Objektorientierte Softwareentwicklung

Vorlesungsreihe

BrücknerJarosch
Automation + Informatik

Objektorientierte Softwareentwicklung

Vorlesung 1

Motivation und Grundlagen

Vorstellung

- Peter Brückner
- Mail: peter@bj-ig.de
- Bei Mail: [OOSE Aufgabe Matrikelnummern] Anliegen
- URL http://www.bj-ig.de
- Automation und Compilerbau

Organisation

- Zeit: Samstag, 9:00-12:30 Uhr
- Ort: HU 211/212
- Modus: 14 Vorlesungen und eine Klausur
- Klausur: 60 Minuten / 40 Punkte

Voraussetzungen

- Algorithmen und Datenstrukturen
- Datenbanken
- Logisches und analytisches Denken
- Nützlich: C, Java oder C++ Kenntnisse

Inhalte

- Motivation und Historie
- Kurze Einführung (Wiederholung) in UML
- Objektorientierter Entwurf (Spezifikation, Analyse und Design)
- Objektorientierte Programmierung
- Design Patterns (OO)
- C++-Programmierung

Bücher

- **UML Distilled – Third Edition**.
Martin Fowler, Kendall Scott; Addison-Wesley 2003
ISBN: 0-321-19368-7
- **The C++ Programming Language**.
Bjarne Stroustrup; Addison-Wesley Professional 2000
ISBN: 0201700735
- **Unified Modelling Language** User Guide.
Grady Booch, James Rumbaugh, Ivar Jacobson
Addison Wesley Longman 1999
ISBN: 0201571684
- **Object-oriented Software Construction**. Bertrand Meyer
Prentice Hall PTR 1997
ISBN: 0136291554
- **Design Patterns. Elements of Reusable Object- Oriented Software**. Erich Gamma, Richard Helm, Ralph Johnson; Addison-Wesley Professional 1997 ISBN: 0201633612

URLS

- Object Managment Group: (OMG)
  http://www.omg.org
- Rational (bei IBM)
  http://www.rational.com
- Bjarne Stoustrup
  http://www.research.att.com/~bs/
- Bertrand Meyer
  http://www.inf.ethz.ch/personal/meyer/

Software Qualitätsmerkmale

- Funktionalität, Korrektheit,Vollständigkeit (formal)
- Zuverlässigkeit, Robustheit (formal)
- Benutzbarkeit (informal)
- Effizienz/Performance (informal)
- Änderbarkeit/Wartbarkeit (informal)
- Übertragbarkeit (Portierbarkeit)/Skalierbarkeit (informal)
- Zusätzlich:Lesbarbeit, Redundanzfreiheit, Modularität, Nebenwirkungsfreiheit, Wiederverwendbarkeit, Interoperabilität

Motive und Historie (Assembler)

- Am Anfang war die CPU/ Architektur unbestimmt
- Dann Festlegung auf eine Architektur, der im wesentlichen alle Prozessoren folgen.
- Assemblerprogrammierung
- unportabel, nicht interoperabel (Aufrufschema, Datenformate, etc),
- nicht wiederverwendbar, nicht wartbar, unzuverlässig
- Trotzdem wurden Programme geschrieben!
- Insbesondere Programme zum Schreiben von Programmen

Motive und Historie („Hochsprachen/Imperativ“)

- COBOL/FORTRAN/BASIC
- Compiler und Interpreter wurden erfunden; sie Übersetzen Hochsprache nach Assembler-Code
- damit wurde ein Ziel erreicht: Portabilität
- Compiler muss auch Portiert werden
- Damit das funktionieren kann muss ein einheitliches Modell des Rechners aus Sicht des Compilers her code/data/stack/heap Modell, Aufrufkonventionen erleichtern die Interoperabilität
- Kompatibilität

Motive und Historie (Struktur)

- PASCAL,C (Strukturierte Programmierung)
- Unterprogramme (Prozedurale Programmierung)
- Records (Strukturen)
- Gültigkeitsbereiche für Variablen (nicht Typen)
- Assembler -> C (portabler Assembler)
- Modul:Pascal->Modula, Turbo Pascal (Modulare Programmierung)
- Verbesserung:
  - Wartbarkeit
  - Wiederverwendbarkeit
  - Zuverlässigkeit
- Probleme:
  - Code wird oft kopiert.
  - Daten stehen getrennt von Unterprogrammen
  - viele Programmiermodelle verhindern Zusammenspiel von Unterprogrammen aus verschiedenen Quellen

Motive und Historie (Prä-OO)

- Objektorientierte Vorgänger (Programmierung mit Abstrakten Datentypen)
- simple FILE stream io (siehe Beispiel)
- Xt/Motif (horror cast)
- Beispiel: linux kernel

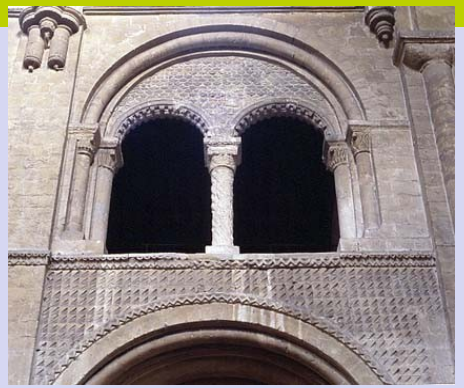
```
C-Interface for File-IO (stdio)
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
int fread(void *ptr, int size, int nmemb, FILE *stream);
int fwrite(void *ptr, int size, int nmemb, FILE *stream);
int ftell(FILE *stream);
int fseek(FILE *stream, int offset, int whence);
int fclose(FILE *stream);

class FILE { // interface
public:
  FILE(string file);
  virtual ~FILE();
  virtual int read(void *ptr,int size);
  virtual int write(void *ptr,int size);
  virtual int tell() const;
  virtual int seek(int pos,int whence);
};
```

```
// Method Table
static struct file_operations i2cdev_fops = {
        .read           = i2cdev_read,
        .write          = i2cdev_write,
        .ioctl          = i2cdev_ioctl,
        .open           = i2cdev_open,
        .release        = i2cdev_release,
};
  register_chrdev(I2C_MAJOR, "i2c", &i2cdev_fops);
// open function
static int i2cdev_open(...,struct file *file) {
  file->private_data=client;
}
// read function
static... i2cdev_read(struct file *file,.....) {
...
  struct i2c_client *client = (struct i2c_client *)file-
>private_data;
}
```

Motive und Historie (OO-Sprachen)

