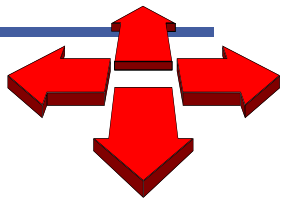
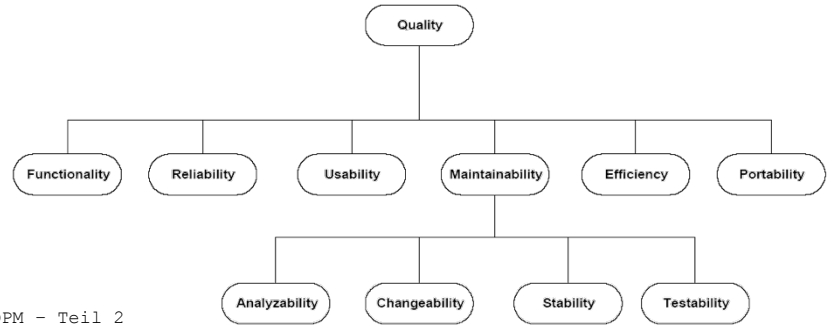
Übersicht Klassenentwurf



- Erneute Motivation: Softwarequalität
- Richtlinien für den Klassenentwurf
  - Anwendungsfachliche und technische Klassen
  - Kopplung
  - Kohäsion
- SOLID

Wdh.: Motivation: Qualität von Software

- Wir streben nach **möglichst hoher Qualität** bei der Softwareerstellung.
- Die Qualität von Software kann nach B. Meyer differenziert werden in
  - **Äußere bzw. externe Qualität** (Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz) und
  - **Innere bzw. interne Qualität** (Verständlichkeit, Wartbarkeit, Modularität).
  - In der Benutzung zählt nur die äußere Qualität, die aber über interne Qualität erreicht und gesichert wird.



Qualität von Klassenentwürfen

- Differenzieren des Begriffs **Entwurf**:
  - Als Bezeichnung für die **bestehende Struktur** eines Systems.
    - Im Sinne von „Hier haben wir einen guten Entwurf.“
    - Welche Elemente existieren? Wie arbeiten diese zusammen?
  - Als Bezeichnung der **Tätigkeit des Entwerfens**:
    - Im Sinne von „Beim Klassenentwurf (sprich: beim Entwerfen der Klassen) haben wir festgestellt, dass die Aufteilung nicht einfach ist.“
    - Beinhaltet neben dem Treffen von Entscheidungen auch eine Planungskomponente.
    - Prozessunterstützung beispielsweise durch **CRC-Karten**.
- **Aber wann ist ein Klassenentwurf im ersten Sinn „gut“?**

Leitbild: Entwurf nach Zuständigkeiten

- **Entwurf nach Zuständigkeiten** (engl.: Responsibility-Driven Design) ist eine Entwurfsphilosophie, die von Rebecca Wirfs-Brock et al. Ende der 80er Jahre formuliert wurde.

„Objects are not just simple bundles of logic and data. They are responsible members of an object community.“

- Jedes Objekt in einem objektorientierten System sollte für eine klar definierte Aufgabe zuständig sein.
- Dieser Ansatz geht u.a. auf die Forderung nach „Separation of Concerns“ von Dijkstra zurück.

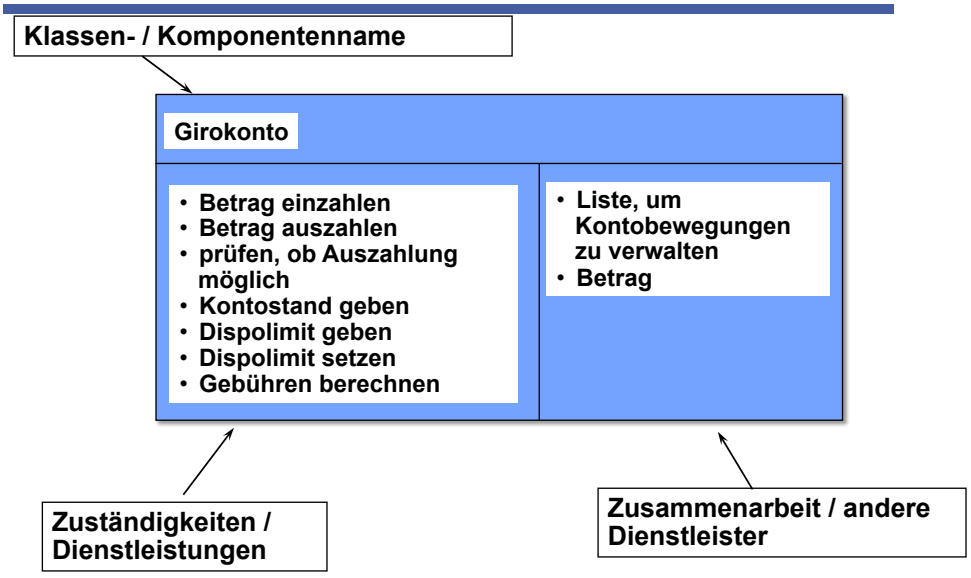
Aktuelles Buch: Wirfs-Brock, McKean: *Object Design – Roles, Responsibilities and Collaborations*, Addison-Wesley 2002

CRC: Class – Responsibility – Collaboration

- **Klassen- / Komponentenname** (**C**lass / Component Name)
  Der Klassen- / Komponentenname ist ein wichtiger Begriff des Anwendungsgebiets bzw. des Anwendungssystems. Bei der Analyse ist er häufig Teil der Fachsprache.
- **Zuständigkeiten** (**R**esponsibility)
  Die Zuständigkeiten charakterisieren die von der Klasse / Komponente erbrachten Dienstleistungen. Sie sind ein in sich zusammenhängendes Angebot an potentielle Klienten.
- **Zusammenarbeit** (**C**ollaboration)
  In diesem Teil werden andere Anbieter von Dienstleistungen (also: andere Klassen / Komponenten) benannt, an die Zuständigkeiten delegiert werden, um die eigene Dienstleistung zu erbringen.

