Prinzipien des Software-Entwurfs

Software-Praktikum, Sommer 2016

Andreas Zeller • Universität des Saarlandes

Problem

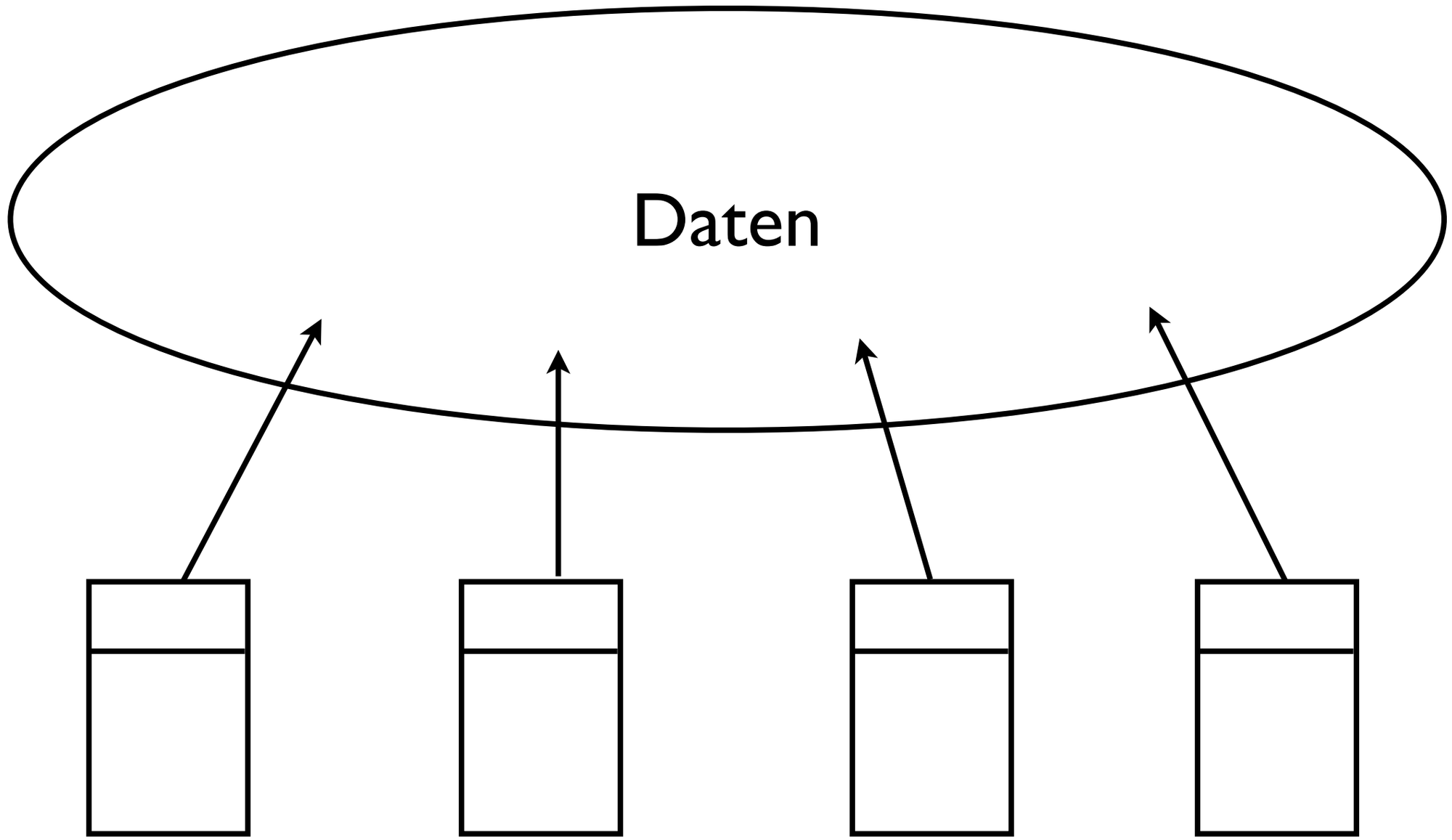
- Software lebt oft sehr viel länger als ursprünglich geplant
- Software muss beständig angepasst werden
- Anteil Wartungskosten 50–80%
- Hauptziel: Software *änderbar* und *wiederverwendbar* machen – bei geringem Risiko und niedrigen Kosten.

Programmierstile

- Chaotisch
- Prozedural
- Modular
- Objektorientiert

Chaos

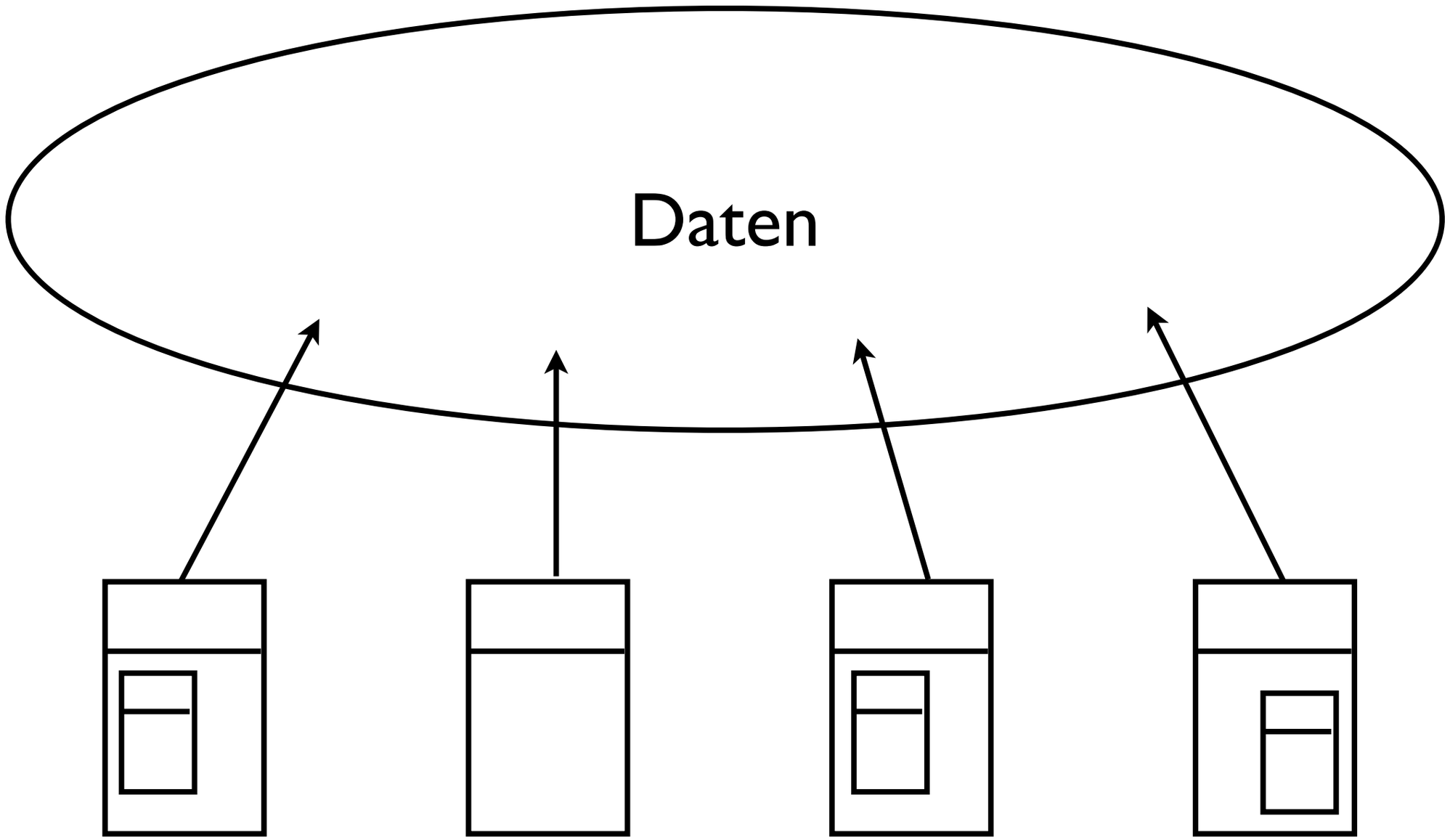
Fortran • Algol (1954–1958)