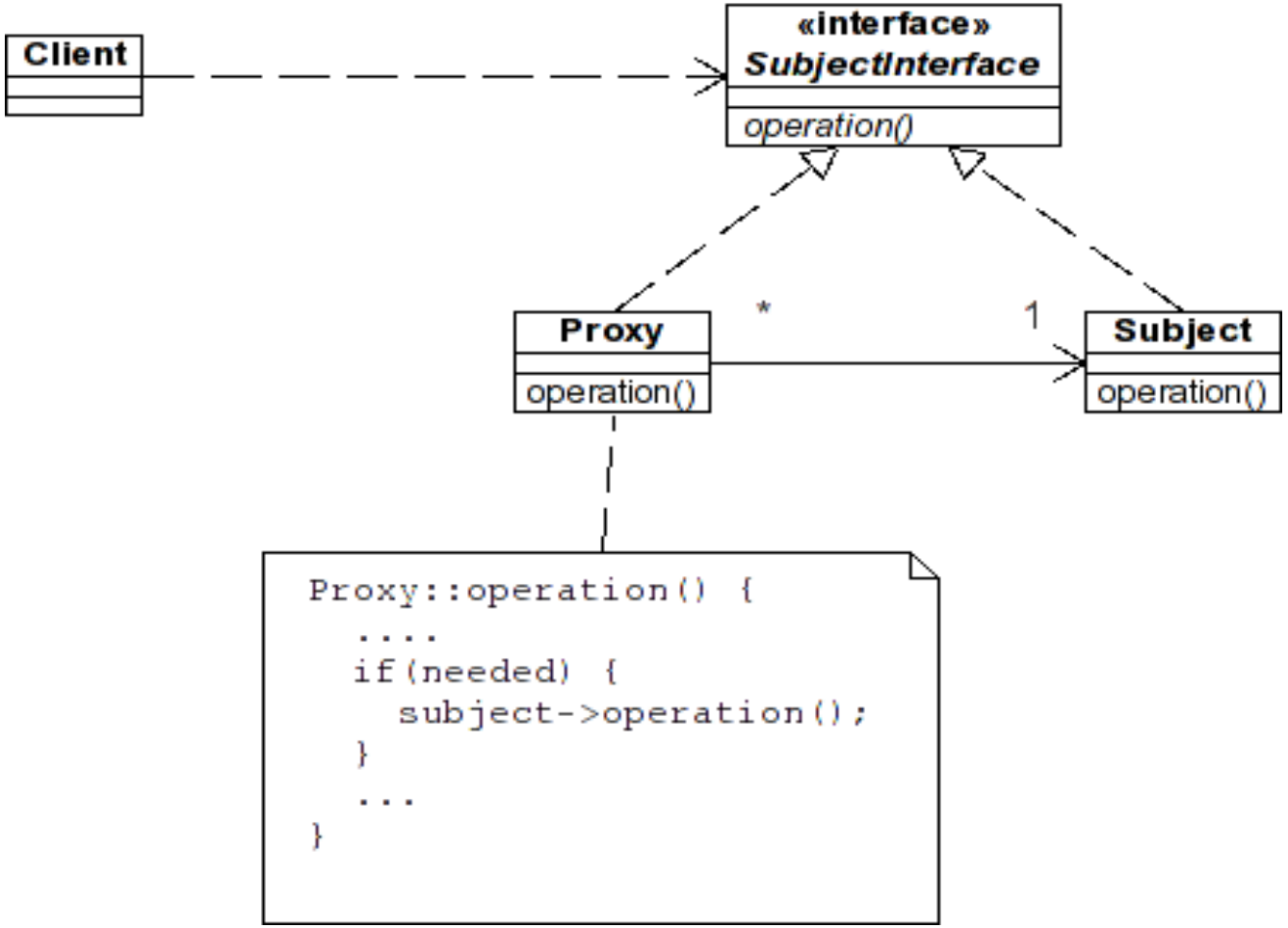
- Simula 67(??)
- Smalltalk (vollständig objektorientiert)
- C++
- Java

Proxy Pattern

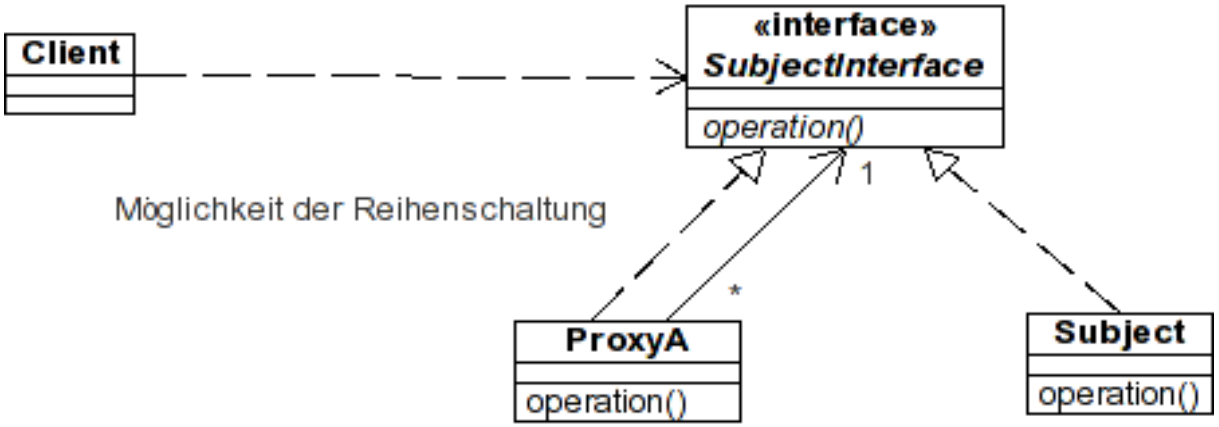


- Problem: Manchmal ist der Zugriff auf bestimmte Objekte besonders kostspielig oder langwierig oder soll überwacht werden.
- Idee: Ein Stellvertreter – ein Objekt mit dem gleichen Interface (damit der Benutzer nichts davon bemerkt)
- Teilnehmer: Client, Service,ServiceInterface, Proxy
- Variationen: Virtual Proxy, Smart Reference, Remote Proxy, Protocoll Proxy, Protection Proxy

Proxy (allgemein)