Absolut lesenswerter Klassiker dazu:
Kent Beck, Ward Cunningham: “A Laboratory for Teaching Object-Oriented Thinking”, Proceedings OOPSLA, ACM SIGPLAN Notices, Volume 24, Issue 10, pp. 1-6, 1989.

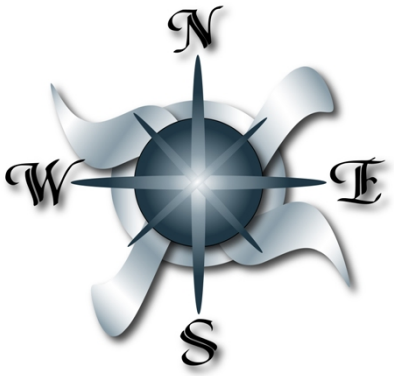
CRC-Karten: Drei Abschnitte



SE2 – OOPM – Teil 2 7

Wie kommen wir zu einem guten Entwurf?

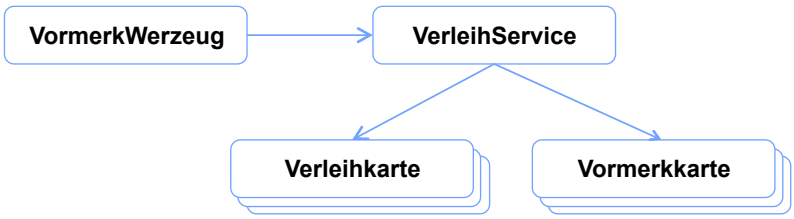
- **Richtlinien für den Klassentwurf**
  - Allgemein: Entwurf nach Zuständigkeiten
  - Lose Kopplung
    - Geeignete Schnittstellen wählen
    - Entwurfsentscheidungen kapseln
    - Geheimnisprinzip
    - Zyklen vermeiden
    - Law of Demeter
  - Hohe Kohäsion
    - für Klassen und für Methoden
    - Code-Duplizierung vermeiden
    - Geeignete Bezeichner wählen
    - Große Einheiten vermeiden



SE2 – OOPM – Teil 2 8

Beispiel: Zuständigkeit beim Vormerken in der Mediathek

- In der **Mediathek** sollte die Funktionalität des **Vormerkens** eingebaut werden.
- Die GUI war schon vorgegeben, es sollten lediglich die Services und Materialien erweitert werden.
- In der GUI gibt es einen **Button**, der nur dann **aktiviert** werden soll, wenn alle ausgewählten Medien für den ausgewählten Kunden **vormerkbar** sind.
- Wer ist dafür **zuständig**, diese Information zu ermitteln?

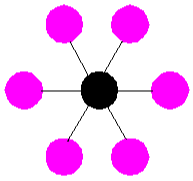


Wiederholung: Trenne Fachlogik und Technik

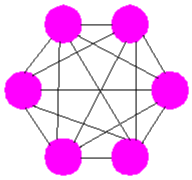
- Anwendungsfachliche Klassen, die vor allem die Fachlogik modellieren, sollten deutlich von rein technisch motivierten Klassen unterscheidbar sein.

Kopplung, objektorientiert

- **Kopplung** (engl.: coupling) bezeichnet den **Grad der Abhängigkeiten zwischen** den **Einheiten** eines Softwaresystems.
  - **Abhängigkeiten** können aus objektorientierter Sicht sein:
    - Benutzt-Beziehungen
    - Vererbungsbeziehungen
  - **Einheiten** können sein:
    - Methoden
    - Klassen
    - Pakete
    - Subsysteme
- Je mehr Abhängigkeiten in einem System existieren, desto stärker ist die Kopplung.
- Wir streben beim Klassenentwurf nach **möglichst loser Kopplung** (engl.: loose coupling).



Wenn eine Klasse mit allen anderen Klassen verbunden ist, ist das ihre maximal mögliche Kopplung.



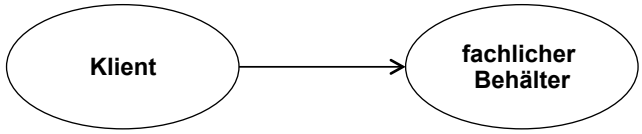
Lose Kopplung

- Lose Kopplung zwischen Klassen ist erstrebenswert, weil...
  - wir den Text einer einzelnen **Klasse besser verstehen** können, wenn wir nicht viele andere Klassendefinitionen lesen müssen;
  - wir eine Klasse **leichter ändern** können, wenn nicht viele andere Klassen von dieser Änderung betroffen sind.
- Mit anderen Worten: lose Kopplung erleichtert die Wartung.