Programme teilen sich Daten

Prozeduren

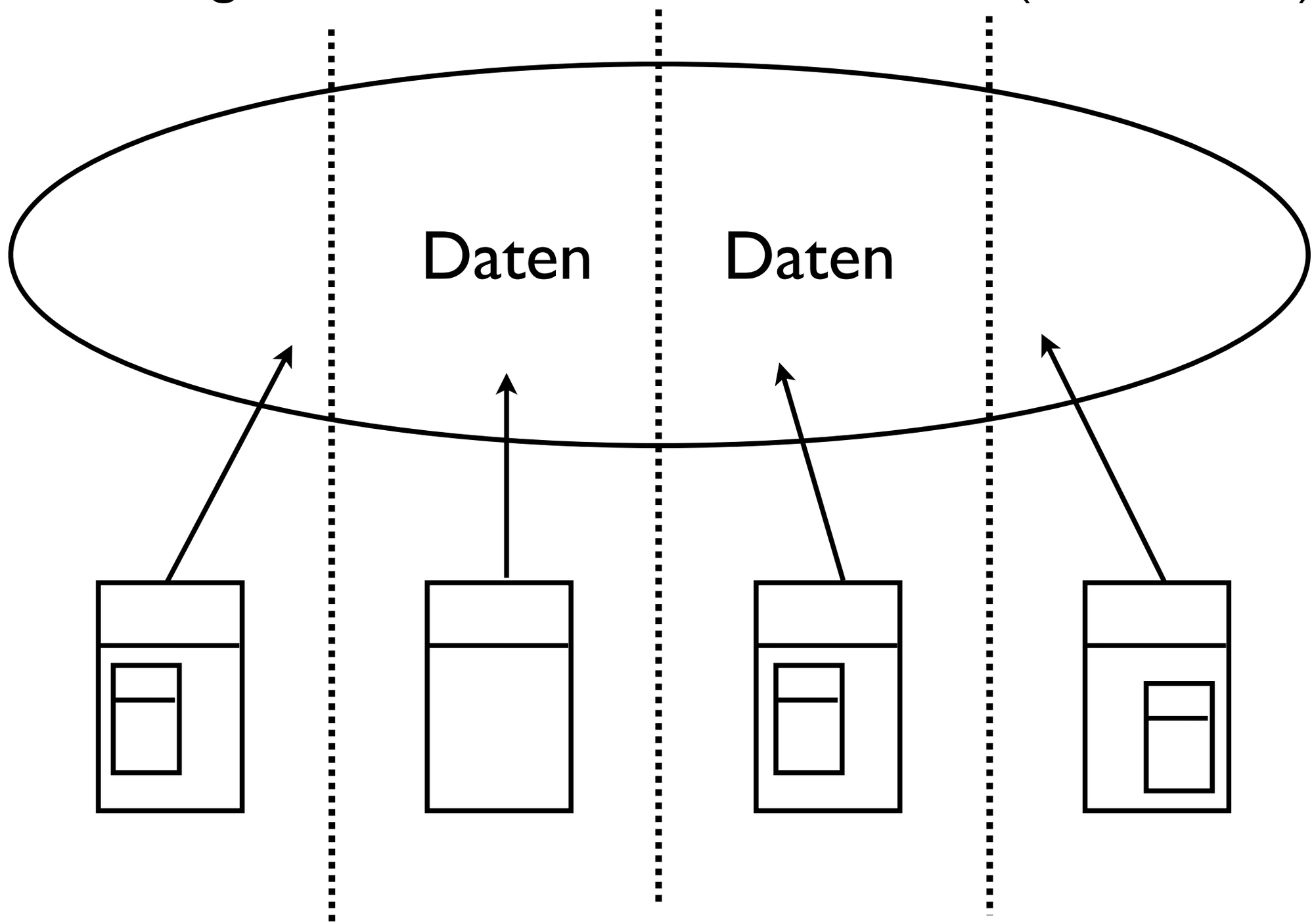
Fortran • Algol • Cobol • Lisp (1959–1961)