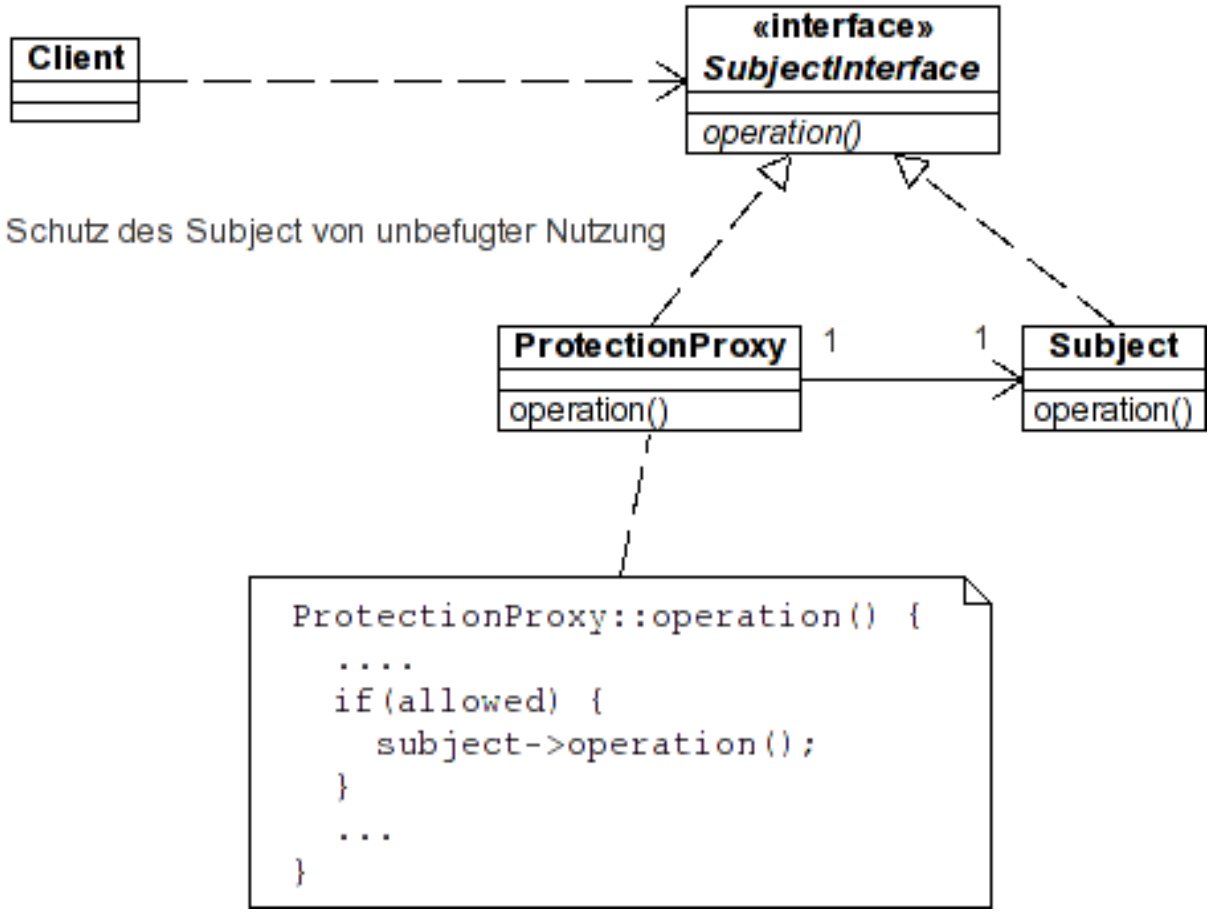


ProxyA (erweitert)

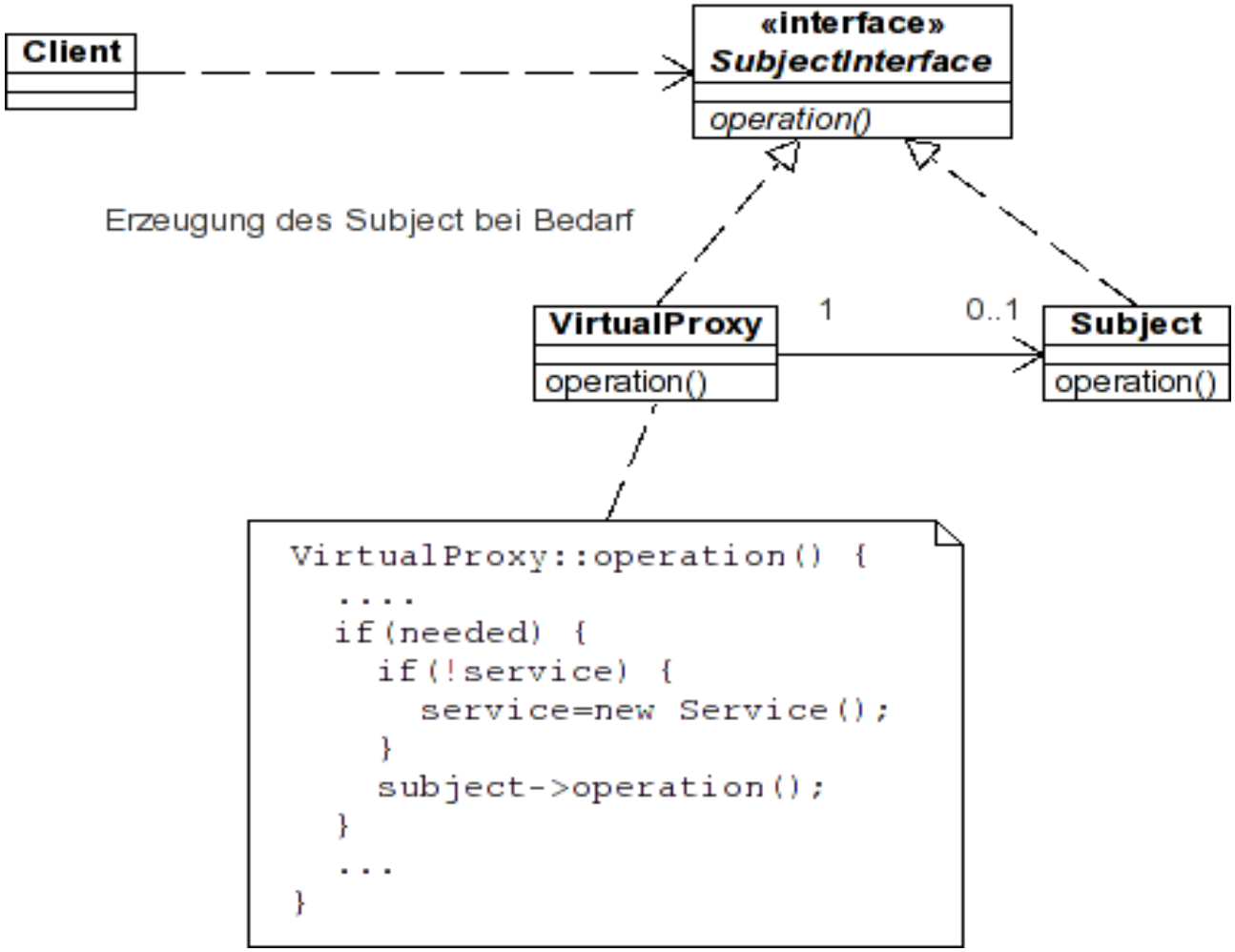



```
ProxyA::operation() {
  ....
  if(needed) {
    subject->operation();
  }
  ...
}
```