- Wir unterscheiden **explizite** und **implizite** Kopplung:
  - Eine Kopplung ist **explizit**, wenn sie **formal nachweisbar** ist.
    Bsp.: Zugriff eines Klienten auf öffentliche Exemplarvariablen ist durch Analysewerkzeuge als explizite enge Kopplung nachweisbar.
  - **Implizite** Kopplung ist **nicht formal nachweisbar**; sie ist deshalb deutlich unangenehmer.
    Bsp.: Ein Entwickler implementiert sein Wissen über die Interna eines Dienstleisters in eine Klientenklasse hinein.

Beispiel für implizite Kopplung

Schnittstelle u.a.:
Liefert alle Elemente nacheinander.



Implementation:
Liefert die Elemente alphabetisch sortiert.

Implementierer kennt die Implementation des Behälters und verlässt sich auf die alphabetische Sortierung.

Monate später:
Ein Wartungsprogrammierer ersetzt die Implementation des fachlichen Behälters; er hält sich sauber an die Schnittstelle, liefert aber nicht mehr alphabetisch sortiert.

Effekt: Der Klienten-Code zerbricht.

Geeignete Schnittstellen wählen

- Über die Schnittstellen von Klassen werden die meisten Abhängigkeiten definiert.
- Für eine Schnittstelle sollte gelten, dass sie möglichst...
  - **vollständig**,
  - **klar**,
  - **konsistent**,
  - **minimal**,
  - und **bequem benutzbar** ist.

- Wie bei vielen Entwurfsrichtlinien gilt auch hier: diese Ziele können sich teilweise widersprechen.

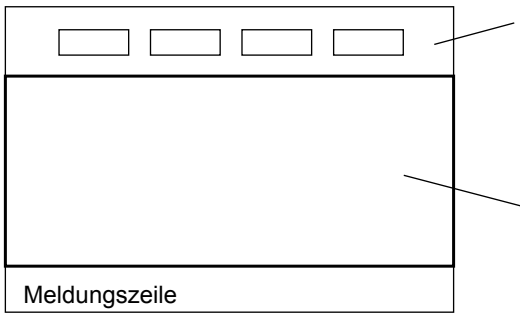
© Horstmann, Object-Oriented Design and Patterns, Wiley 2006

Konkretisierung: Verwende Interfaces

- Wir haben Interfaces als ein explizites Sprachkonstrukt von Java kennen gelernt, mit dem die Schnittstelle einer Klasse explizit modelliert werden kann.
- Häufig empfiehlt es sich, die Kopplung zwischen zwei Klassen zu verringern, indem ein Interface modelliert wird, das nur die Operationen zusammenfasst, die die eine Klasse von der anderen benutzt.
  - Erfüllt unter anderem die Forderung nach schmalen Schnittstellen.
  - Macht eine konkrete Kopplung deutlicher.
  - Kann helfen, statische Zyklen aufzulösen.
- Zentrales Zitat aus der Einleitung zu Gamma et al.:

  „Program to an interface, not an implementation.“

Beispiel: geschachtelte GUI



`Hauptfenster`, ein JFrame mit BorderLayout: JButtons in der Nord-Komponente, ein JLabel in der Südkomponente.

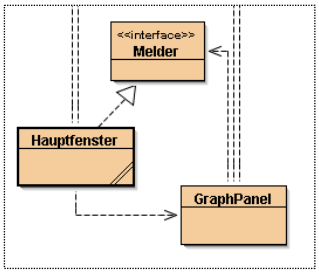
Als Center-Komponente: `GraphPanel`, ein spezieller JPanel, der Knoten eines Graphen darstellen kann.

Problem: `Hauptfenster` muss `GraphPanel` kennen, um beim Klick auf einen Button einen neuen Knoten auf dem `GraphPanel` zu erzeugen. Aber der `GraphPanel` muss auch das `Hauptfenster` kennen, weil Aktionen im Panel zu Meldungen in der Meldungszeile führen können. Was tun?

Lösung: Abhängigkeit per Interface explizit machen

- Das `GraphPanel` muss nicht die gesamte Schnittstelle des `Hauptfensters` kennen. Es benötigt lediglich eine Schnittstelle, über die Meldungen ausgegeben werden können.
- Das `Hauptfenster` implementiert deshalb ein Interface `Melder` mit einer Operation `melde`; wenn das `Hauptfenster` den `GraphPanel` erzeugt, übergibt es als Parameter eine Referenz auf sich selbst vom statischen Typ `Melder`.

- Der direkte Zyklus zwischen den beiden Klassen ist damit, zumindest statisch, aufgelöst.
- Zur Laufzeit besteht der Zyklus aber weiter, beide Objekte verfügen über eine Referenz auf das jeweils andere Objekt.



Entwurfsentscheidungen kapseln

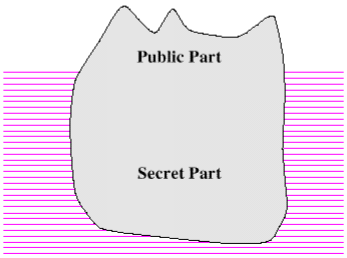
- Modellieren ist das Fällen von Entwurfsentscheidungen.
- Jede Entscheidung wirkt sich auf nachfolgende Entscheidungen aus.
- Es sollten deshalb bei der Modellierung die Entscheidungen **zuerst** getroffen werden, die sich mit **größter Wahrscheinlichkeit nicht ändern** werden (unveränderliche Kernabstraktionen eines Anwendungsbereichs).
- Entscheidungen hingegen, die sich mit **hoher Wahrscheinlichkeit ändern** können (wie die Art der Benutzungsschnittstelle, Technologien), sollten wir **kapseln** und damit **leichter austauschbar** halten.
- Wie identifizieren wir diese? Wir brauchen
  - ausreichend Informationen über den Kontext
  - Erfahrung
  - Augenmaß …

Geheimnisprinzip

Geheimnisprinzip: Nur relevante Merkmale einer Entwurfs- und Konstruktionseinheit sollten für Klienten sichtbar und zugänglich sein.

