

# Entwurf

Prof. Sven Apel

Universität des Saarlandes



# Teil III

## Entwurfsprinzipien

# Problem

Software lebt oft sehr viel länger als ursprünglich geplant

Software muss beständig angepasst werden

Anteil Wartungskosten 50–80%

Hauptziel: Software *änderbar* und *wiederverwendbar* machen – bei geringem Risiko und niedrigen Kosten.

# Grundprinzipien

des objektorientierten Modells

Abstraktion

Einkapselung

Modularität

Hierarchie

Ziel: *Änderbarkeit* + *Wiederverwendbarkeit*

# Grundprinzipien

des objektorientierten Modells

Abstraktion

Einkapselung

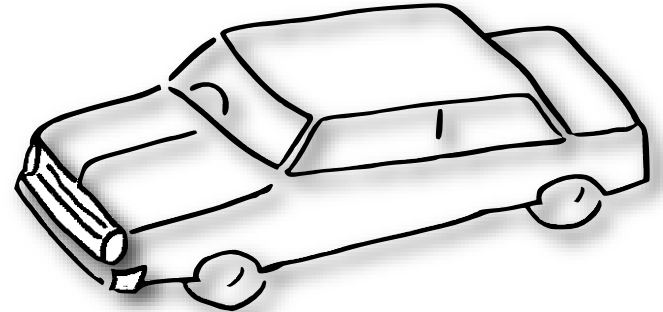
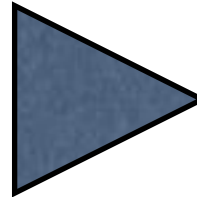
Modularität

Hierarchie

# Abstraktion



Konkretes Objekt



Allgemeines Prinzip

# Abstraktion...

Hebt *gemeinsame* Eigenschaften von Objekten hervor

Unterscheidet zwischen *wichtigen* und *unwichtigen* Eigenschaften

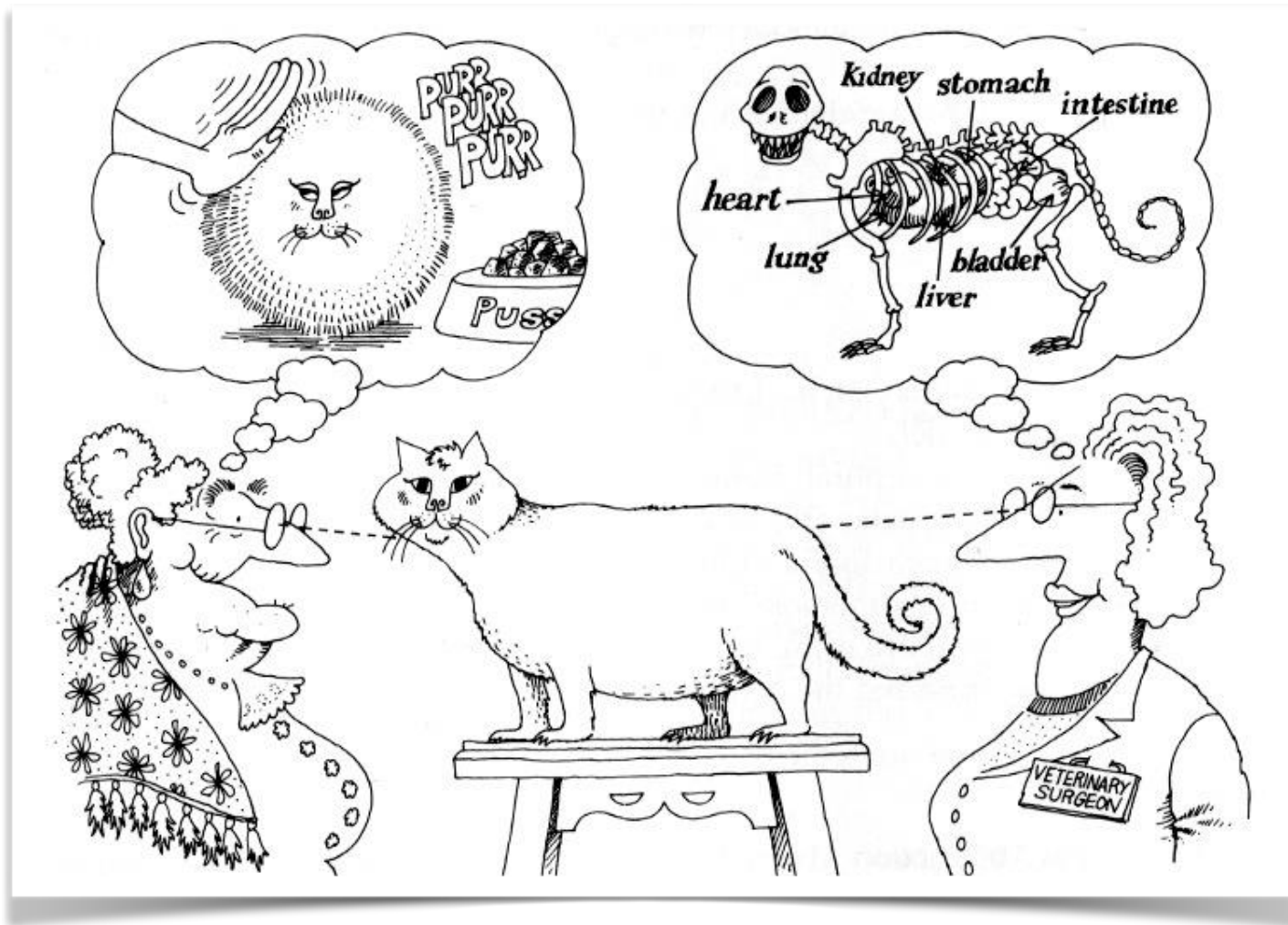
Muss unabhängig von Objekten verstanden werden können