Unterprogramme mit Parametern

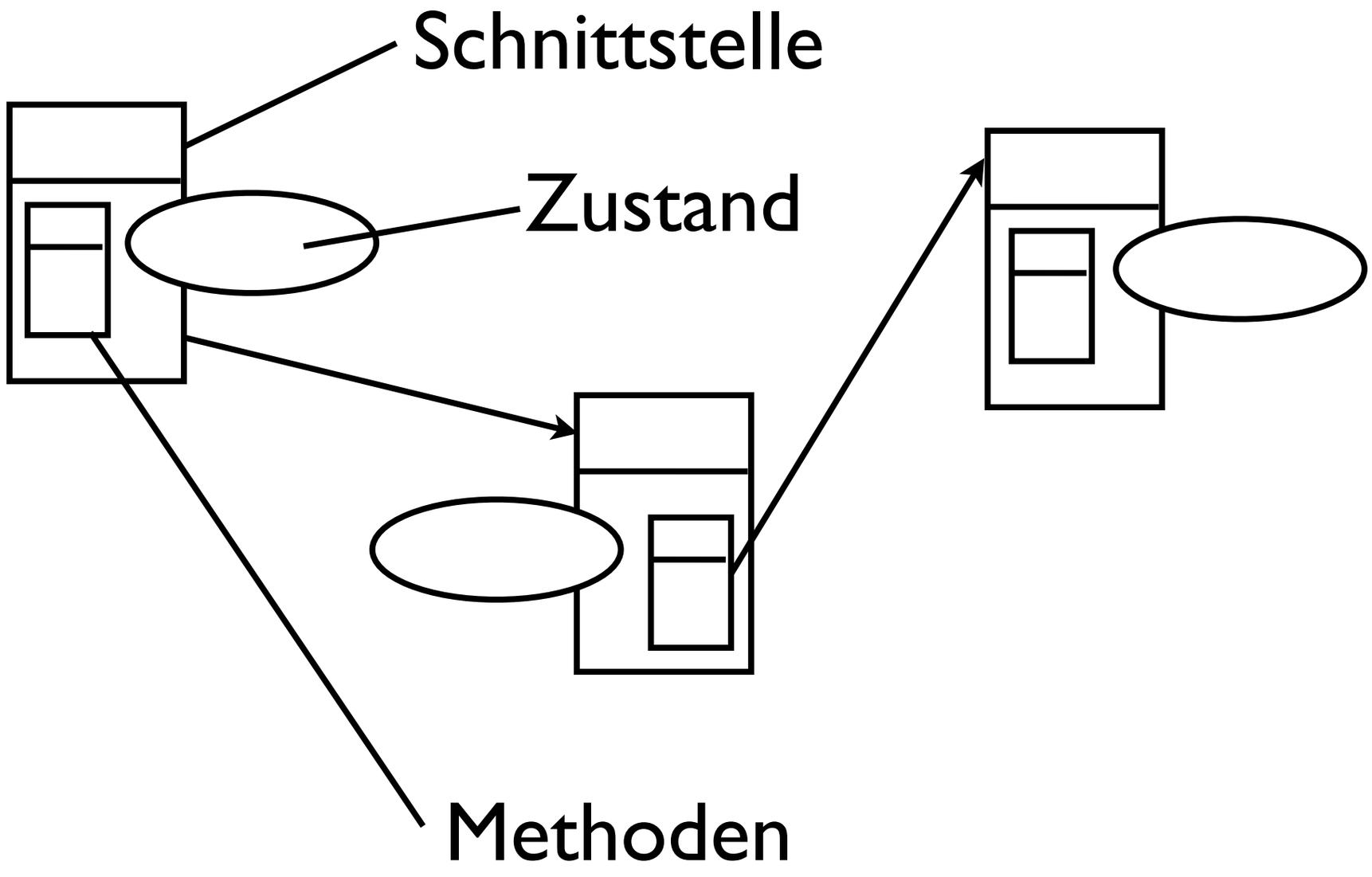
Module

PL/1 • Algol 68 • Pascal • Modula • Simula (1962–1970)



Objekte

Smalltalk • C++ • Ada • Eiffel • Java (1980–)



Übersicht

Generation	Kontrolle	Daten
chaotisch	beliebig	beliebig
prozedural	Prozedur	beliebig
modular	Prozedur	Modul
objektorientiert	Methode	Objekt