Details der Implementierung sollten verborgen bleiben.

Schon allein das Wissen über Interna kann missbraucht werden!



Das **Geheimnisprinzip** (Information Hiding) geht zurück auf die Arbeit von Parnas:
D. L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Vol 15:12, p.1053-1058, Dec. 1972.

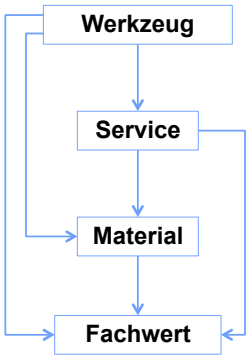
© Meyer

Zyklen vermeiden

- Klassen können in einer zyklischen Abhängigkeit zueinander stehen.
- Klassen-Zyklen haben etliche Nachteile:
  - Die beteiligten Klassen lassen sich **schlecht unabhängig** voneinander **testen**.
  - Die **Initialisierungsreihenfolge** der entstehenden Objekte ist möglicherweise **unklar**.
  - Es gibt **keinen** offensichtlichen **Einstiegspunkt**, um sich in den Entwurf einzulesen.
- In ähnlicher Weise gelten diese Aussagen für andere Einheiten des Softwareentwurfs (Pakete, Subsysteme).
- Zyklen sind oft nicht leicht zu erkennen.
- Generell gilt die Regel:
  Zyklische Abhängigkeiten sollten vermieden werden.

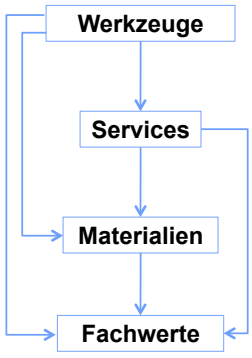
Beispiel: keine Paket-Zyklen durch Entwurfsregeln

Elementtypen:



- Die **SE2-Entwurfsregeln** benennen vier **Elementtypen**, aus denen sich ein interaktives System zusammensetzt, und legen erlaubte Benutzt-Beziehungen fest.
- Für SE2 empfehlen wir die Verwendung von entsprechend benannten **Paketen**: Werkzeuge, Services, Materialien und Fachwerte.
- Wenn die Entwurfsregeln in einem Softwaresystem auf Klassenebene eingehalten sind, dann sind sie transitiv **auch auf Paketebene eingehalten**.

Pakete:



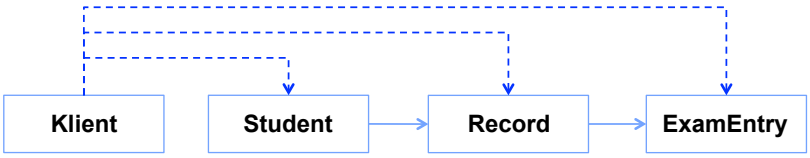
Law of Demeter, Original

- Ursprüngliche Definition (als **Law of Demeter for Functions/Methods**) war bereits lediglich eine **Entwurfsrichtlinie**:
  - Eine **Methode m** eines **Objektes o** sollte ausschließlich Methoden von folgenden Objekten aufrufen:
    - **von o selbst;**
    - **von Parametern von m;**
    - **von Objekten, die m erzeugt;**
    - **von Exemplarvariablen von o.**

- Umgangssprachlich: „Sprich nur mit Deinen engsten Freunden!“
- Hinweise auf Verletzungen sind lange Methodenaufrufketten in einer Zeile:
  - **student.getRecord().getExamEntry("SE2/2006").addResult(93);**

Lange Aufrufzeilen = hohe Kopplung

- **student.getRecord().getExamEntry("SE2/2011").addResult(93);**



- Wenn wir das Traversieren von Objektgeflechten in einer Methode ausimplementieren, **koppeln** wir die Methode zu stark an eine Struktur, deren Änderung dadurch erschwert wird.

Law of Demeter, heute

- Zeitgemäße Anpassungen:
  - „If you delegate, delegate fully."
  - „Don't message your delegate object's objects."
  - „Don't be aware of how some object works. Don't work at a lower level than is necessary."
  - **Tell, don't ask**.
- Auch beim Einhalten des LoD hilft immer wieder die Frage: **Wer sollte für diese Aufgabe zuständig sein?**

Law of Demeter, Beispiele

- Statt:

  student.getRecord().getExamEntry("SE2/2006").addResult(93);

  besser:

  student.scored(markedExam);

- **Nicht** einen Satelliten nach seinen Antriebsdüsen **fragen** und eine davon veranlassen, etwas mehr Gas zu geben; stattdessen dem Satelliten **sagen**, dass er sich neu ausrichten soll.