# Abstraktion

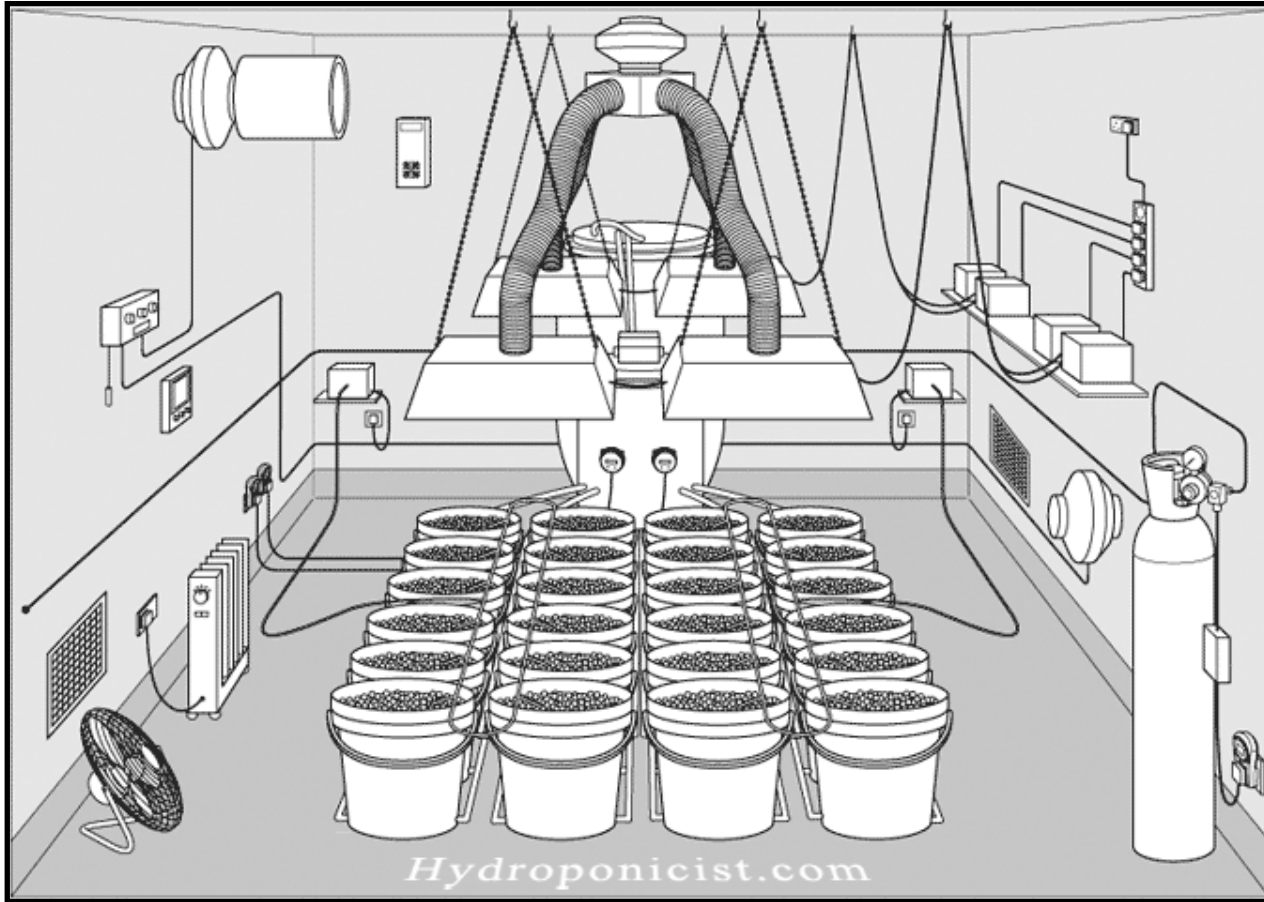
“An abstraction denotes the *essential characteristics* of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.”



# Blickwinkel



# Beispiel: Sensoren



# Ingenieurslösung

```
void check_temperature() {  
    short *sensor = 0x80004000;  
    short low = sensor[0x20];  
    short high = sensor[0x21];  
    int temp_celsius = low + high * 256;  
    if (temp_celsius > 50) {  
        turn_heating_off()  
    }  
}
```

# Abstrakte Lösung

Irrelevante Details  
werden ausgeblendet

```
typedef float Temperature;  
typedef int Location;  
  
class TemperatureSensor {  
public:  
    TemperatureSensor(Location);  
    ~TemperatureSensor();  
  
    void calibrate(Temperature actual);  
    Temperature currentTemperature() const;  
    Location location() const;  
  
private: ...  
}
```

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung

Modularität

Hierarchie

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung

Modularität

Hierarchie

# Einkapselung

Kein Teil eines komplexen Systems soll von internen Details eines anderen *abhängen*

Ziel: Änderungen vereinfachen

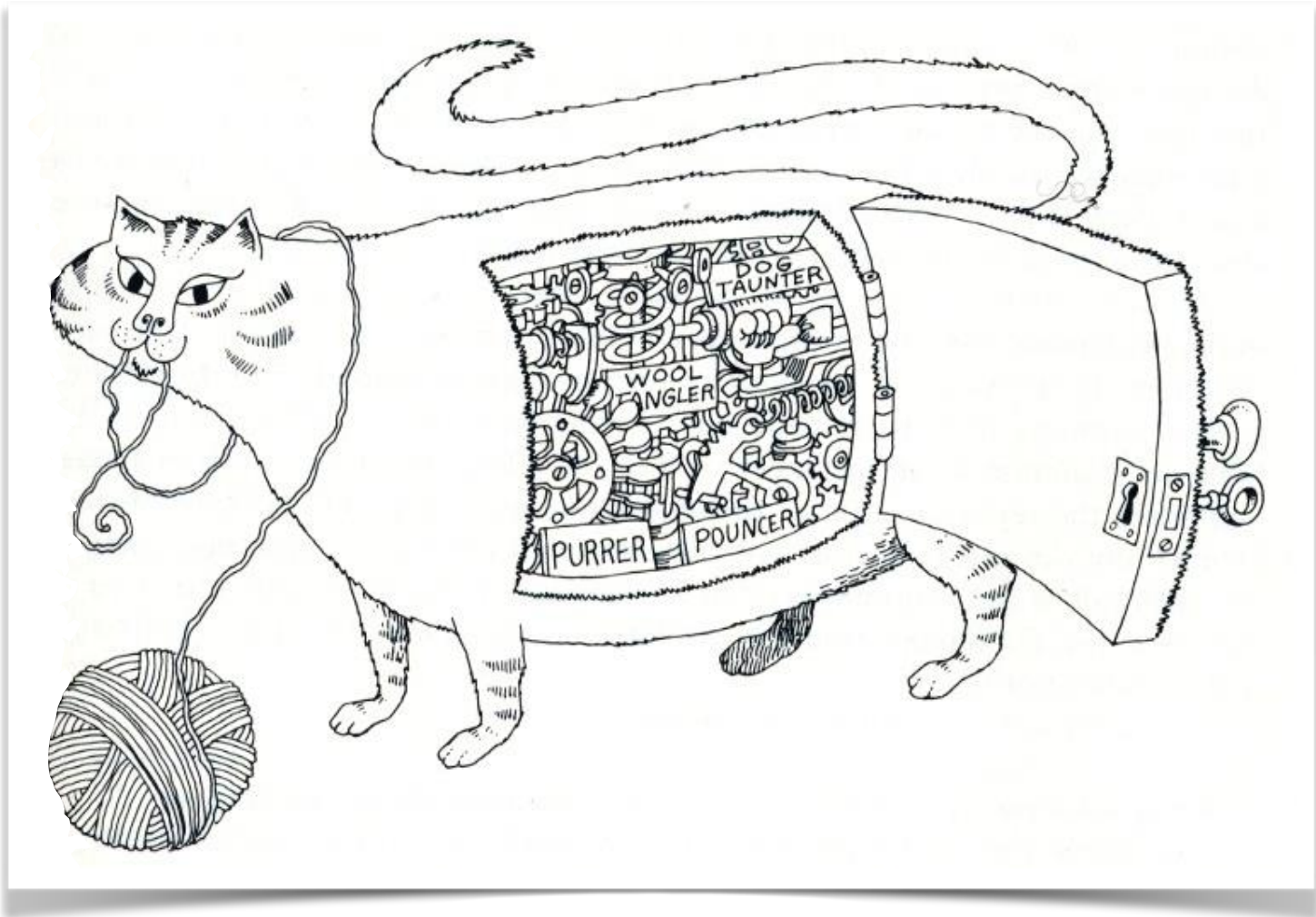
*Geheimnisprinzip*: Die internen Details (Zustand, Struktur, Verhalten) werden zum *Geheimnis* eines Objekts

# Einkapselung

“Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and its behavior; encapsulation serves to *separate* the *contractual interface* of an abstraction and its *implementation*.”




