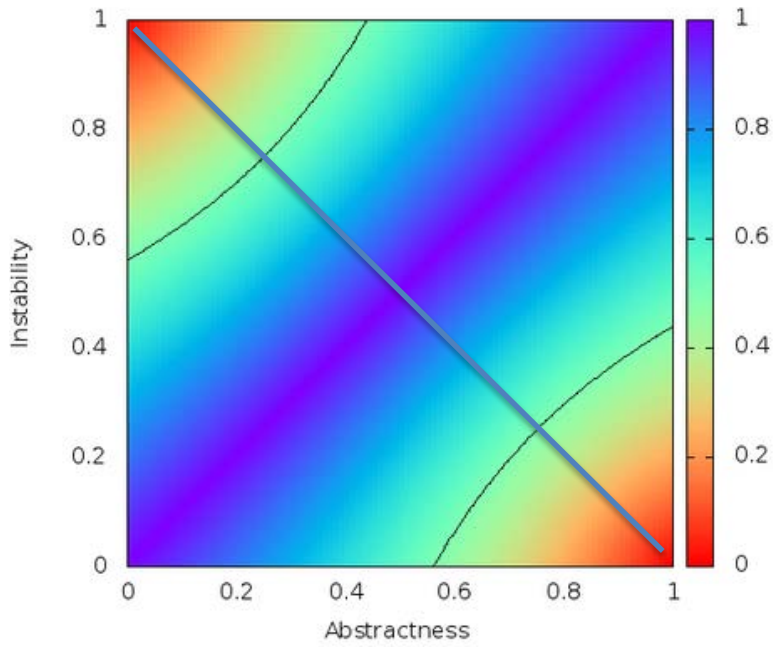
D – Distanz zur Ideallinie
A - Abstraktheit eines Moduls
I - Instabilität eines Moduls

Je größer die Distanz ist, desto schlechter ist das Stable-Abstractions-Prinzip erfüllt

Packaging-Prinzipien: Stable-Abstractions-Principle

- **Stable-Abstractions-Principle**:
- Alternative Definition der Distanzfunkiton

$$D' = \sqrt{(2)} - \left| \left(\sqrt{((1-I)^2 + A^2)} - \sqrt{(I^2 + (1-A)^2)} \right) \right| / \sqrt{(2)}$$



http://www.lexicalscope.com/blog/2012/10/31/alternative-distance-function-for-stable-abstractions-principle/

D – Distanz zur Ideallinie
A - Abstraktheit eines Moduls
I - Instabilität eines Moduls

Je größer die Distanz ist, desto schlechter ist das Stable-Abstractions-Prinzip erfüllt