außerdem: logikbasiert, regelbasiert, constraintbasiert, funktional…

Grundprinzipien

des objektorientierten Modells

- Abstraktion
- Einkapselung
- Modularität
- Hierarchie

Ziel: Änderbarkeit + Wiederverwendbarkeit

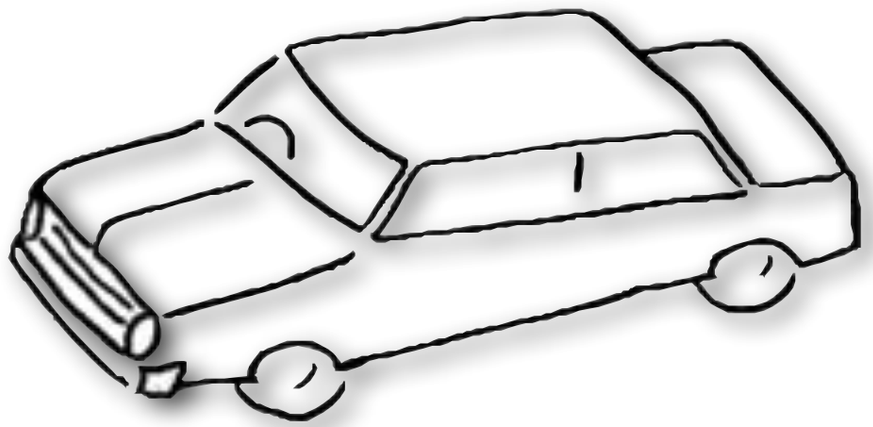
Grundprinzipien

des objektorientierten Modells

- Abstraktion
- Einkapselung
- Modularität
- Hierarchie

Abstraktion

Konkretes Objekt



Allgemeines Prinzip

Abstraktion…

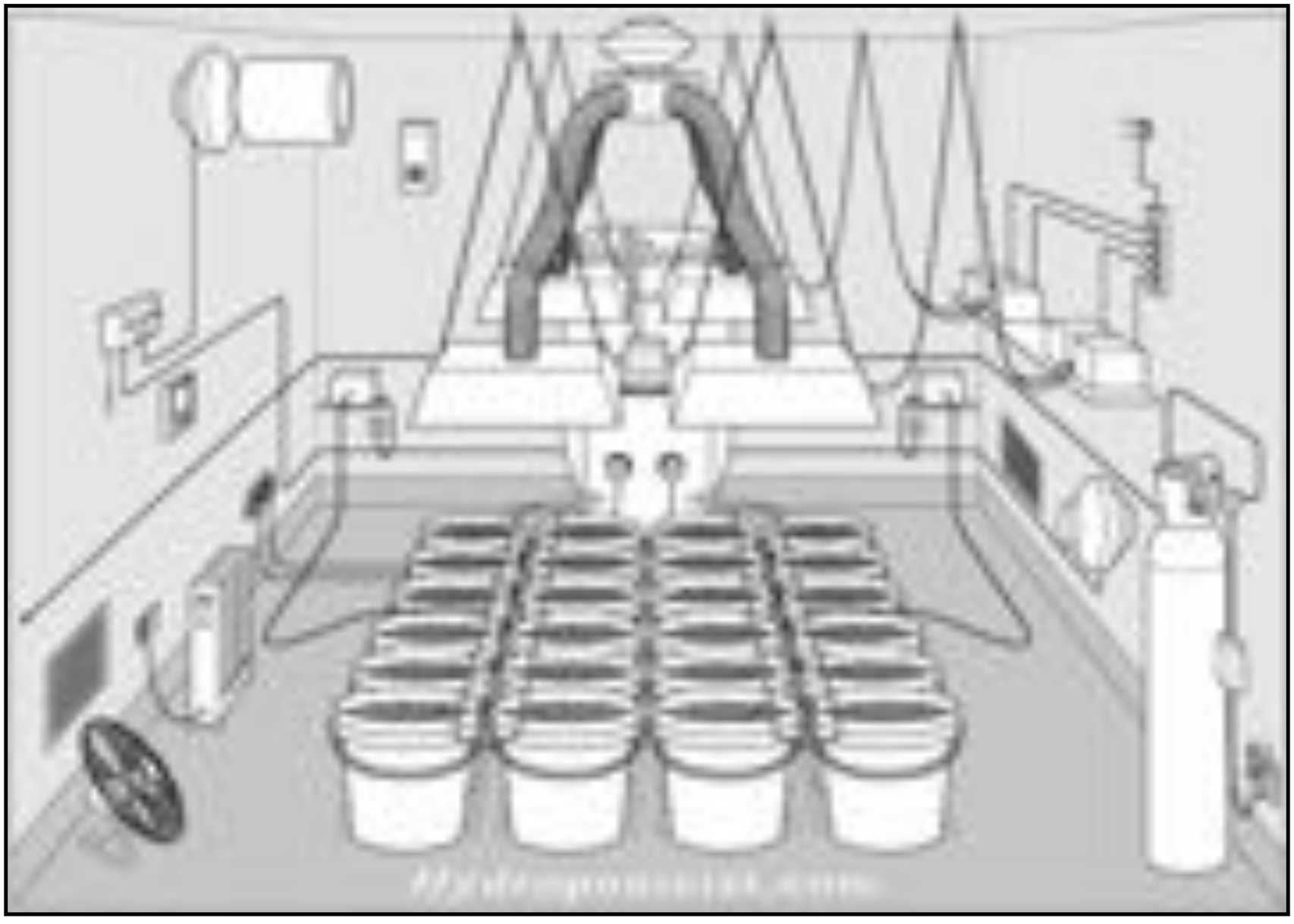
- Hebt gemeinsame Eigenschaften von Objekten hervor
- Unterscheidet zwischen wichtigen und unwichtigen Eigenschaften
- Muss unabhängig von Objekten verstanden werden können

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



Ingenieurs-Lösung

```
void check_temperature() {
    // vgl. Dokumentation AEG-Sensor Typ 700, S. 53
    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

```
typedef float Temperature;
typedef int Location;

class TemperatureSensor {
public:
    TemperatureSensor(Location);
    ~TemperatureSensor();

    void calibrate(Temperature actual);
    Temperature currentTemperature() const;
    Location location() const;

private: …
}
```

Alle Details der Implementierung bleiben versteckt

Abstraktion anders

Grundprinzipien

des objektorientierten Modells

- Abstraktion – Details verstecken
- Einkapselung
- Modularität
- Hierarchie

Grundprinzipien

des objektorientierten Modells

- Abstraktion – Details verstecken
- Einkapselung
- Modularität
- Hierarchie

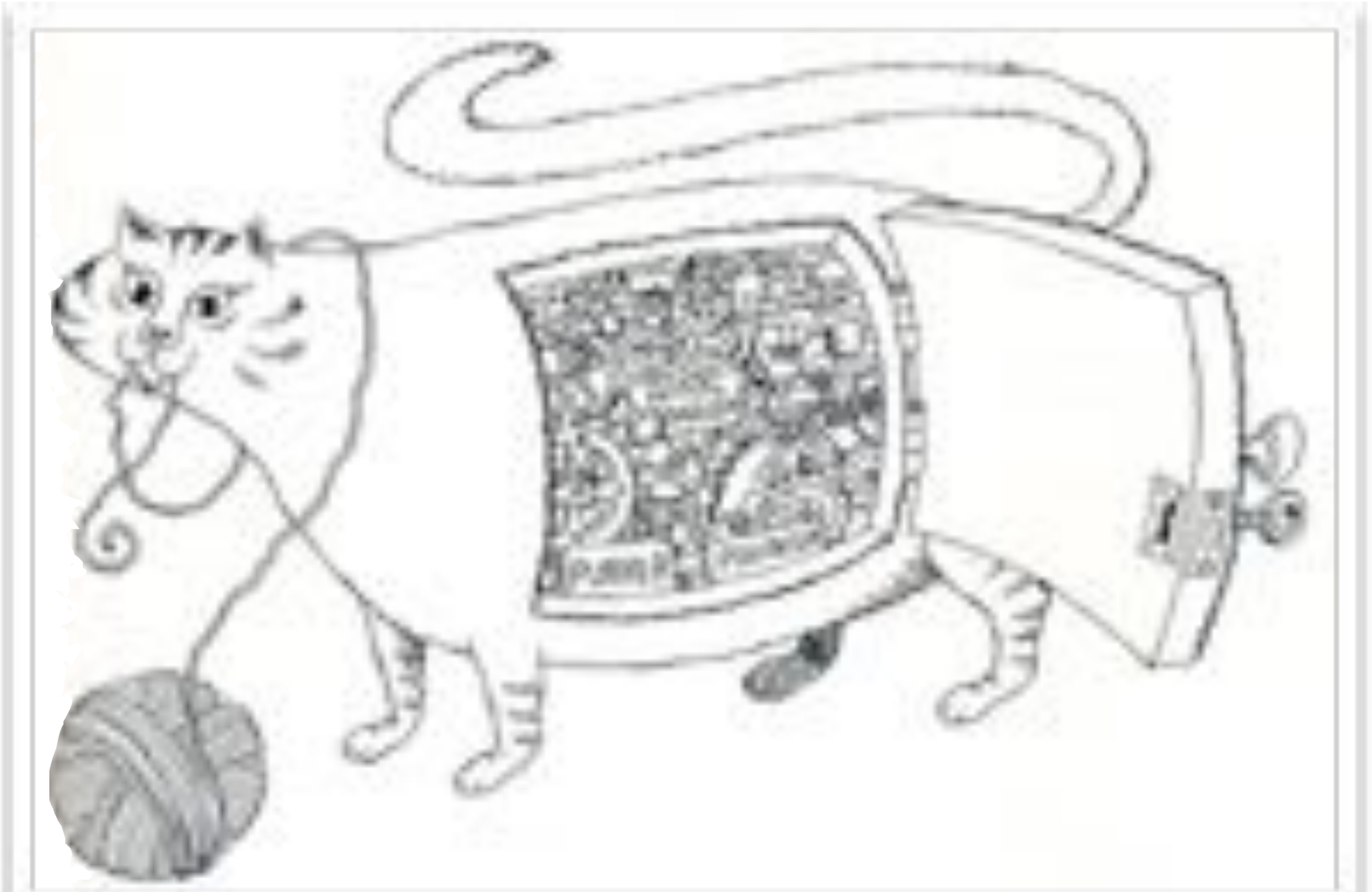
Einkapselung

- 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

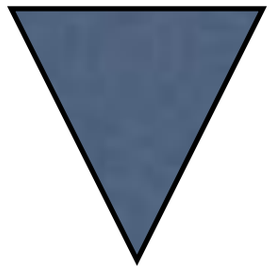
Antizipation des Wandels

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

- Zahlen-Literale
- String-Literale
- Präsentation und Interaktion

Zahlen-Literale