# Einkapselung



# Aktiver Sensor

```
class ActiveSensor {  
public:  
    ActiveSensor(Location)  
    ~ActiveSensor();  
  
    void calibrate(Temperature actual);  
    Temperature currentTemperature() const;  
    Location location() const;  
  
    void register(void (*callback)(ActiveSensor *));  
  
private: ...  
}
```

wird aufgerufen,  
wenn sich  
Temperatur ändert



Verwaltung der Callbacks ist *Geheimnis* des Sensors

# Antizipation des Wandels

Eigenschaften, die sich vorhersehbar ändern, sollten in *spezifischen Komponenten* isoliert werden.

```
class Employee {  
    String name;  
}
```



```
class Employee {  
    Name name;  
}
```

```
class Name {  
    String first, middle, last;  
    boolean equals(Name name) { ... }  
}
```

# Verschiedene Varianten

```
class Stack {  
    private LinkedList store;  
    void push(Object o) { ... }  
    Object pop() { ... }  
}
```

```
class Stack {  
    private ArrayList store;  
    void push(Object o) { ... }  
    Object pop() { ... }  
}
```

```
class Stack {  
    private HashTree store;  
    void push(Object o) { ... }  
    Object pop() { ... }  
}
```

```
class Stack {  
    private Object[] store;  
    void push(Object o) { ... }  
    Object pop() { ... }  
}
```

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung – Änderungen lokal halten

Modularität

Hierarchie

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung – Änderungen lokal halten

Modularität

Hierarchie

# Modularität

Grundidee: Teile isoliert betrachten, um Komplexität in den Griff zu bekommen (*“Teile und Herrsche”*)

Programm wird in *Module* aufgeteilt, die speziellen Aufgaben zugeordnet sind

Module sollen *unabhängig* von anderen geändert und wiederverwendet werden können

# Modularität





# Modularität

“Modularity is the property of a system that has been decomposed into a set of *cohesive* and *loosely coupled* modules.”

# Modul-Balance

Ziel 1: Module sollen *Geheimnisprinzip* befolgen – und so wenig nach außen dringen lassen, wie möglich

Ziel 2: Module sollen *zusammenarbeiten* – und hierfür müssen sie Informationen austauschen

Diese Ziele stehen im *Konflikt* zueinander

# Modul-Prinzipien

Hohe Kohäsion

Schwache Kopplung

Demeter-Prinzip

# Hohe Kohäsion

Module sollen *logisch zusammengehörige* Funktionen enthalten

Erreicht man durch Zusammenfassen der Funktionen, die auf denselben *Daten* arbeiten

# Schwache Kopplung

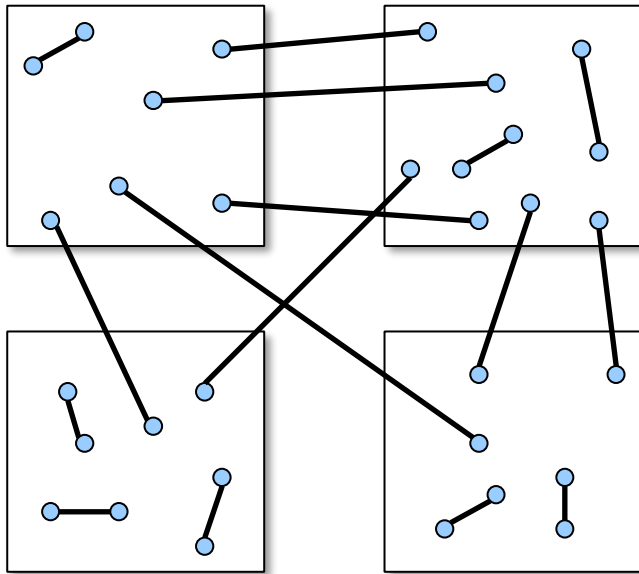
Änderungen in Modulen sollen sich nicht auf andere Module auswirken

Erreicht man durch

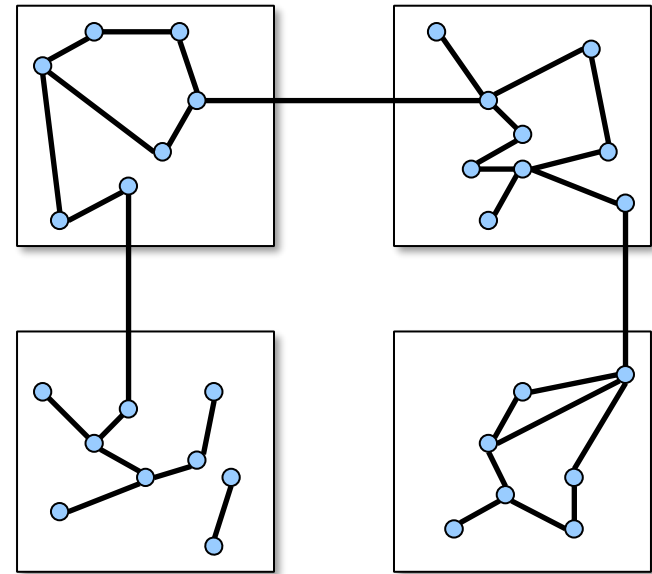
- Geheimnisprinzip

- Abhängigkeit zu möglichst wenig Modulen

Geringe Kohäsion  
Starke Kopplung



Hohe Kohäsion  
Schwache Kopplung

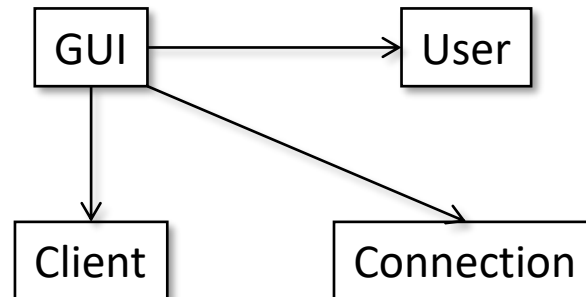


# Beispiel: Starke Kopplung

GUI ist unnötigerweise von Connection abhängig!

```
// Display friend for a particular user

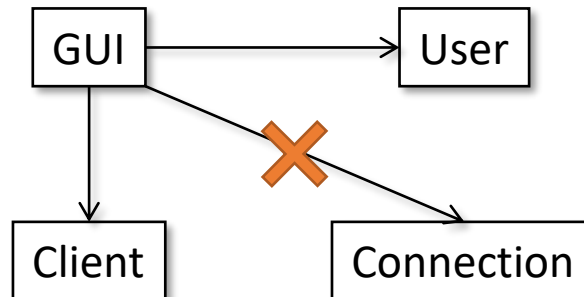
class GUI {
    void displayFriends(User user, Client client) {
        Friend[] friends = (Friend[]) client.getConnection().Execute("getFriends:" + user.Name);
        ...
    }
}
```



# Beispiel: Entkopplung

```
// Display friend for a particular user

class GUI {
    void displayFriends(User user, Client client) {
        Friend[] friends = client.getFriends(user);
        ...
    }
}
```





# Demeter-Prinzip



Grundidee: So wenig wie möglich über Objekte und ihre Struktur annehmen

Verfahren:

Methodenaufrufe auf *Freunde* beschränken

"Write shy code modules that don't reveal anything unnecessary to other modules and that don't rely on other modules' implementations."