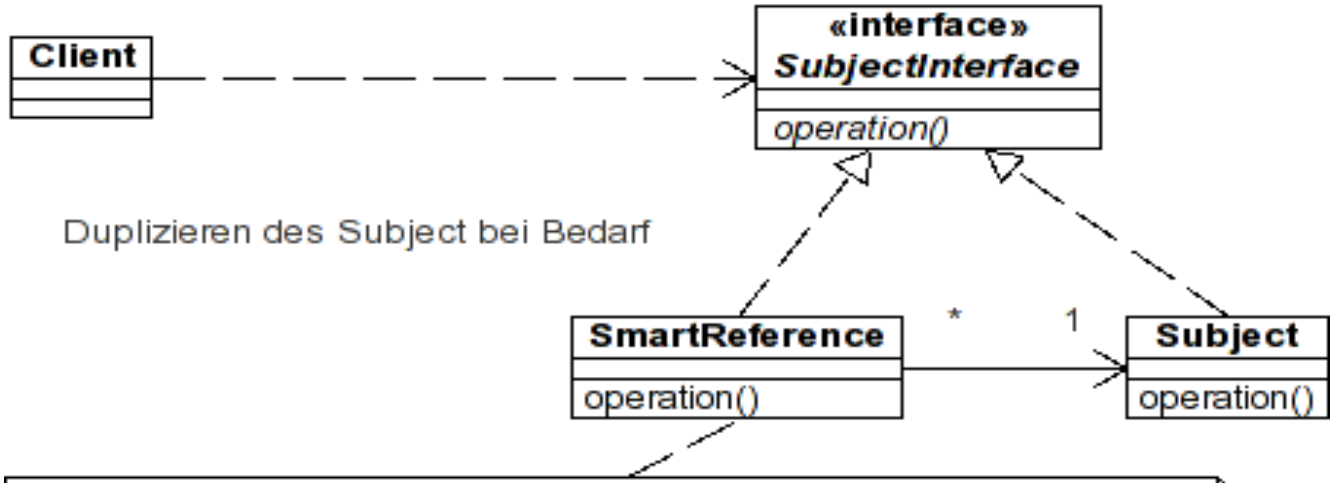
ProtectionProxy



VirtualProxy

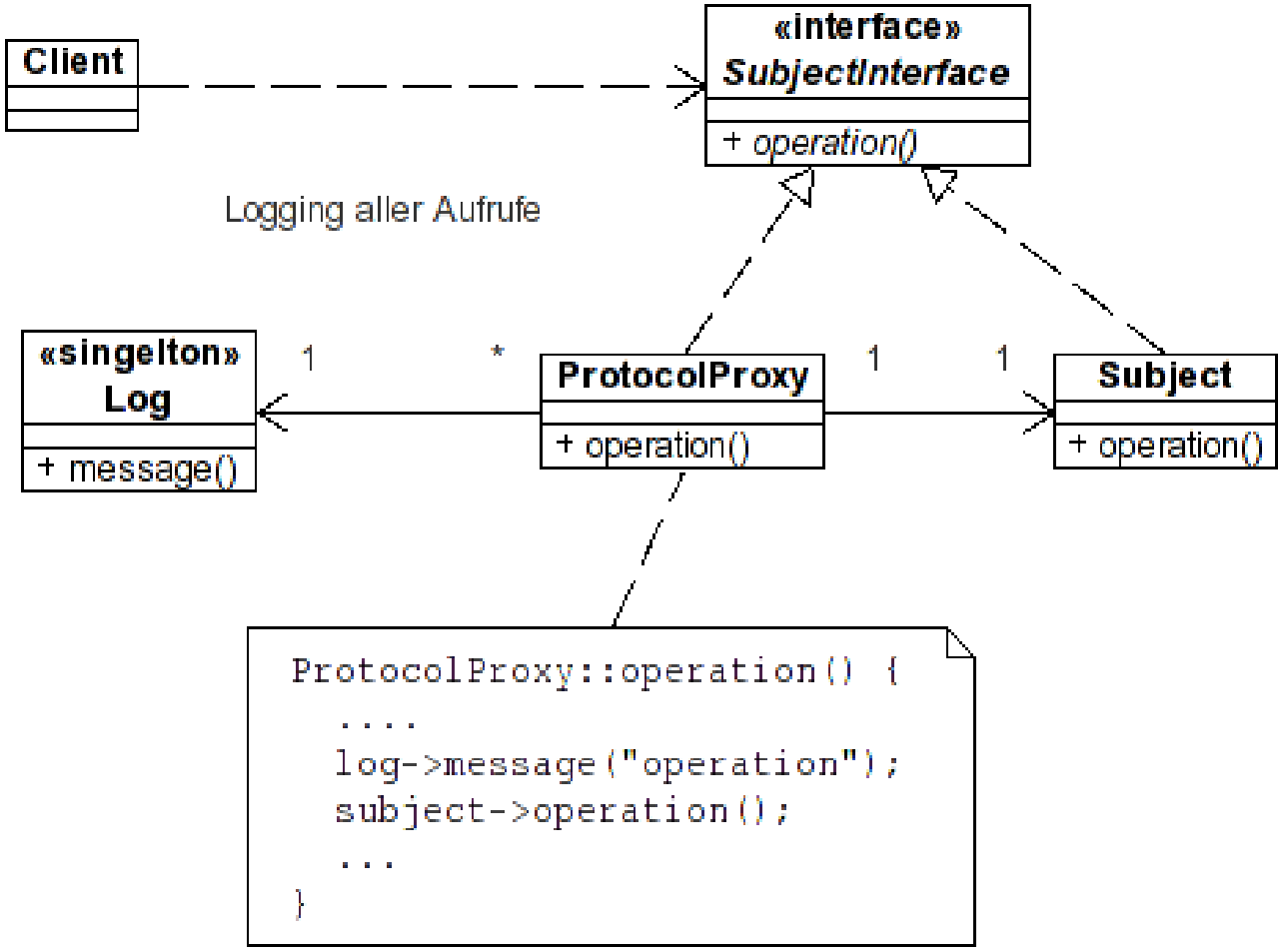


SmartReference




```
SmartReference::operation() const {
  service->operation();
}

SmartReference::changeOperation() {
  service=service->clone();
  service->operation();
}
```

ProtocolProxy (LogProxy)



Anti-Pattern: Cut and Paste

- Problem: Die Wiederverwendung von Code wird durch Kopieren erreicht. Dadurch werden z.B. Fehler mit kopiert und bei Korrektur nicht mit korrigiert.
- Idee: Benutzung von Methoden der Programmiersprache zur Implementierung von Wiederverwendung.
- Möglichkeiten: Unterprogramme, Vererbung, Bibliothek, Komponente, Template

Anti-Pattern: Latenzprobleme - Falsche Aufteilung der Applikation

- Problem: Die Aufteilung der Applikation führt zu einer erheblichen Anzahl von Kommunikationsschritten mit Antwort (syncron). Dabei sind die beteiligten Systeme die meiste Zeit mit dem Warten auf Kommunikation beschäftigt. Mit keinem technischen Mittel läßt sich dagegen etwas ausrichten.
- Idee: Vermeidung von 'Roundtrips'
- Möglichkeiten: Asyncrone Operationen, Proxy, größere Granularität in der Kommunikation

Objektorientierte Softwareentwicklung

Vorlesung 2

Spezifikation und Planung mit OO-Mitteln oder

Warum viele Softwareprojekte das Falsche entwickeln!

Ohne Spezifikation? (0)

- „Machen Sie uns ein Programm für den ATM!“
- „Messen der Radialkraftschwankung implementieren“
- „Erstellen Sie einen PIM“
- „Wir brauchen für unsere Plätzchen-Maschine ein Betriebsprogramm“
- „Unser CAD-System erzeugt ´dwg-Dateien´ und unser Plotter kann die nicht plotten. Kann man da etwas tun?“
- „Wir haben schon den kompletten Taupunktregler entwickelt – jetzt fehlt nur noch die Software!“

Ohne Spezifikation? (1)

Charakter der Entwicklung:
Studie oder Produktentwicklung?

Ohne Spezifikation? (2)

Anwender:
Profi oder Amateur?

Ohne Spezifikation? (3)

Mittel und Werkzeuge:
Platin oder Plaste?

Ohne Spezifikation? (4)

Resourcen:
GROß oder klein?

Vor der Spezifikation

- Die Sprache der Anwendung erlernen (Domainsprache)
- Glossar als Grundlage der Kommunikation
- Beispiele
  - ATM (Geldautomat)
  - RKS (Radialkraftschwankung)
  - Cash Flow (Umsatz+Kosten?)

Spezifikation (formale Spezifikation)

- Welche Probleme soll die Software lösen?
- (Kann man diese Probleme anders lösen als durch Software?)
- Was muss das Softwaresystem als Ganzes leisten?
- Welche konkreten Teilschritte führen zur Lösung?
- Welche Interaktionen sind mit dem System geplant?
- Welche Abläufe sind im System zu modellieren?
- Welches Modell soll aus Nutzersicht entstehen (Use Cases)

Spezifikation (informale Spezifikation)

- Festlegung der Anforderungen
  - Charakter des Produkts (Prototyp …. Vollprodukt)
  - Zeithorizont
  - Anwenderklasse
- Festlegung der Systemumgebung
  - Rechnerarchitekturem
  - Resourcen
  - Entwicklungsumgebung
  - Ablaufumgebung
  - Mengengerüste
  - Sprache(n)

OO-Analyse

Warum ist Softwareentwicklung so kompliziert?

- Komplexität
- Fehlerträchtigkeit
- Unüberprüfbarkeit
- Zwang zur Zusammenarbeit oder Auseinandersetzung

OO-Analyse (2)

Wie werden in der Natur Probleme mit Komplexität gelöst?