```
int a[100]; for (int i = 0; i <= 99; i++) a[i] = 0;
```

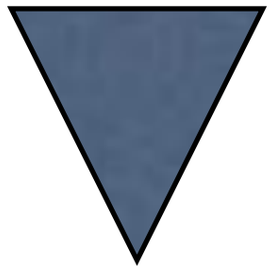


```
const int SIZE = 100;
int a[SIZE]; for (int i = 0; i < SIZE; i++) a[i] = 0;
```

~~const int ONE_HUNDRED = 100;~~
~~int a[ONE_HUNDRED];~~

Zahlen-Literale

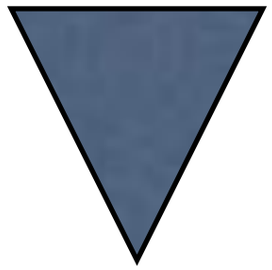
```
double brutto = netto * 1.19;
```



```
final double MWST = 1.19;
double brutto = netto * MWST;
```

String-Literale

```
if (sensor.temperature() > 100)
    printf(“Wasser kocht!”);
```



```
if (sensor.temperature() > BOILING_POINT)
    printf(message(BOILING_WARNING, “Wasser kocht!”);

if (sensor.temperature() > BOILING_POINT)
    alarm.handle_boiling();
```

Grundprinzipien

des objektorientierten Modells

- Abstraktion – Details verstecken
- Einkapselung – Änderungen lokal halten
- Modularität
- Hierarchie

Grundprinzipien

des objektorientierten Modells

- Abstraktion – Details verstecken
- 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 – Module sollen logisch zusammengehörige Funktionen enthalten
- Schwache Kopplung – Änderungen in Modulen sollen sich nicht auf andere Module auswirken
- Demeter-Prinzip – nur mit Freunden reden

Hohe Kohäsion

- Module sollen logisch zusammengehörige Funktionen enthalten
- Erreicht man durch Zusammenfassen der Funktionen, die auf denselben *Daten* arbeiten
- Bei objektorientierter Modellierung “natürliche” Zusammenfassung

Schwache Kopplung

- Änderungen in Modulen sollen sich nicht auf andere Module auswirken
- Erreicht man durch
  - Geheimnisprinzip
  - Abhängigkeit zu möglichst wenig Modulen

Demeter-Prinzip

- Grundidee: So wenig wie möglich über Objekte und ihre Struktur annehmen
- Verfahren: Methodenaufrufe auf *Freunde* beschränken

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: Beispiel

```
class Uni {
    Prof boring = new Prof();
    public Prof getProf() { return boring; }
    public Prof getNewProf() { return new Prof(); }
}

class Test {
    Uni uds = new Uni();
    public void one() { uds.getProf().fired(); }
    public void two() { uds.getNewProf().hired(); }
}
```

Demeter: Beispiel

```
class Uni {
    Prof boring = new Prof();
    public Prof getProf() { return boring; }
    public Prof getNewProf() { return new Prof(); }
    public void fireProf(...) { ... }
}

class BetterTest {
    Uni uds = new Uni();
    public void betterOne() { uds.fireProf(...); }

}
```

Pipelines

Folgen von Funktionen als *Verarbeitungsschritte* können als *Ketten* genutzt werden