– Dave Thomas



# Zugelassene Aufrufe

Eine Methode sollte nur Methoden folgender Objekte aufrufen:

1. des eigenen Objekts
2. ihrer Parameter
3. erzeugter Objekte
4. direkte Teile des eigenen Objekts



*“single dot rule”*

# Demeter-Prinzip → Entkopplung

```
// Display friend for a particular user

class GUI {
    void displayFriends(User user, Client client) {
        Friend[] friends = (Friend[]) client.GetConnection().Execute("getFriends:" + user.Name);
        ...
    }
}
```



```
// Display friend for a particular user

class GUI {
    void displayFriends(User user, Client client) {
        Friend[] friends = client.getFriends(user);
        ...
    }
}
```

# Demeter-Prinzip

Reduziert Kopplung zwischen Modulen

Verhindert direkten Zugang zu Teilen

Beschränkt verwendbare Klassen

Verringert Abhängigkeiten

Sorgt für zahlreiche neue Methoden

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung – Änderungen lokal halten

Modularität – Informationsfluss kontrollieren

Hohe Kohäsion • Schwache Kopplung • nur mit Freunden reden

Hierarchie

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung – Änderungen lokal halten

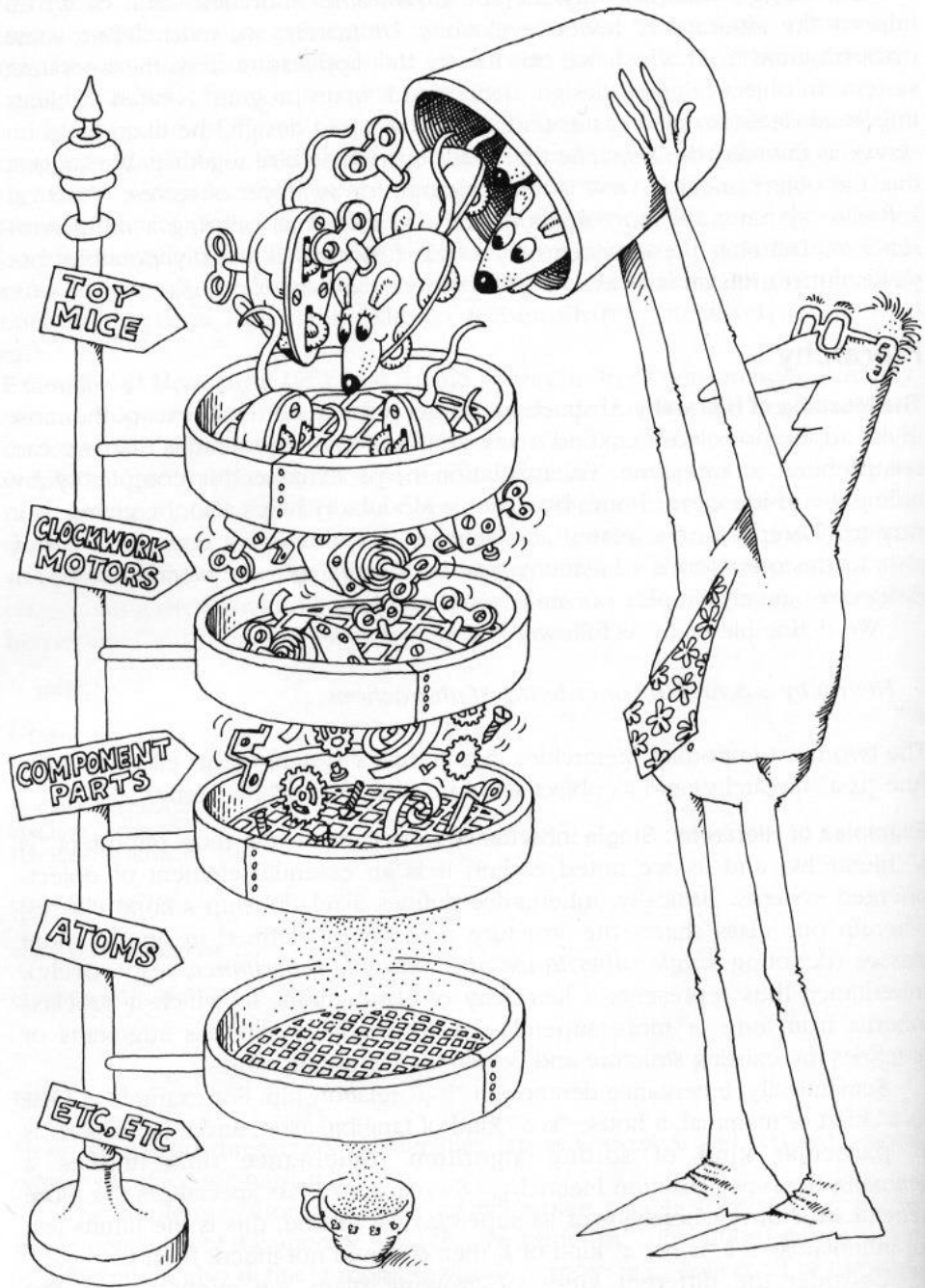
Modularität – Informationsfluss kontrollieren

Hohe Kohäsion • Schwache Kopplung • nur mit Freunden reden

Hierarchie

# Hierarchie

“Hierarchy is a ranking or ordering of abstractions.”