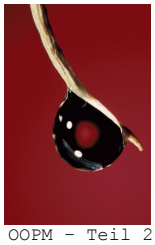
- **Getter** und **Setter** als Verstoß gegen das LoD:
  - Avoid thinking in terms of getting and setting variables: „I know you're in there, variable; come out with your hands up. If it wasn't for this pesky object-orientation that's has been foisted upon us and that I don't have the time or inclination to get to understand, I'd be able to read and write your state." (Deacon, Object-Oriented Analysis and Design, Addison-Wesley 2005)

Kohäsion, objektorientiert

- Mit **Kohäsion** (engl.: cohesion) bezeichnen wir den Grad (Anzahl und Verschiedenheit) der Aufgaben, die eine Softwareeinheit zu erfüllen hat.
- Wenn eine Einheit für genau eine logische Aufgabe zuständig ist, dann sprechen wir von **hoher Kohäsion** (engl.: high cohesion).
- Auf Ebene des Klassenentwurfs unterscheiden wir
  - Kohäsion von Methoden
  - Kohäsion von Klassen
- Wir streben nach **möglichst hoher Kohäsion**.

Bedeutung des Wortes Kohäsion in der Chemie und der Physik:

*„Molekulare Kraft zwischen Molekülen eines Stoffes. Sie bewirkt den **inneren Zusammenhalt** eines Stoffes."*



In der Linguistik:

*Verknüpfung von Textelementen (Sätze, Teilsätze, Redeeinheiten) zu einer **sinnvollen Einheit** auf der Oberfläche .*

Kohäsion von Methoden

- **Eine Methode sollte nur für genau eine wohldefinierte Aufgabe zuständig sein.**

```
public enum TrigoFunc {
    sin, cos, tan, arctan, ...
}
...
public double trigo(TrigoFunc func,
                    double x) ...
```

Methode mit niedriger Kohäsion

Methoden mit hoher Kohäsion

```
public double sin(double x) ...
public double cos(double x) ...
public double tan(double x) ...
public double arctan(double x) ...
...
```


Kohäsion von Klassen

- **Eine Klasse (bzw. jedes ihrer Exemplare) sollte genau eine klar umrissene Einheit repräsentieren.**

```
public class Program {
    public void initializeCommandStack()…
    void pushCommand(Command command)…
    Command popCommand()…
    void shutdownCommandStack()…
    void InitializeReportFormatting()
    void formatReport(Report report)…
    void printReport(Report report)…
    void InitializeGlobalData()…
    void ShutdownGlobalData()…
}
```

Klasse mit niedriger Kohäsion

Klasse mit hoher Kohäsion

```
public class Film {
    public Film(String titel,
                int laenge,
                FSK fsk,
                boolean ueberlaenge)…
    public String gibTitel()…
    public int gibLaenge()…
    public FSK gibFSK()…
    public boolean hatUeberlaenge()…
}
```

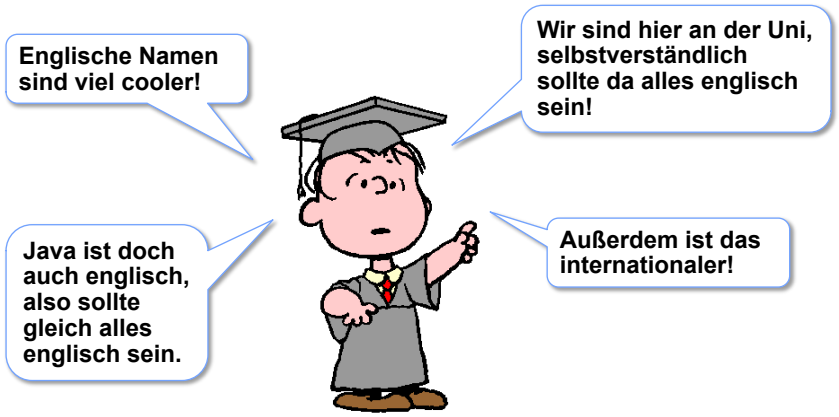

Geeignete Bezeichner wählen

- Hohe Kohäsion zeigt sich unter anderem daran, dass geeignete Bezeichner für Entwurfseinheiten gewählt werden können.
  - Eine **konkrete Klasse** sollte durch ein **treffendes Substantiv** benennbar sein.
  - Eine **Methode** sollte durch ein **treffendes Verb** benennbar sein.
- Situationen, in denen eine geeignete Wahl schwer fällt, lassen häufig auf einen schlechten Entwurf schließen.
- Insbesondere **und-Bezeichner** (wie „einfuegenUndBerechnen“ oder „StarterUndVermittler“) sind ein deutlicher Hinweis auf **niedrige Kohäsion**.

Extreme Ausprägung: Schreibe Quelltext so, dass der Kunde ihn auch lesen kann!

Englische oder deutsche Begriffe im Quelltext?

- Eine immer wieder gestellte Frage: Sollen wir unsere **Klassen** und **Methoden mit deutschen oder mit englischen Namen** versehen?



Englisch oder Deutsch: kontextabhängig!

- Es gibt keine abschließende Antwort auf diese Frage; sie muss **für jedes Projekt neu** beantwortet werden.
- Eine wichtige Entscheidungshilfe: die **Trennung von fachlichen und technischen Klassen**.
  - Technische Klassen sollten mit englischen Namen benannt werden (Frame, Button, Listener, Writer, Transaction, …).
  - Für die fachlichen Klassen sollte die Sprache abhängig vom fachlichen Kontext gewählt werden:
    - Bei einer Versicherung, die alle fachlichen Begriffe auf Deutsch verwendet, wären englische Begriffe sehr aufgesetzt und würden sehr schnell „unscharf“.
    - In einem Forschungsprojekt über Robotik mit internationaler Beteiligung sollten sicher auch die fachlichen Klassen englisch benannt werden.

Deutsch und Englisch gemischt?