```
long countLongStrings =
   myList.stream()
      .filter(element -> element.length() > 4)
      .count();
```

Da alle Funktionen auf demselben Objekt arbeiten, verletzt dies nicht die Demeter-Regel

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 – Details verstecken
- Einkapselung – Änderungen lokal halten
- **Modularität – Informationsfluss kontrollieren**
  Hohe Kohäsion • Schwache Kopplung • nur mit Freunden reden
- Hierarchie

Grundprinzipien

des objektorientierten Modells

- Abstraktion – Details verstecken
- 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/Close-Prinzip – Klassen sollten offen für Erweiterungen sein
- Liskov-Prinzip – Unterklassen sollten nicht mehr verlangen, und nicht weniger liefern
- Abhängigkeits-Prinzip – Abhängigkeiten sollten nur zu Abstraktionen bestehen

Hierarchie-Prinzipien

- Open/Close-Prinzip – Klassen sollten offen für Erweiterungen sein
- Liskov-Prinzip – Unterklassen sollten nicht mehr verlangen, und nicht weniger liefern
- Abhängigkeits-Prinzip – Abhängigkeiten sollten nur zu Abstraktionen bestehen

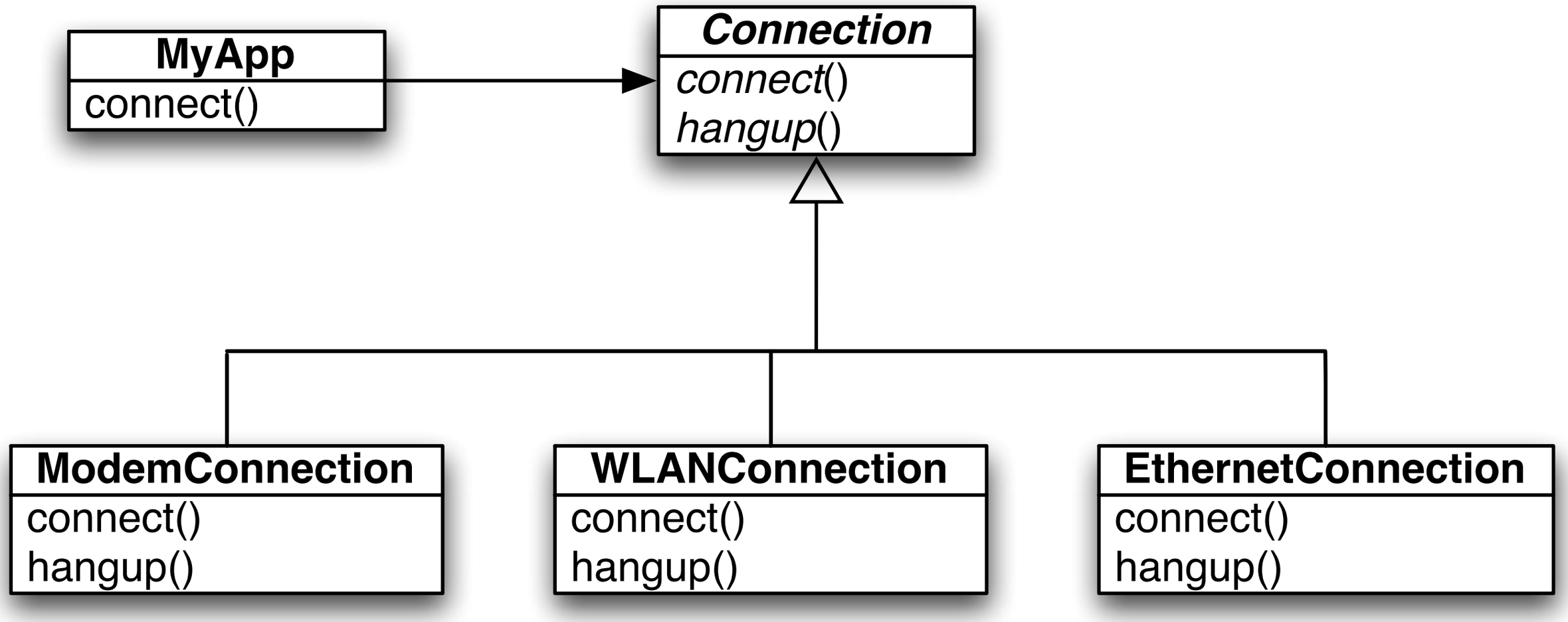
Open/Close-Prinzip

- Eine Klasse sollte *offen* für Erweiterungen, aber *geschlossen* für Änderungen sein
- Wird durch *Vererbung* und *dynamische Bindung* erzielt

Ein Internet-Anschluss

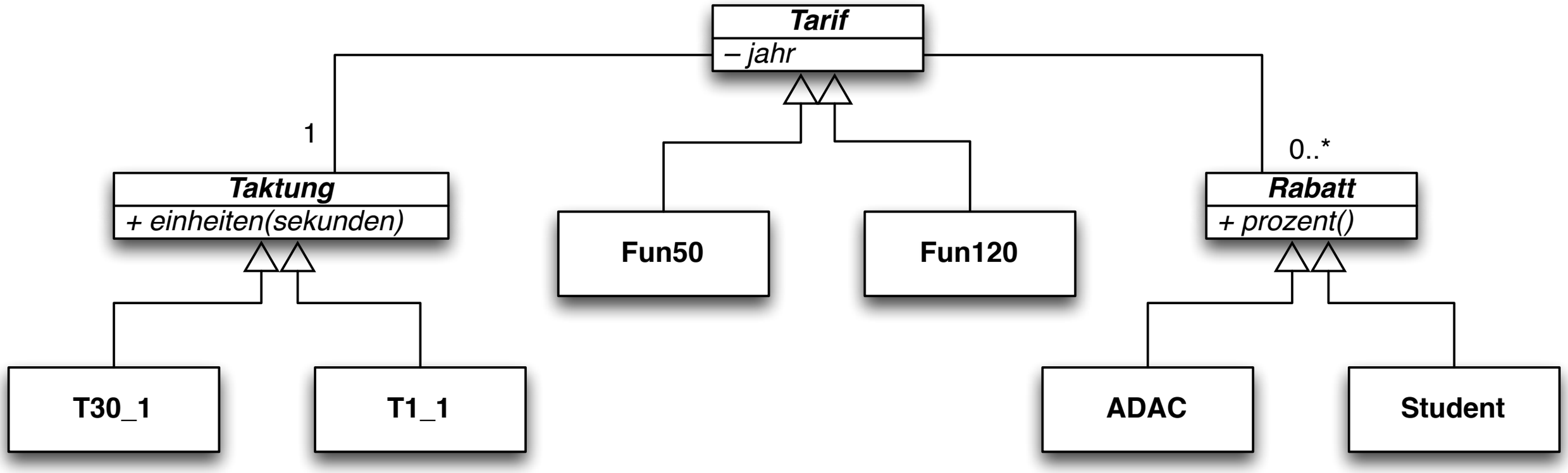
```
void connect() {
     if (connection_type == MODEM_56K)
     {
          Modem modem = new Modem();
          modem.connect();
     }
     else if (connection_type == ETHERNET) …
     else if (connection_type == WLAN) …
     else if (connection_type == UMTS) …
}
```

Lösung mit Hierarchien