# Wichtige Hierarchien

“hat”-Hierarchie –

*Aggregation* von Abstraktionen

Ein *Auto* hat drei bis vier *Räder*

“ist-ein”-Hierarchie –

*Verallgemeinerung* über Abstraktionen

Ein *ActiveSensor* ist ein *TemperatureSensor*

# Wichtige Hierarchien

“hat”-Hierarchie –

*Aggregation* von Abstraktionen

Ein *Auto* hat drei bis vier *Räder*

“ist-ein”-Hierarchie –

*Verallgemeinerung* über Abstraktionen

Ein *ActiveSensor* ist ein *TemperatureSensor*

# Hierarchie-Prinzipien

Open/Closed-Prinzip

Liskov-Prinzip

Abhängigkeits-Prinzip

# Open/Closed-Prinzip

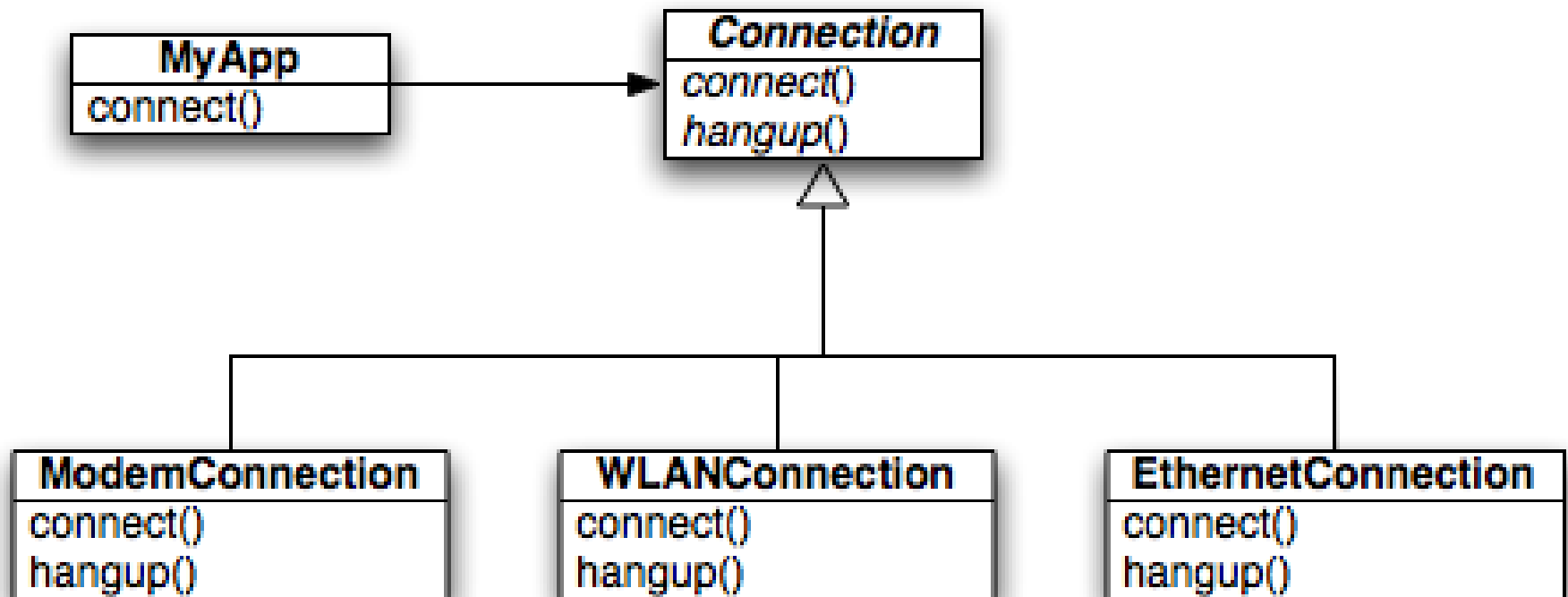
Eine Klasse sollte *offen* für Erweiterungen, aber *geschlossen* für Änderungen sein

Wird durch *Vererbung* und *dynamische Bindung* erzielt

# Ein Internet-Anschluss

```
void connect() {  
    if (connection_type == MODEM_56K)  
    {  
        Modem modem = new Modem();  
        modem.connect();  
    }  
    else if (connection_type == ETHERNET) ...  
    else if (connection_type == WLAN) ...  
    else if (connection_type == UMTS) ...  
}
```

# Lösung mit Hierarchien



# Liskov-Prinzip

Eine Unterklasse sollte stets an Stelle der Oberklasse treten können:

- Gleiche oder *schwächere* Vorbedingungen

- Gleiche oder *stärkere* Nachbedingungen

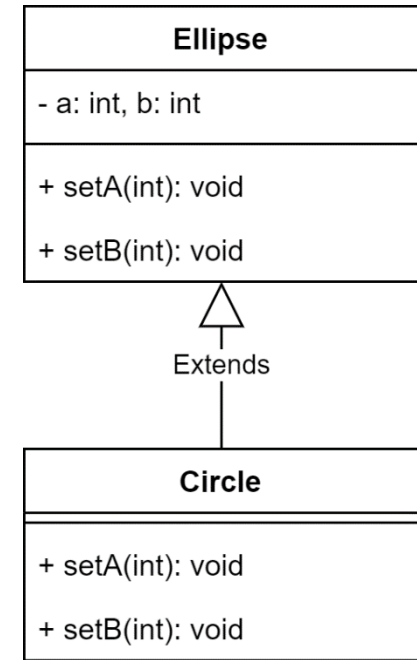
Abgeleitete Methoden sollten *nicht mehr erwarten* und *nicht weniger liefern*.

# Kreis vs. Ellipse

Jeder Kreis ist eine Ellipse

Ist diese Hierarchie sinnvoll?

Nein, da ein Kreis *mehr erwartet* und *weniger liefert*



Was ist wenn man die Größen der Halbachsen eines Kreises separat verändern will, wie bei einer Ellipse?

1. Entweder man setzt nur eine Halbachse → *kein Kreis mehr*
2. Oder man setzt beide gleich → *Kreis verhält sich nicht mehr wie eine Ellipse*



# Kreis vs. Ellipse

Jeder Kreis

Ist diese Hierarchie

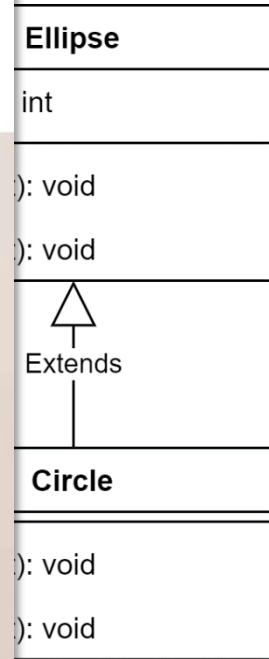
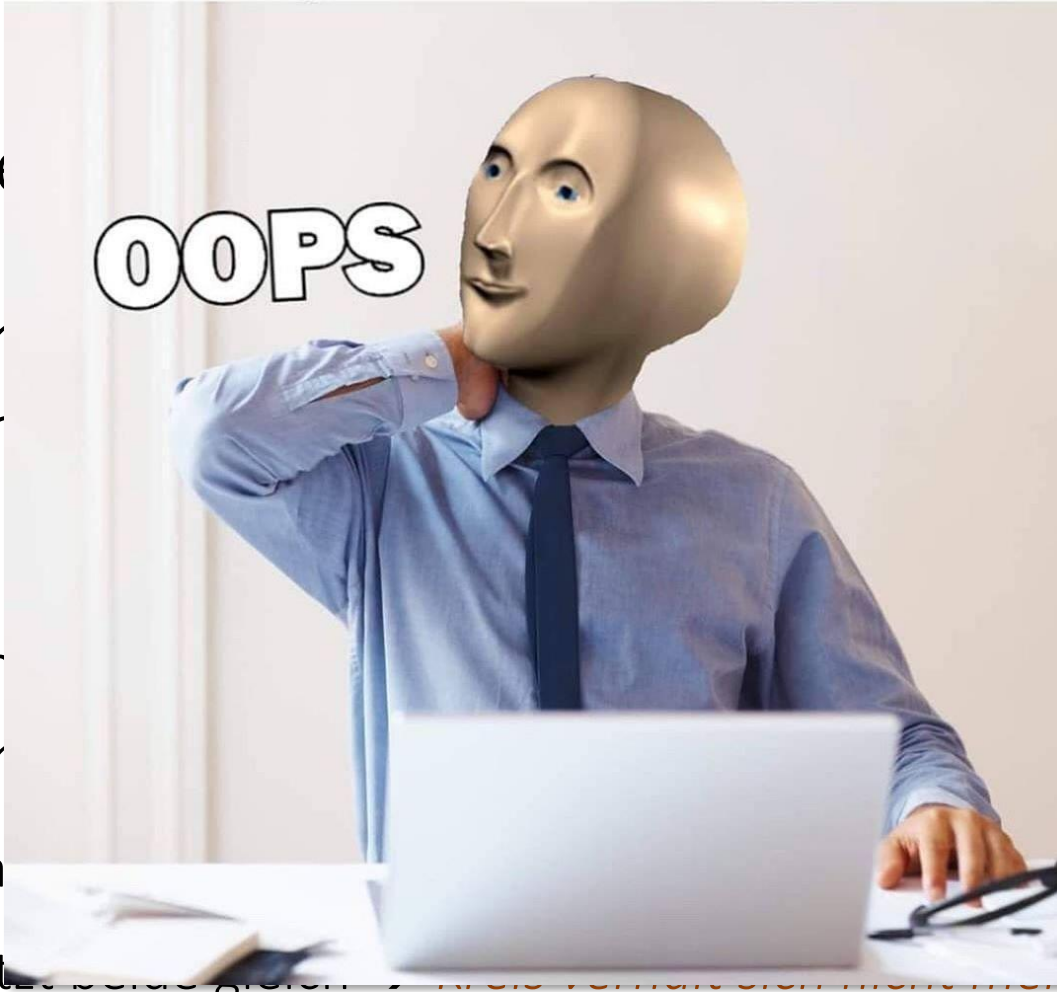
Nein, da eine Ellipse *erwartet* und