- Durch einen deutschsprachigen fachlichen Kontext kann es zu Bezeichnern kommen, die sich aus einem deutschen und einem englischen Teilbegriff zusammensetzen.
- Das kann man als hässlich empfinden; es ist aber tatsächlich eher eine Chance als ein Nachteil, weil dann sogar innerhalb von Bezeichnern fachlicher und technischer Anteil sauber unterschieden sind.

- Beispiele aus der Mediathek:
  - `BearbeitenButton`: technisch eine Schaltfläche, die fachlich das Bearbeiten eines Mediums ermöglicht.
  - `getMedienart`: technisch ein Getter, der eine fachliche Eigenschaft auslesbar macht.
  - `RückgabeUI`: technisch zuständig für das User Interface, fachlich zuständig für die Rückgabe von Medien.

Konkretisierung: möglichst genaue Bezeichner wählen

- Beobachtung bei Systementwürfen:
  - **Verschmelzen ist leichter als Aufteilen**.
  - Es ist leichter, zwei verschiedene Begriffe zusammenzuführen (etwa, weil sich ihre Differenzierung als unnötig erweist), als einen Begriff in zwei neue aufzuspalten.
- Beispiel: Software für einen **CD-Verleih**.
  - Wir modellieren zu Beginn als zentrale Abstraktion die **Klasse CD**.
  - Später stellen wir fest: Wir müssen die Modellierung eines Albums unterscheiden von der Modellierung eines konkretes Exemplars dieses Albums, das ausgeliehen werden kann (denn es kann mehrere verleihbare Exemplare eines Albums geben).
  - Frage: Was wären geeignete Namen für die beiden Abstraktionen?

Code-Duplizierung vermeiden

- Wenn ein Stück Quelltext in identischer Form an mehreren Stellen eines Systems definiert ist, sprechen wir von **Code-Duplizierung**.
- Code-Duplizierung ist problematisch, weil
  - üblicherweise an einer Stelle nicht erkennbar ist, an welchen anderen Stellen derselbe Quelltext erscheint, und
  - Änderungen an einem der Duplikate eventuell auch an allen anderen Duplikaten ausgeführt werden müssen; dies kann bei der Wartung übersehen werden (implizite Kopplung).
- Code-Duplizierung ist auch ein **Zeichen niedriger Kohäsion**:
  - Wenn zwei Einheiten dieselbe (Teil-)Aufgabe erledigen, ist bei mindestens einer von beiden die Zuständigkeit falsch zugeordnet.

Große Einheiten vermeiden

- Ein bekanntes Anti-Pattern - die „**Gott-Klasse**“:
  - Sie ist für alles zuständig und hält 90% des Quelltextes.
- Methoden, die **mehrere Bildschirmseiten** lang sind, sind schlecht wartbar.
- Klassen mit mehreren hundert Methoden oder Exemplarvariablen sind meist zu groß!
- Wenn einzelne Einheiten zu groß werden, ist dies ein Hinweis auf **niedrige Kohäsion**:
  - Wenn eine Einheit **sehr viele** Aufgaben erfüllt, erfüllt sie vermutlich **zu viele** Aufgaben.

Zum Vertiefen: die SOLID-Prinzipien

- **Robert C. Martin** hat Anfang der 2000er Jahre zentrale Entwurfsprinzipien für objektorientierte Systeme kompakt unter dem Begriff **SOLID** zusammengefasst.
- **SOLID** ist ein Akronym, das fünf Prinzipien des objektorientierten Entwurfs umfasst:
  - **S**RP: Single Responsibility Principle
  - **O**CP: Open Closed Principle
  - **L**SP: Liskov Substitution Principle
  - **I**SP: Interface Segregation Principle
  - **D**IP: Dependency Inversion Principle
- Auch diese Prinzipien basieren in ihrem Kern auf dem Streben nach **niedriger Kopplung** und **hoher Kohäsion**.

Single Responsibility Principle (SRP)

- Jede Klasse soll nur eine fest definierte Aufgabe erfüllen
- „There should never be more than one reason for a class to change.“
- **Gegenbeispiel** (Verletzung des SRP):

```
interface Modem  {
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

- Das Modem implementiert zwei Zuständigkeiten:
  - **Connection Management (dial und hangup)**
  - **Data Communication (send und receive)**
- Die zwei Zuständigkeiten können sich aus unterschiedlichen Motiven ändern.

Open Closed Principle (OCP)

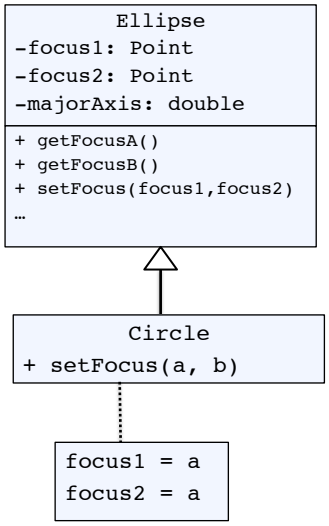
- „A module should be open for extension but closed for modification“
- Komponenten sollen offen für Erweiterungen sein, aber geschlossen für Veränderungen
- Komponenten sollen „wie ausgeliefert“ benutzt werden können, trotzdem erweiterbar sein
- Realisiert mit Polymorphie
- Zukünftige Erweiterungen sollen ohne Änderung des bestehenden Quellcodes möglich sein
- Nur Abhängigkeit von Abstraktionen erlauben, nicht Abhängigkeit von konkreten Implementierungen

Liskov Substitution Principle (LSP)

- „Subclasses should be substitutable for their base classes“
- Unterklassen erfüllen alle Verträge ihrer Oberklassen, überschriebene Methoden besitzen keine stärkeren Vorbedingungen und keine schwächeren Nachbedingungen
- Unterklassen sollen nicht mehr erwarten und nicht weniger liefern als ihre Oberklassen

Kreis-Ellipsen-Problem:

```
public void myMethod (Ellipse e) {
   Point p1 = new Point (1,1);
   Point p2 = new Point (2,2);
   e.setFocus (p1,p2);
   assert (e.getFocusB() == p2);
   // Fehler beim Kreis!}
```



SE2 – OOPM – Teil 2 39

Interface Segregation Principle (ISP)

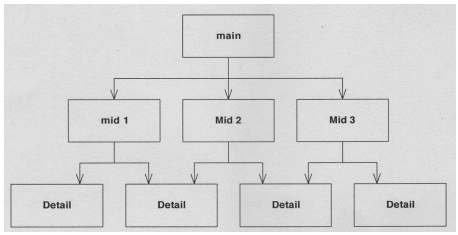
- **„Many client specific interfaces are better than one general purpose interface“**
- Wenn eine Klasse verschiedene Klienten hat, sollten die Methoden für den jeweiligen Klienten in ein entsprechendes Interface extrahiert werden
- Wenn ein oder zwei Klient-Typen dieselbe Methode verwenden, sollte die Methode zu beiden Interfaces hinzugefügt werden.

SE2 – OOPM – Teil 2 40

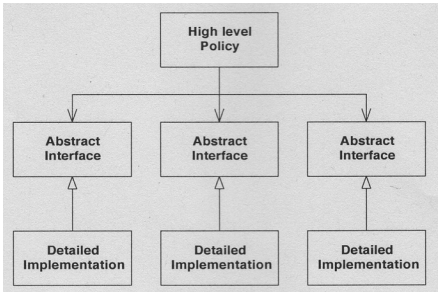
Dependency Inversion Principle (DiP)

- „Depend upon abstractions. Do not depend upon concretions.“
- Es sollte keine Abhängigkeit zu konkreten Klassen geben

prozedural



OO-Design



Inversion of Control (IoC)

- Hollywood-Principle:
  „Don‘t call us, we‘ll call you“
- Kontrollfluss geht vom Framework aus
  - Framework ruft eine Methode des Systems auf, nicht umgekehrt.
- Beispiel:
  - ActionListener in Swing („actionPerformed“)

Dependency Injection (DI)

- *Ward Cunningham:*
  „DI is a key element of agile architecture“

- Bei DI werden Abhängigkeiten von außen gesteuert.
  - Interface Injection (Type 1 DI)
  - Setter Injection (Type 2 DI)
  - Constructor Injection (Type 3 DI)

Type 3 DI: Construktor Injection

- Am Konstruktor werden benötigte Objekte mitgegeben:

```
public class MovieLister
{
    private final MovieFinder _finder;
    public MovieLister(MovieFinder finder)
    {
        _finder = finder;
    }
    ...
}
```


Type 2 DI: Setter Injection

- An der Klasse sind Setter-Methoden um die benötigten Objekte zu setzen:

```
public class MovieLister
{
    private MovieFinder _finder;
    public void setMovieFinder(MovieFinder finder)
    {
        _finder = finder;
    }
    ...
}
```


Type 1 DI: Interface Injection

- Es wird ein Interface definiert, um die Injection zu kennzeichnen:

```
public interface InjectFinder
{
    void injectFinder(MovieFinder finder);
}
```

- Die entsprechenden abhängigen Klassen implementieren das Interface:

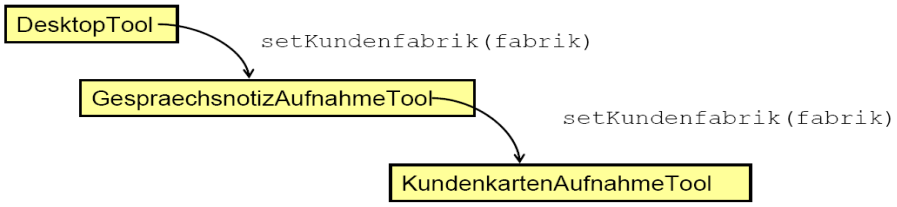
```
public class MovieLister implements InjectFinder
{
    public void injectFinder(MovieFinder finder)
    {
        this.finder = finder;
    }
}
```


Folge…

Ein Objekt bekommt alles gesetzt, was es braucht - z.B. im Konstruktor oder per Setter

Probleme:

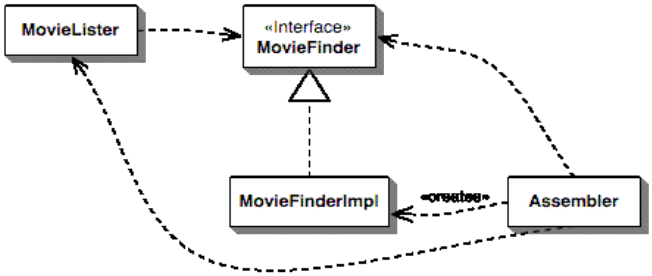
- Das erzeugende Objekte muss alle Objekte kennen, die das erzeugte Objekt braucht
- Das führt zum „Durchschleifen“ von Parametern und zu langen Parameterlisten!



SE2 – OOPM – Teil 2 47

Abhilfe: Dependency Injection Framework

- Die Erzeugung fast aller Objekte wird von einer anderen Instanz erledigt
- Diese Instanz kennt alle benötigten Objekte oder weiß, wie diese erzeugt werden können



SE2 – OOPM – Teil 2 48

Entwurfsmuster um

- Kopplung bei Erzeugung aufzuheben:
  - **Factory** - Schnittstelle zum Erzeugen verwandter Objekte
- Schnittstellen anzupassen:
  - **Adapter** – passt die Schnittstelle einer Komponente an Client an
  - **Bridge** - Entkopple eine Abstraktion von ihrer Implementierung
  - **Fassade** – einheitliche Schnittstelle zu Menge von Schnittstellen
- Entfernte Aufrufe zu kapseln:
  - **Proxy** – vorgelagertes Stellvertreterobjekt
- Komponenten dynamisch und transparent erweitern:
  - **Decorator** - Erweitere ein Objekt dynamisch um Dienstleistungen
- Benachrichtigen ohne den Adressaten zu kennen:
  - **Observer** - 1-zu-n Beziehung zwischen Objekten, so dass die Änderung des Zustands eines Objekts anonyme an n andere übermittelt wird

Factory (Abstrakte Fabrik)

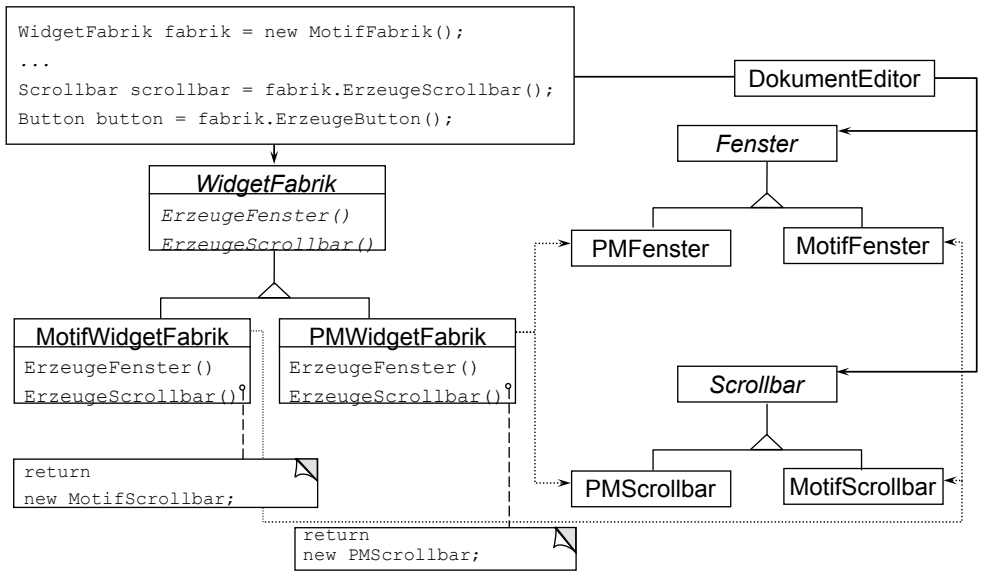
Zweck

"Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen."

Anwendbarkeit

- Ein System (Klassenbibliothek, Rahmenwerk) soll unabhängig davon entwickelt werden, wie die Objekte, die es manipuliert, erzeugt, zusammengesetzt und repräsentiert werden.
- Ein System soll mit einer von mehreren Produktfamilien konfiguriert werden.
- Eine Menge verwandter Produkte wurde entworfen, um zusammen verwendet zu werden, und diese Konsistenzbedingung soll gewahrt werden.
- Es soll eine Klassenbibliothek von Produkten erstellt werden, von denen nur die Schnittstellen, aber nicht ihre Implementierungen offengelegt werden.

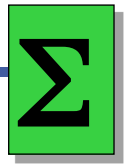
Das Muster „Factory“ (Abstrakte Fabrik)



Diskussion der Factory (Abstrakte Fabrik)

- Der Typ einer konkreten Fabrik erscheint nur einmal in einer Anwendung - dort, wo von ihr ein Exemplar erzeugt wird. Dies macht es einfach, die von einer Anwendung benutzte konkrete Fabrik auszutauschen.
- Da die abstrakte Fabrik eine komplette Familie von Produkten erzeugt, wird dabei die gesamte Produktfamilie auf einmal ausgetauscht.
- Wenn Produkte einer Familie entworfen werden, um zusammenzuarbeiten, und es wichtig ist, daß die Anwendung nur Objekte einer Familie zur Zeit verwendet, kann dies durch eine abstrakte Fabrik leicht sichergestellt werden.
- Das Hinzufügen neuer Produkte zu abstrakten Fabriken erweist sich als schwierig, da hierzu die AbstrakteFabrik-Klasse samt aller Unterklassen schnittstellenmäßig erweitert werden muß.

Zusammenfassung Klassenentwurf



- Die **innere Qualität** eines Softwaresystems bestimmt seine **Wartbarkeit** und die **Portierbarkeit**.
- Die innere Qualität eines **objektorientierten Systems** wird maßgeblich durch den **Klassenentwurf** bestimmt.
- Zahlreiche **Richtlinien** helfen uns, Entwurfsentscheidungen zu treffen.
- Ein Klassenentwurf sollte insbesondere
  - eine **hohe Kohäsion** seiner Bestandteile
  - und eine **niedrige Kopplung** zwischen diesen aufweisen.