```
enum { FUN50, FUN120, FUN240, ... } tarif;
enum { STUDENT, ADAC, ADAC_UND_STUDENT ... } spezial;
enum { PRIVAT, BUSINESS, ... } kundentyp;
enum { T60_1, T60_60, T30_1, ... } taktung;

int rechnungsbetrag(int sekunden)
{
    if (kundentyp == BUSINESS)
        taktung = T1_1;
    else if (tarif == FUN50 || tarif == FUN120)
        taktung = T60_1;
    else if (tarif == FUN240 && vertragsjahr < 2006)
        taktung = T30_1;
    else
        taktung = T60_60;

    if (vertragsjahr >= 2006 && spezial != ADAC)
        taktung = T60_60;
    // usw. usf.
```

Lösung mit Hierarchien



- Neue Rabattstufen, Tarife, etc. können jederzeit hinzugefügt werden!

Hierarchie-Prinzipien

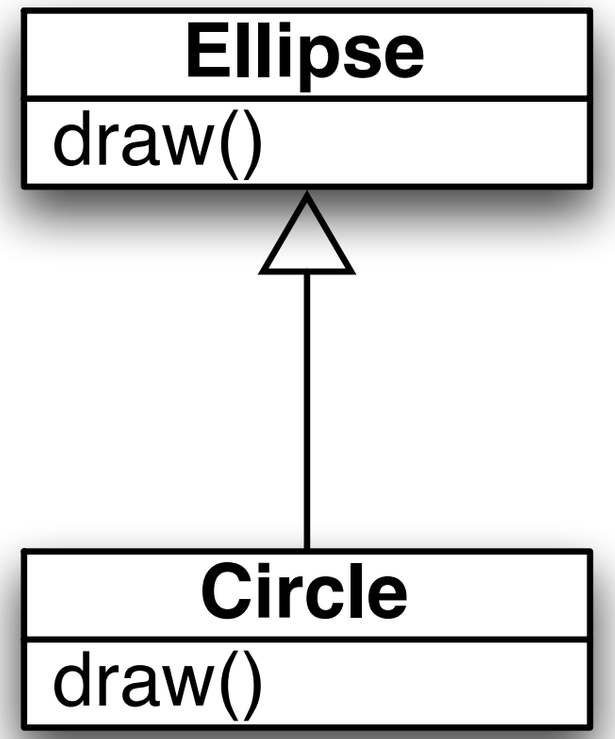
- Open/Close-Prinzip – Klassen sollten offen für Erweiterungen sein
- Liskov-Prinzip – Unterklassen sollten nicht mehr verlangen, und nicht weniger liefern
- Abhängigkeits-Prinzip – Abhängigkeiten sollten nur zu Abstraktionen bestehen

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*



Hierarchie-Prinzipien

- Open/Close-Prinzip – Klassen sollten offen für Erweiterungen sein
- Liskov-Prinzip – Unterklassen sollten nicht mehr verlangen, und nicht weniger liefern
- Abhängigkeits-Prinzip – Abhängigkeiten sollten nur zu Abstraktionen bestehen

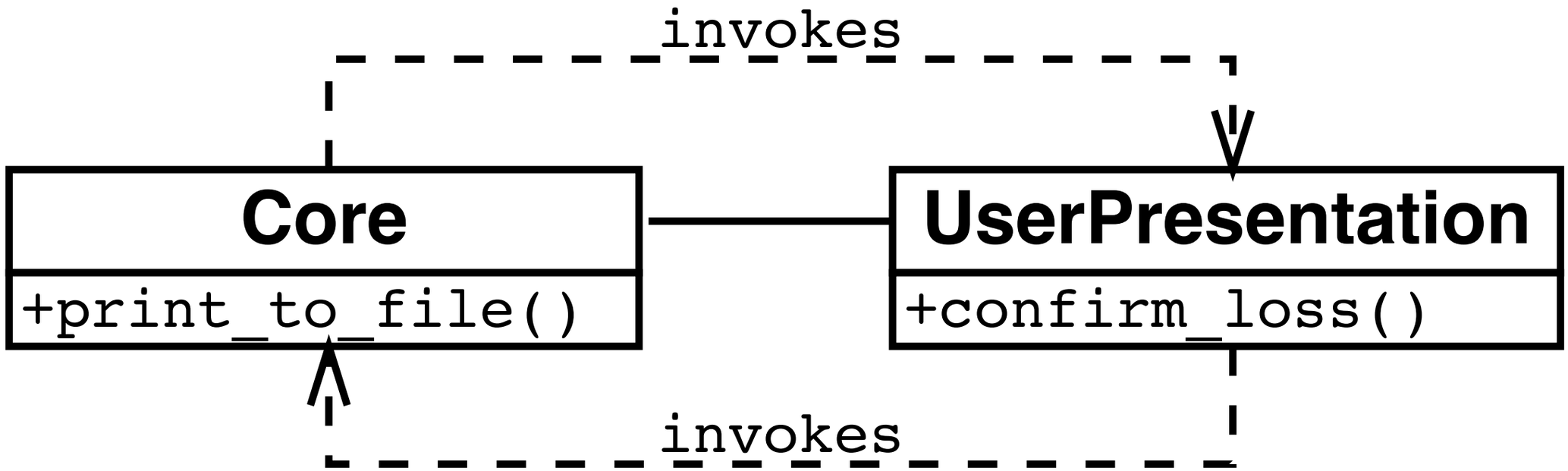