Was ist wenn man sie separat verändert?

1. Entweder man setzt beide gleich
2. Oder man setzt beide gleich

When class Cat extends Table because it has four legs.

OOPS



des Kreises

*wie eine Ellipse*

# Abhängigkeits-Prinzip

Abhängigkeiten sollten nur zu *Abstraktionen* bestehen – nie zu konkreten Unterklassen  
(*dependency inversion principle*)

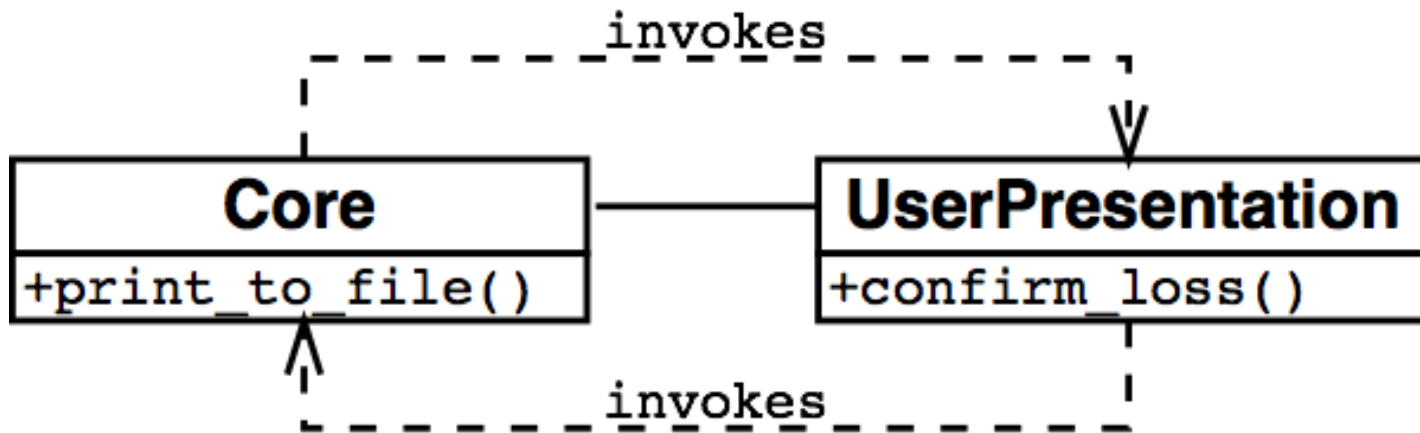
Dieses Prinzip kann gezielt eingesetzt werden,  
um Abhängigkeiten zu brechen

# Abhängigkeit

```
// Print current Web page to FILENAME.  
void print_to_file(string filename)  
{  
    if (path_exists(filename))  
    {  
        // FILENAME exists;  
        // ask user to confirm overwrite  
        bool confirmed = confirm_loss(filename);  
        if (!confirmed)  
            return;  
    }  
  
    // Proceed printing to FILENAME  
    ...  
}
```

# Zyklische Abhängigkeit

Konstruktion, Test, Wiederverwendung einzelner Module werden unmöglich!