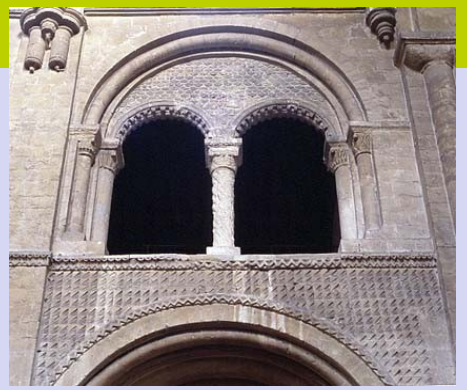
1. Kleine überschaubare Teilsysteme
2. Selbstorganisation
3. Selbstkontrolle
4. Hierarchien
5. Entwicklung vom Einfachen zum Komplexen
6. Wiederverwendung von Mustern
7. Evolutionäre Entwicklung
8. Verschiedene Eigenschaften bei verschiedenen Betrachtungen

OO-Analyse (3)

OO-Paradigmen

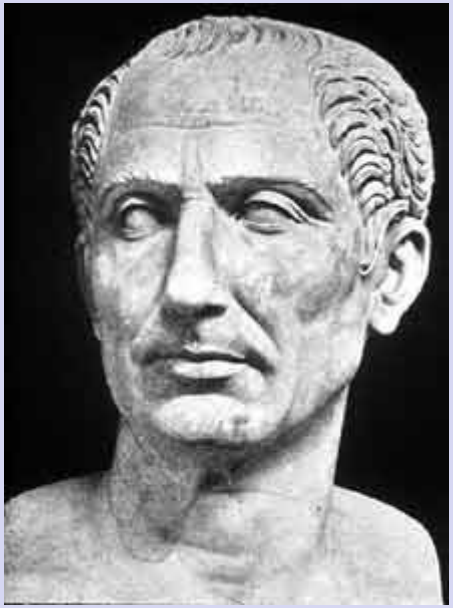
- Encapsulation
- Polymorphismus
- Inheritance
- Abstract Data Types

Composite Pattern



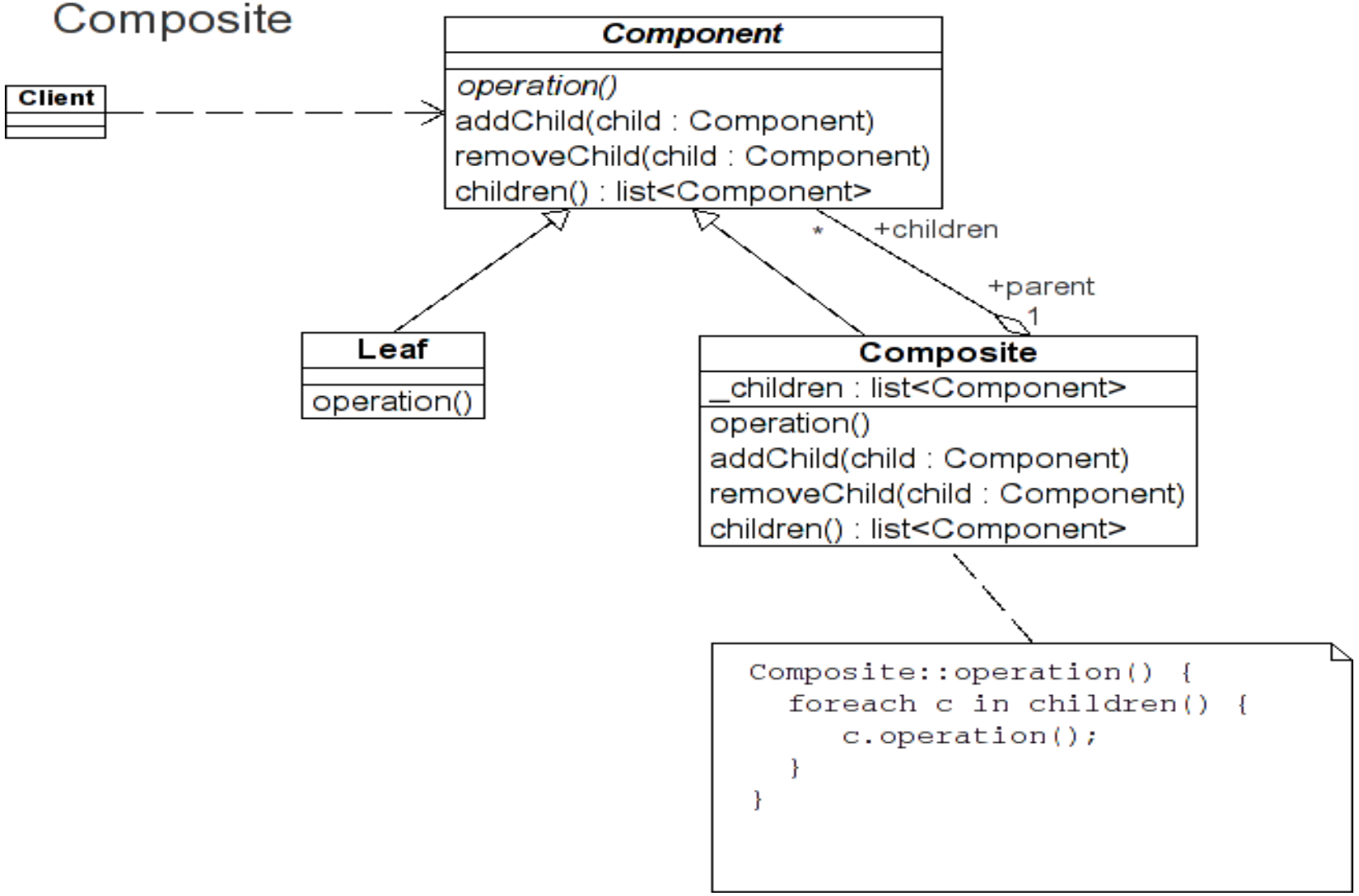
- Problem: Operationen sollen auf eine große Anzahl (auch verschiedener) Objekte angewandt werden. Dabei sollen sich Gruppen von Objekten genau so verhalten wie einzelne.
- Idee: Die Objekte werden in Gruppierende und Einzelobjekte unterschieden. Beide implementieren das gleiche Interface und sind dadurch für den Benutzer nicht zu unterscheiden. Es entsteht ein Objektbaum.
- Teilnehmer: Client, Composite, Compositum, Leaf

Aufbau des römischen Heers zur Zeit Caesars (Composite)



1 Legion	= 10 Kohorten	= 6000 Mann
1 Kohorte	= 3 Manipel	= 600 Mann
1 Manipel	= 2 Zenturien	= 200 Mann
1 Zenturie	= 10 Decurien	= 100 Mann
1 Decurie	(Contubernium)	= 10 Mann