Abhängigkeits-Prinzip

- 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

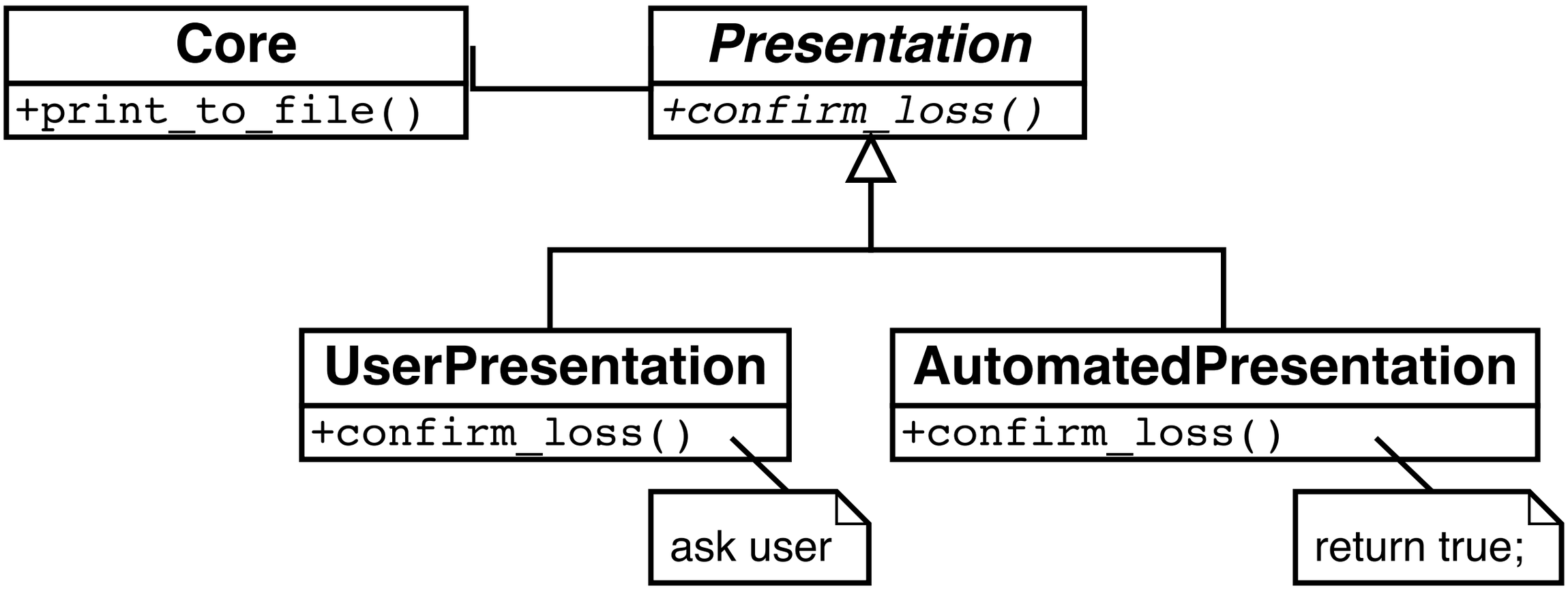


Konstruktion, Test, Wiederverwendung einzelner Module werden unmöglich!

Abhängigkeit

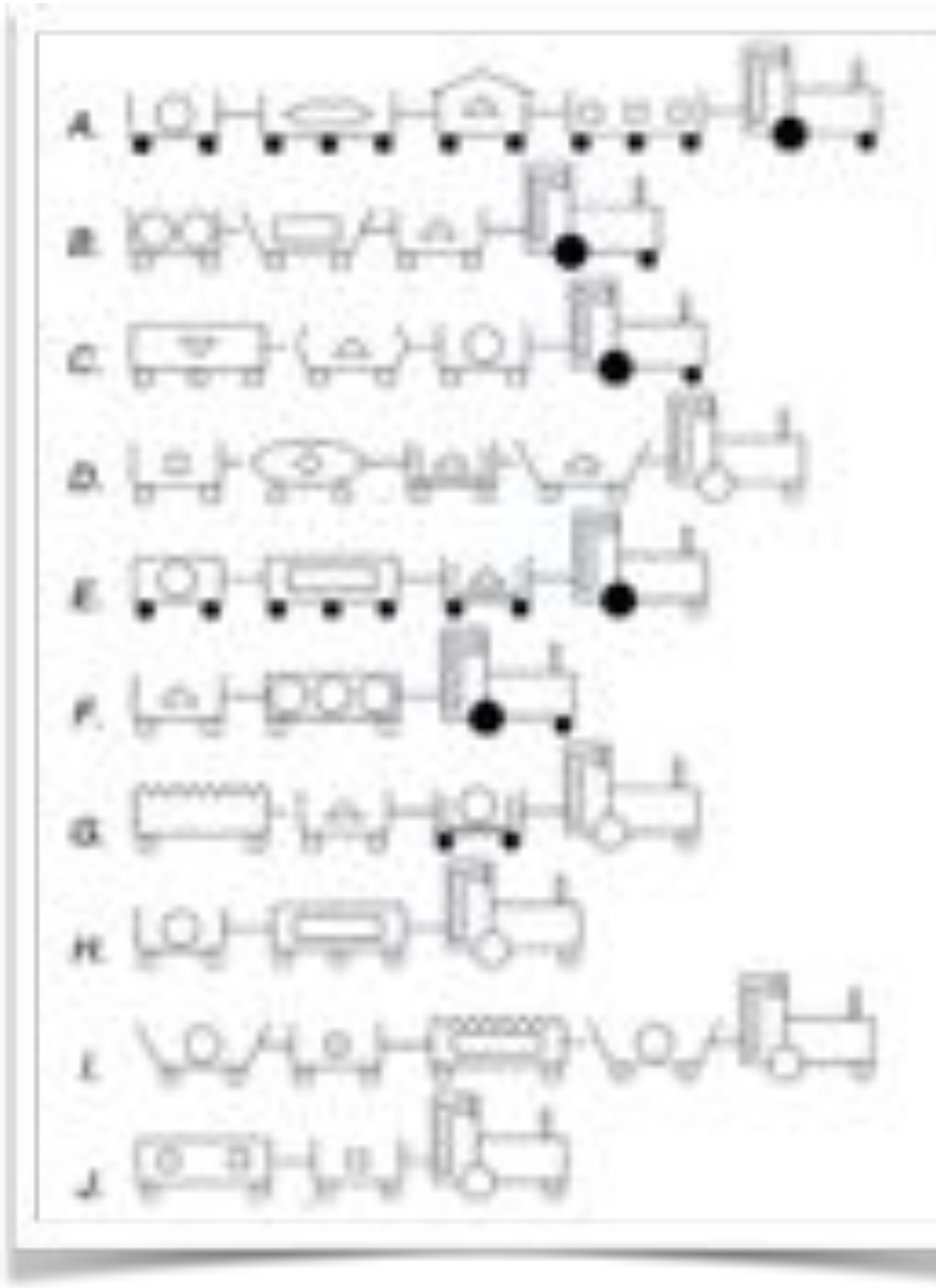
```
// Print current Web page to FILENAME.
void print_to_file(string filename, Presentation *p)
{
    if (path_exists(filename))
    {
        // FILENAME exists;
        // ask user to confirm overwrite
        bool confirmed = p->confirm_loss(filename);
        if (!confirmed)
            return;
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

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

Abhängigkeit von der Abstraktion



Wahl der Abstraktion

- Welche ist die “dominante” Abstraktion?
- Wie wirkt sich die Wahl auf den Rest des Systems aus?



Hierarchie-Prinzipien

- Open/Close-Prinzip – Klassen sollten offen für Erweiterungen sein
- Liskov-Prinzip – Unterklassen sollten nicht mehr verlangen, und nicht weniger liefern
- Abhängigkeits-Prinzip – Abhängigkeiten sollten nur zu Abstraktionen bestehen

Grundprinzipien

des objektorientierten Modells

- Abstraktion – Details verstecken
- 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 – Details verstecken
- 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