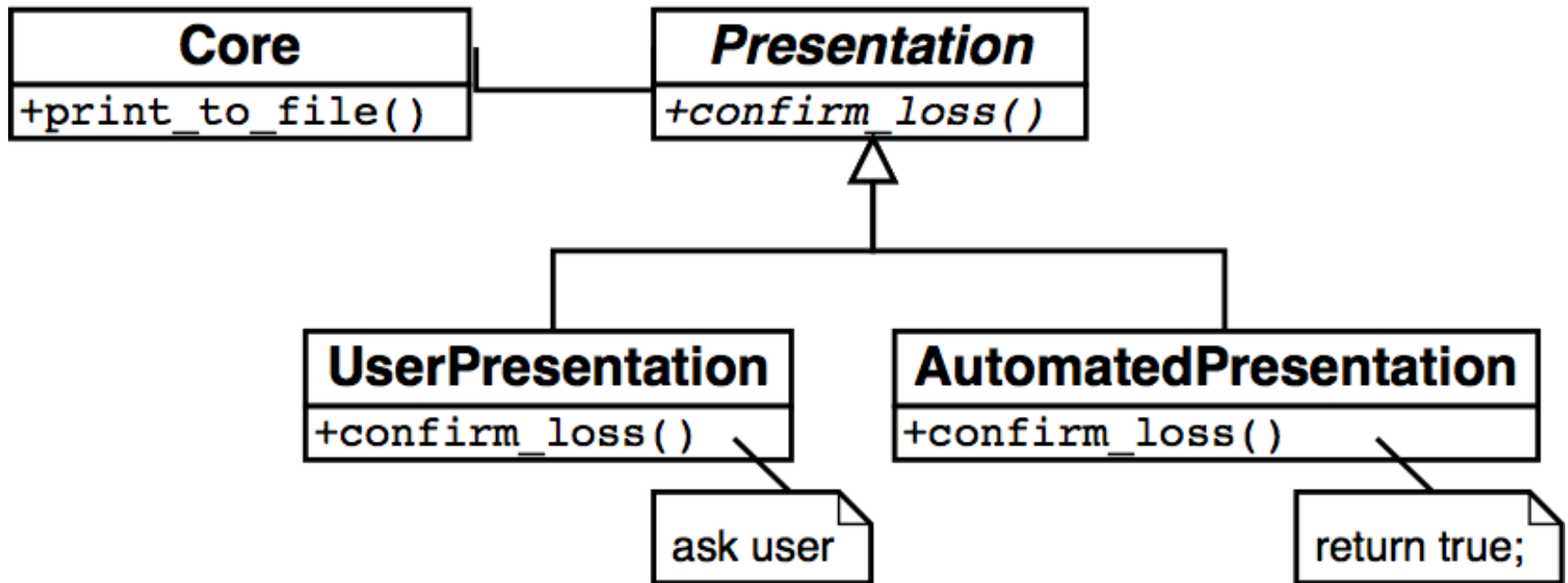


# Abhängigkeit

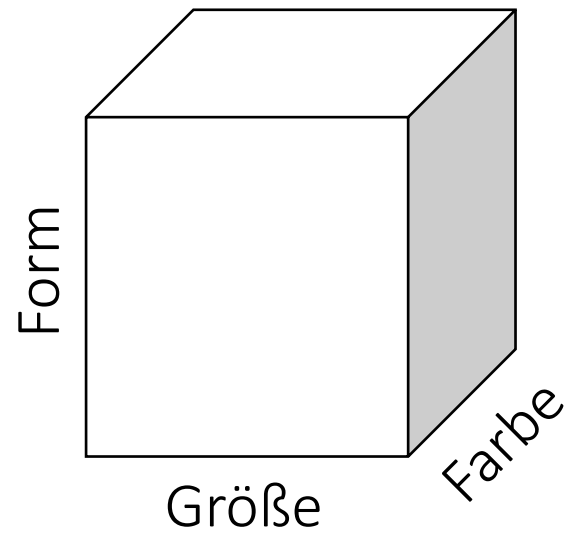
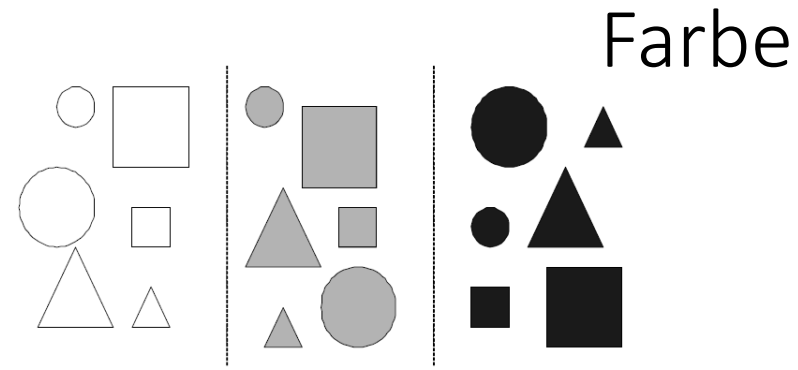
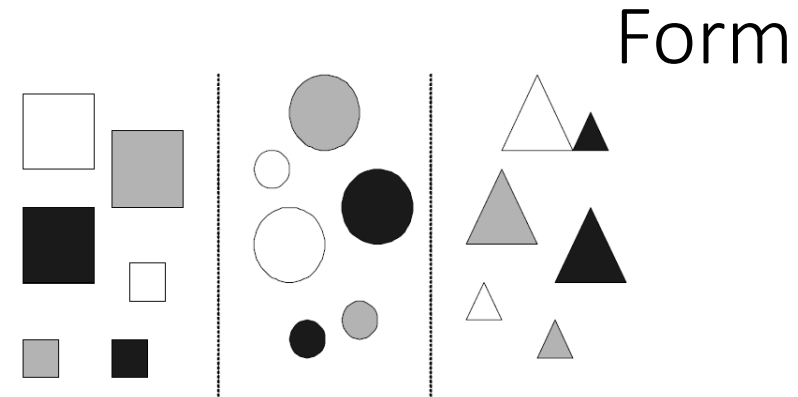
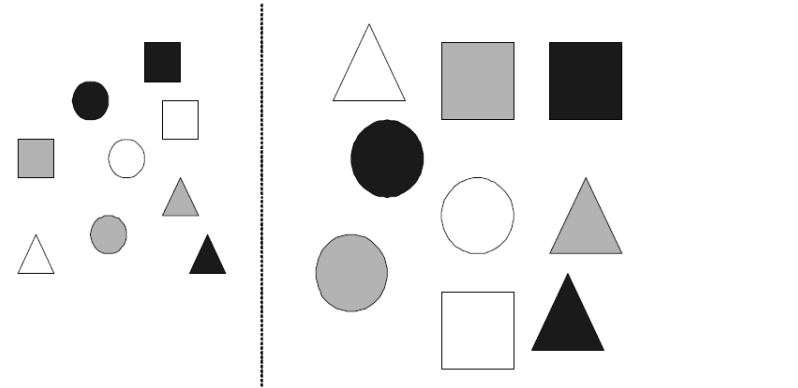
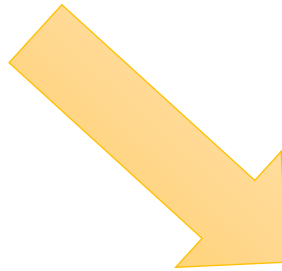
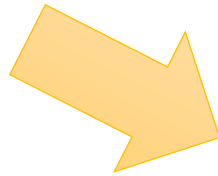
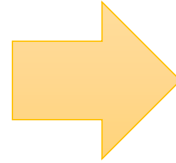
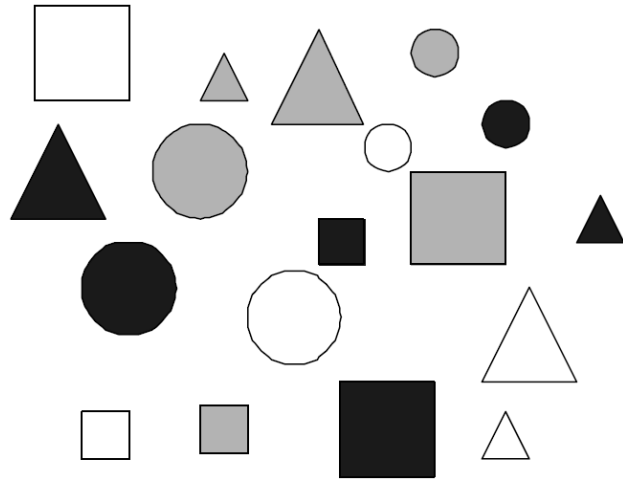
```
// Print current Web page to FILENAME.
void print_to_file(string filename, Presentation *p)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed = p->confirm_loss(filename);
        if (!confirmed)
            return;
    }

    // Proceed printing to FILENAME
    ...
}
```