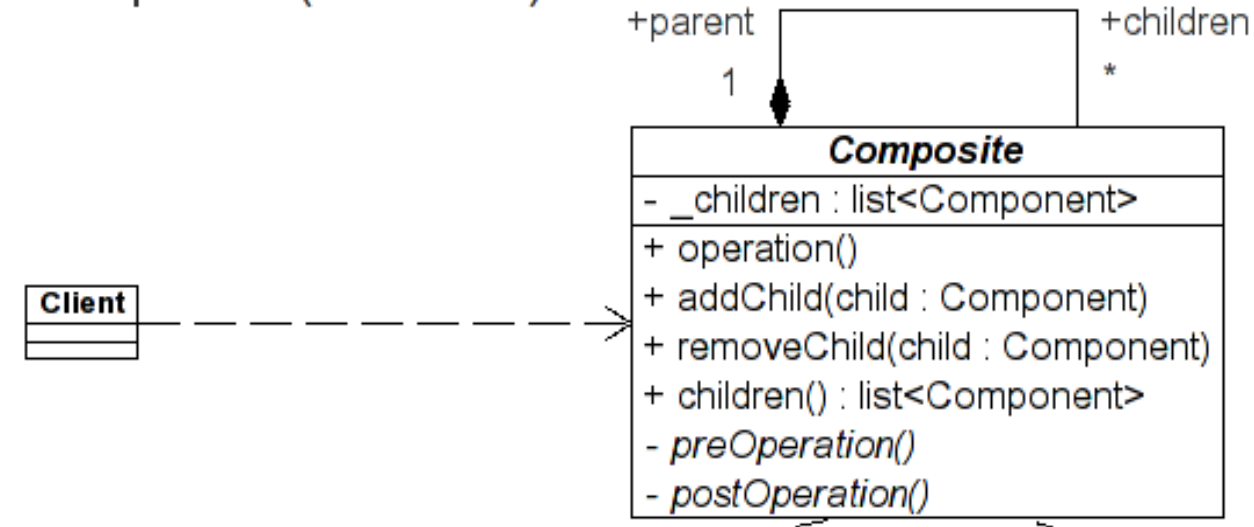
Composite


Component
operation()
addChild(child : Component)
removeChild(child : Component)
children() : list<Component>
Client
Leaf
operation()
Composite
_children : list<Component>
operation()
addChild(child : Component)
removeChild(child : Component)
children() : list<Component>
* +children
+parent
1
Composite::operation() {
foreach c in children() {
c.operation();
}
}

```
#ifndef Component_h
#define Component_h

// Datei Component.h

#include <list>
using namespace std;

class Component { // abstract
public:
 virtual void operation()=0;
 virtual void addChild(Component *c) {}
 virtual void removeChild(Component *c) {}
 list<Component*> children() { return list<Component*>; }
};

#endif
```

```
#include „Component.h“

class Composite : public Component {
public:
  virtual void operation() {
    for(list<Component*>::const_iterator i=
      _children.begin();
      i!=_children.end();
      ++i
    ) {
      Component* c=(*i);
      c->operation();
    }
  }
  void addChild(Component *c) {_children.push_back(c); }
  void removeChild(Component *c) { _children.remove(c); }
  list<Component*> children() { return _children; }
private:
  list<Component*> _children;
};
```

Composite (Variante)




```
Composite::~Composite() {
  while(_children.size()) {
    Composite* c=_children.first();
    _children.removeFirst();
    delete c;
  }
}
```

```
Composite::operation() {
  preOperation();
  foreach c in _children {
     c.operation();
  }
  postOperation();
}
```

Anti-Pattern: Aufblasen der Basisklasse

- Problem: Jede weitere Operation im Composite muss mit einer Erweiterung der Basisklasse (Composite) erkauft werden. Alle abgeleiteten Klassen 'erfahren' so von Spezialwissen das vielleicht nur einige Klassen betrifft. damit wird das Prinzip der Unabhängigkeit von Implementierungen durchbrochen.
- Möglichkeiten: Sorgfältige Planung der Operationen (siehe Variante), Visitor Pattern

Objektorientierte Softwareentwicklung

Vorlesung 3
Formale Spezifikation und
UML-Klassendiagramm

Was bedeutet formal?

- Über Personen und Zeit hinweg stabil
- Lesbar ohne Kontext-Wissen

Beispiel: Biologie (Art, Gruppe, Ordnung, Unterordnung, Gattung sind sprachlich alle austauschbar)

- Austauschbar und verbindbar
- Kann mit Regeln bearbeitet werden
- Kann automatisch verarbeitet werden

Idee: Metaprogrammierung

- Schaffen einer Meta-Programmiersprache zum Spezifizieren von Software
- Einbau vieler Aspekte der Spezifikation
- Möglichkeit zur automatischen Code-Generierung
- Möglichkeit zum automatischen Test

Beispiele für schon vorhandene Metaprogrammierung:

Compilergeneratoren, Automaten-Generator. UI-Generatoren

UML (1)

- UML ist eine Vereinbarung über Begriffe und deren Verwendung
- UML ist eine Vereinbarung über Symbole und deren Semantik
- Deshalb ist UML eine Sprache
- UML ist _keine_ Programmiersprache
- UML hat keine Lexik oder Grammatik im herkömmlichen Sinne (keine formale Sprache)

UML (2)

- Definiert Begriffe und Diagrammtypen:
2. Class-Diagram
3. UseCase-Diagram
4. State-Diagram
5. Sequence-Diagram
6. Collaboration-Diagram
7. Package-Diagram
8. Deployment-Diagram
9. Activity-Diagram

UML-Class Diagram

- Wichtigstes und komplexestes Diagram
- Zeigt die Klassen und ihre Beziehungen
- Kann konzeptionell oder zur Spezifikation eingesetzt werden

UML Class

Klasse

Klassenname
attributes
operations

Person
+ name : string = ""
+ matches(pattern : string) : void

UML Attributes

Attribute

[visibility]name[:type][=initialvalue]

Klassenattribute werden unterstrichen

UML Operations

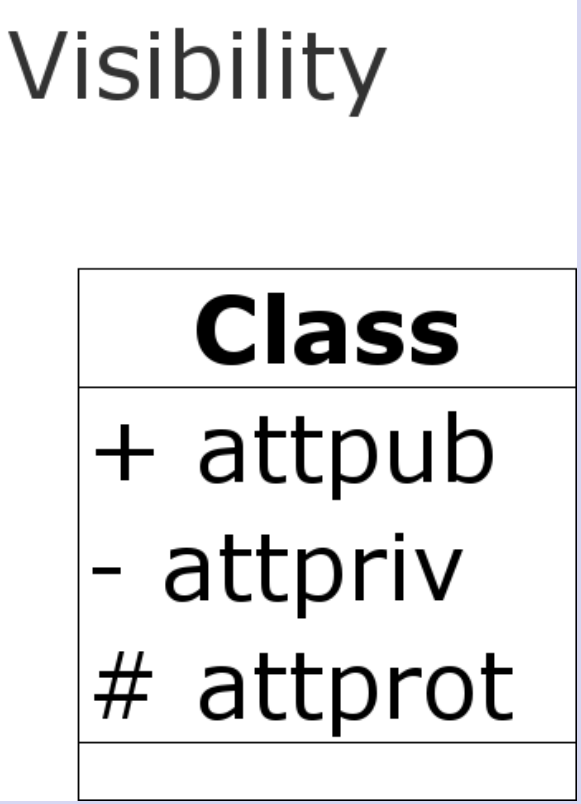
Operationen

[visibility]name([arguments])[:type]