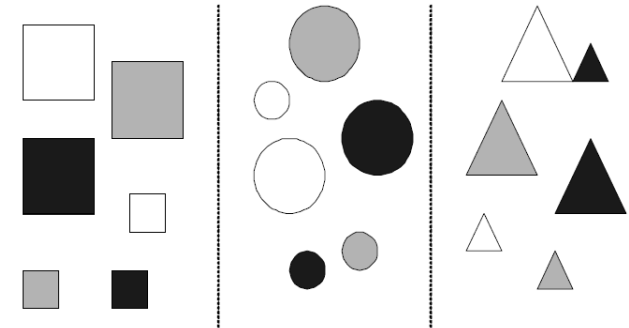
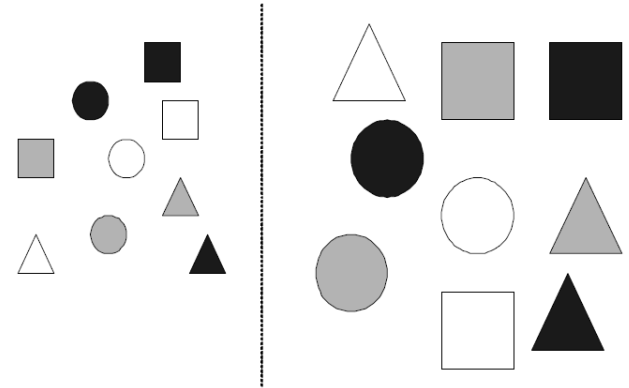
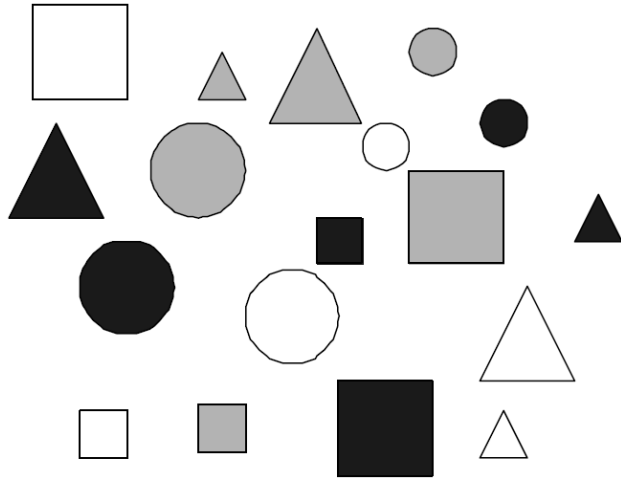
# Abhängigkeit von der Abstraktion



# Wahl der Abstraktion

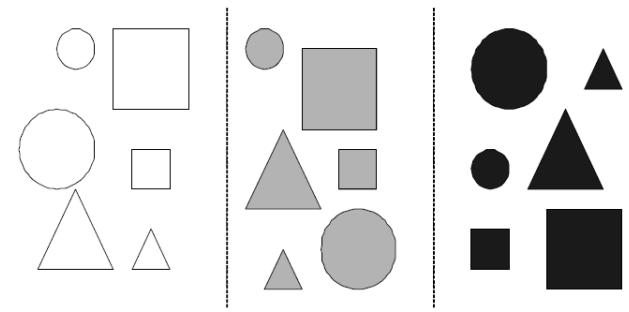


# Wahl der Abstraktion



Welche ist die “*dominante*” Abstraktion?

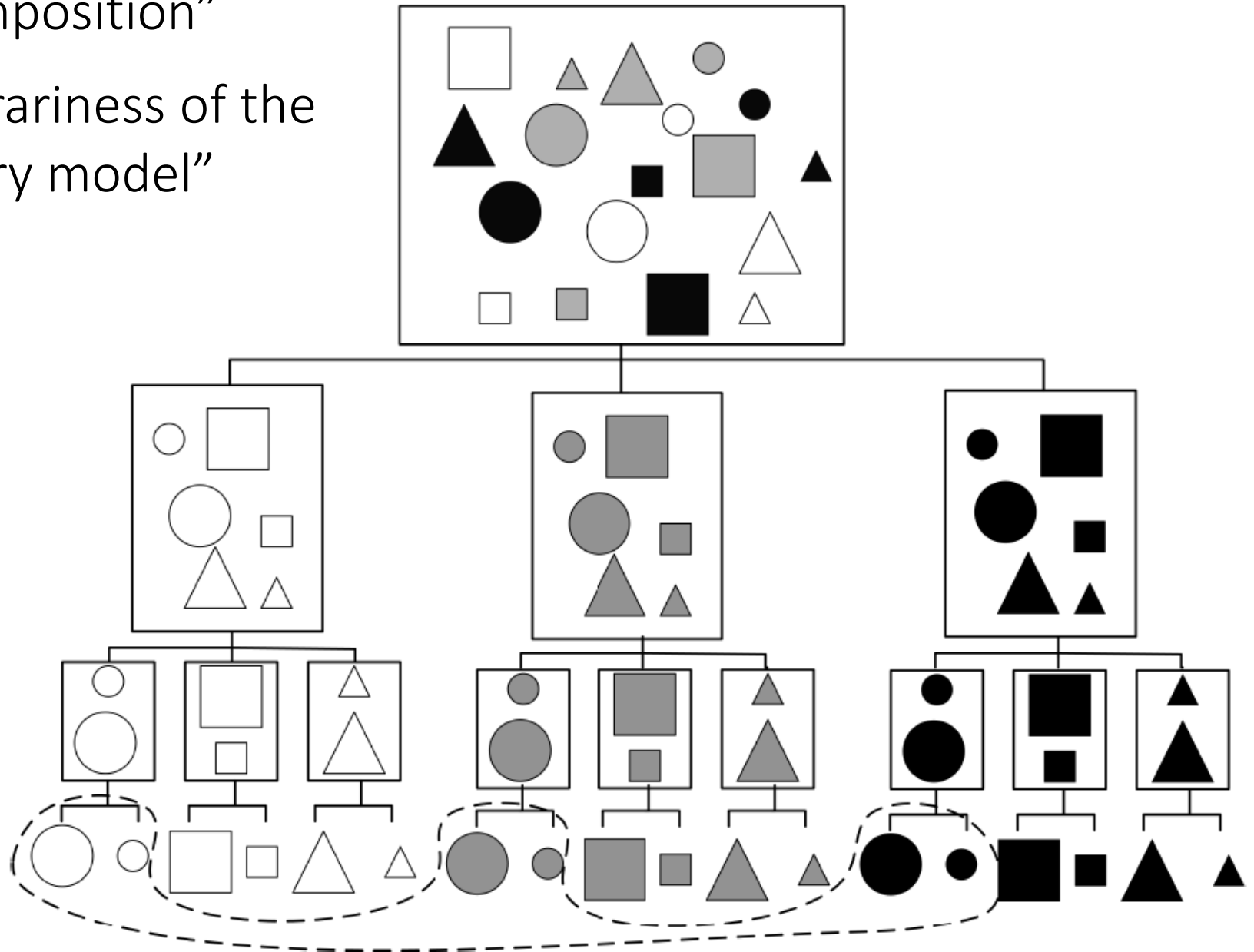
Wie wirkt sich die Wahl auf den Rest des Systems aus?





“Tyranny of the dominant  
decomposition”

“Arbitrariness of the  
primary model”



# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung – Änderungen lokal halten

Modularität – Informationsfluss kontrollieren

Hohe Kohäsion • Schwache Kopplung • nur mit Freunden reden

Hierarchie – Abstraktionen ordnen

Klassen offen für Erweiterungen, geschlossen für Änderungen • Unterklassen, die nicht mehr verlangen und nicht weniger liefern • Abhängigkeiten nur zu Abstraktionen

# Grundprinzipien

des objektorientierten Modells

Abstraktion – Irrelevante Details ausblenden

Einkapselung – Änderungen lokal halten

Modularität – Informationsfluss kontrollieren

Hohe Kohäsion • Schwache Kopplung • nur mit Freunden reden

Hierarchie – Abstraktionen ordnen

Klassen offen für Erweiterungen, geschlossen für Änderungen • Unterklassen, die nicht mehr verlangen und nicht weniger liefern • Abhängigkeiten nur zu Abstraktionen

Ziel: *Änderbarkeit* + *Wiederverwendbarkeit*