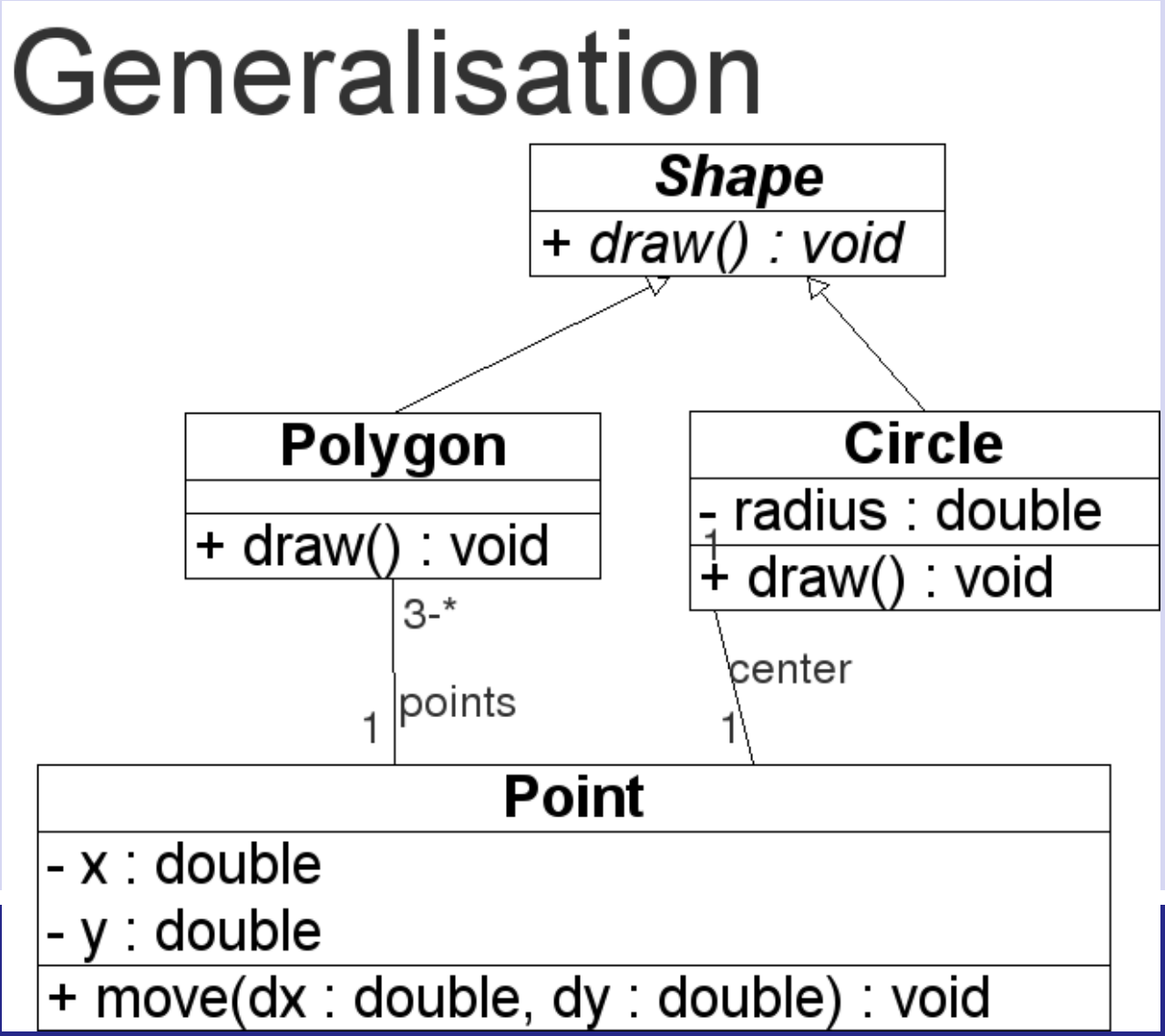
Klassenoperationen werden unterstrichen

abstrakte Operationen werden kursiv dargestellt

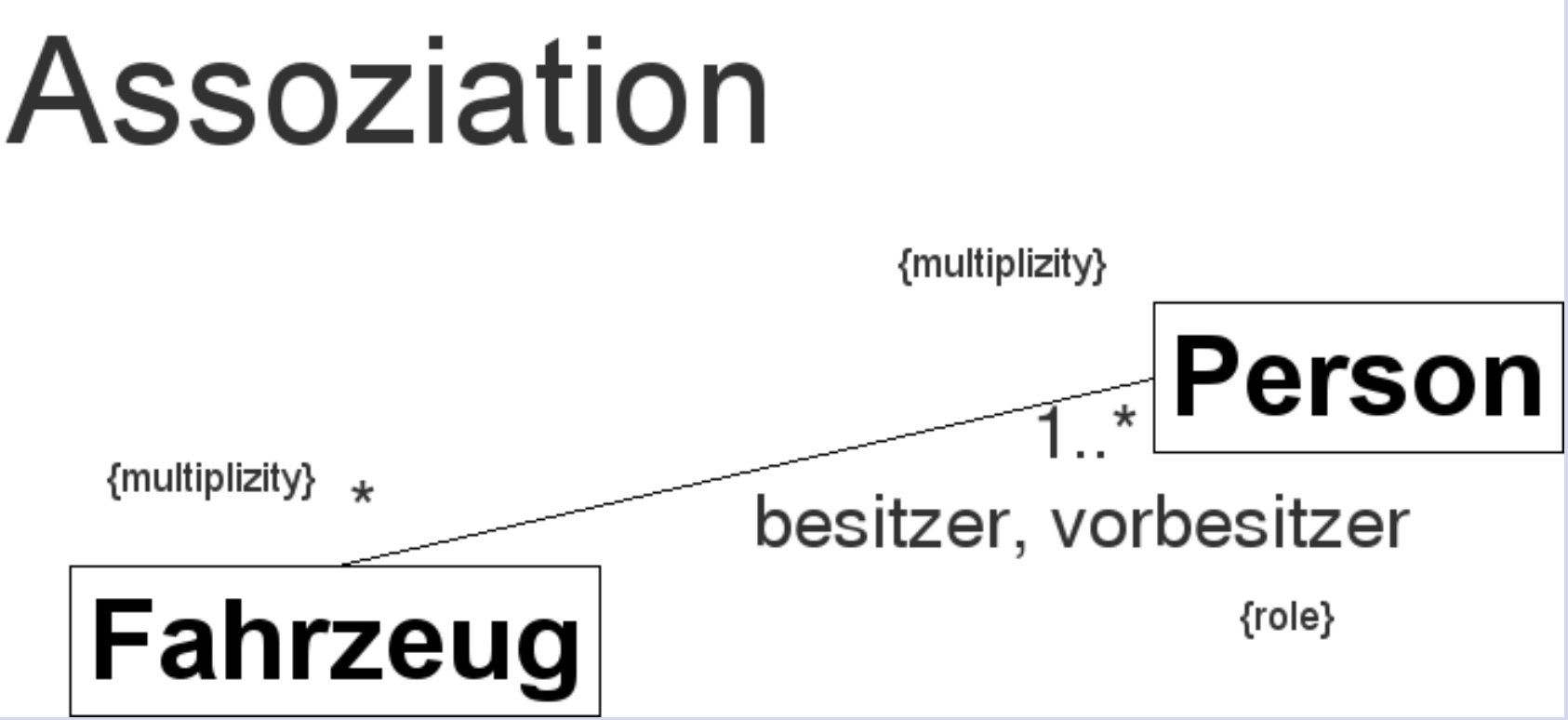
UML Visibility



UML Generalisation



UML Associations

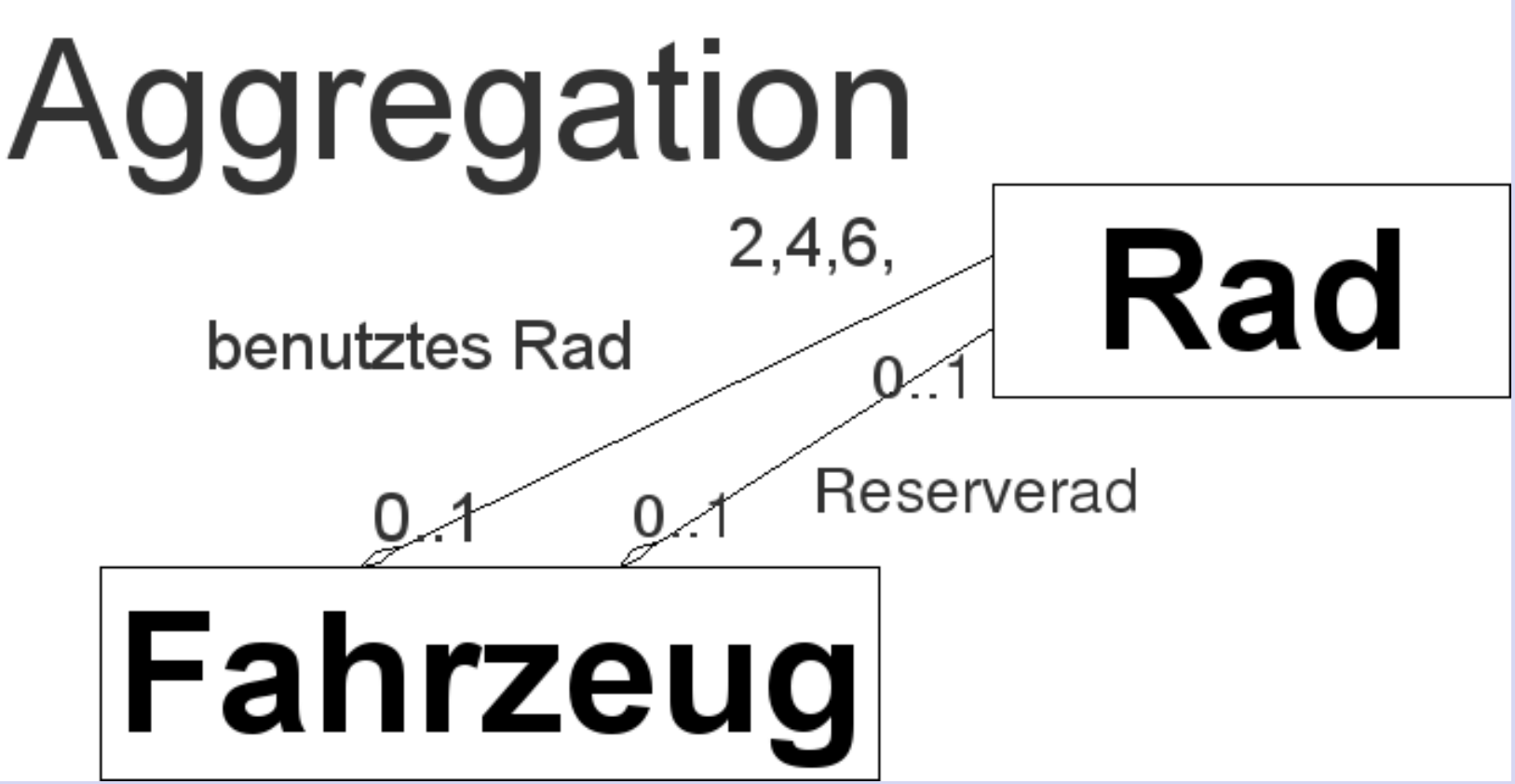


UML Multiplicity

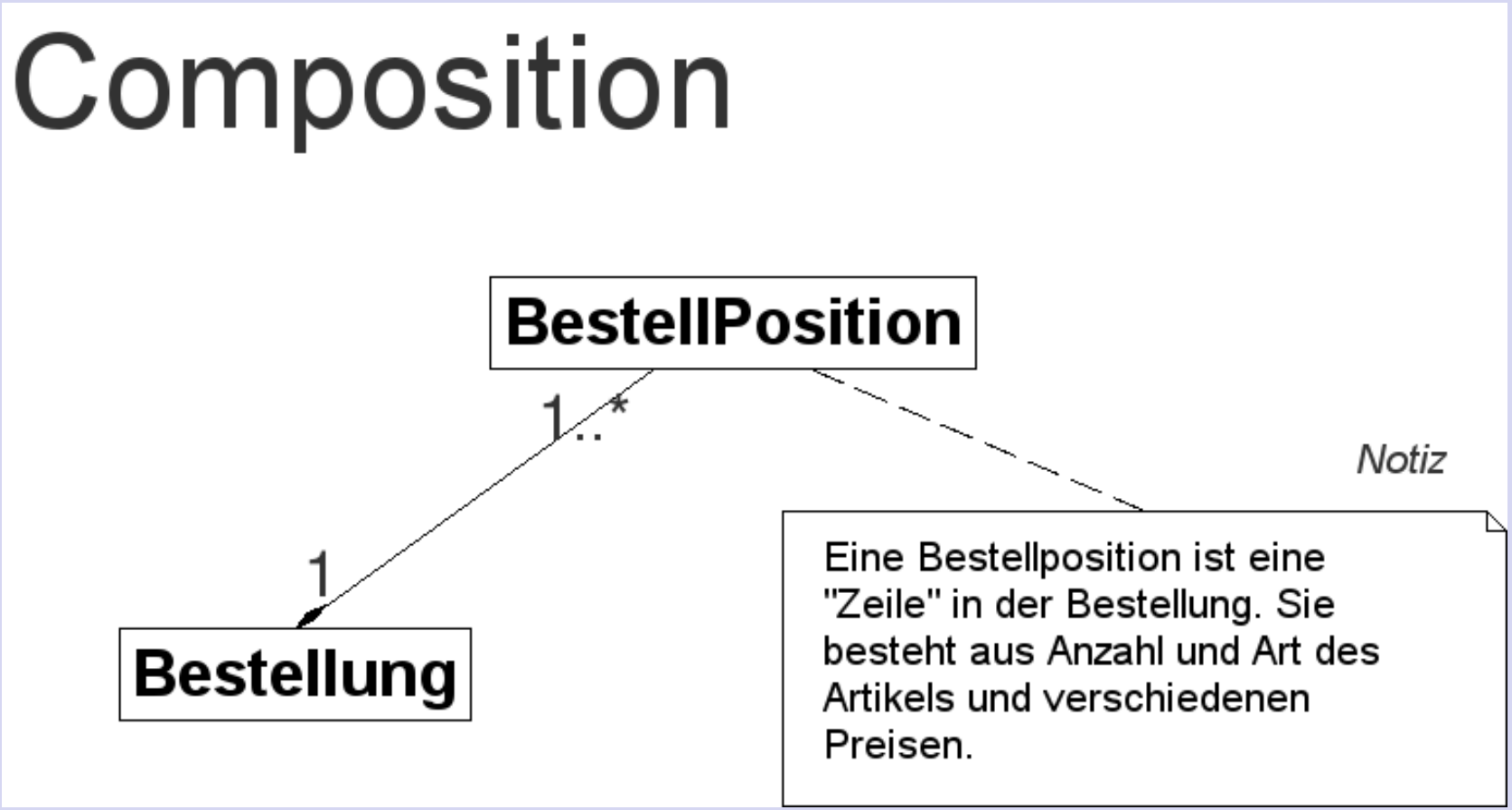
Gibt die Anzahl der Objekte auf der Verknüpfungsseite an:

1	exakt ein Objekt
0..1	optional
0..*	beliebig viele Objekte
1..*	mindestens ein Objekt
1,5-7	ein oder 5 bis 7 Objekte (eher selten)

UML Aggregation



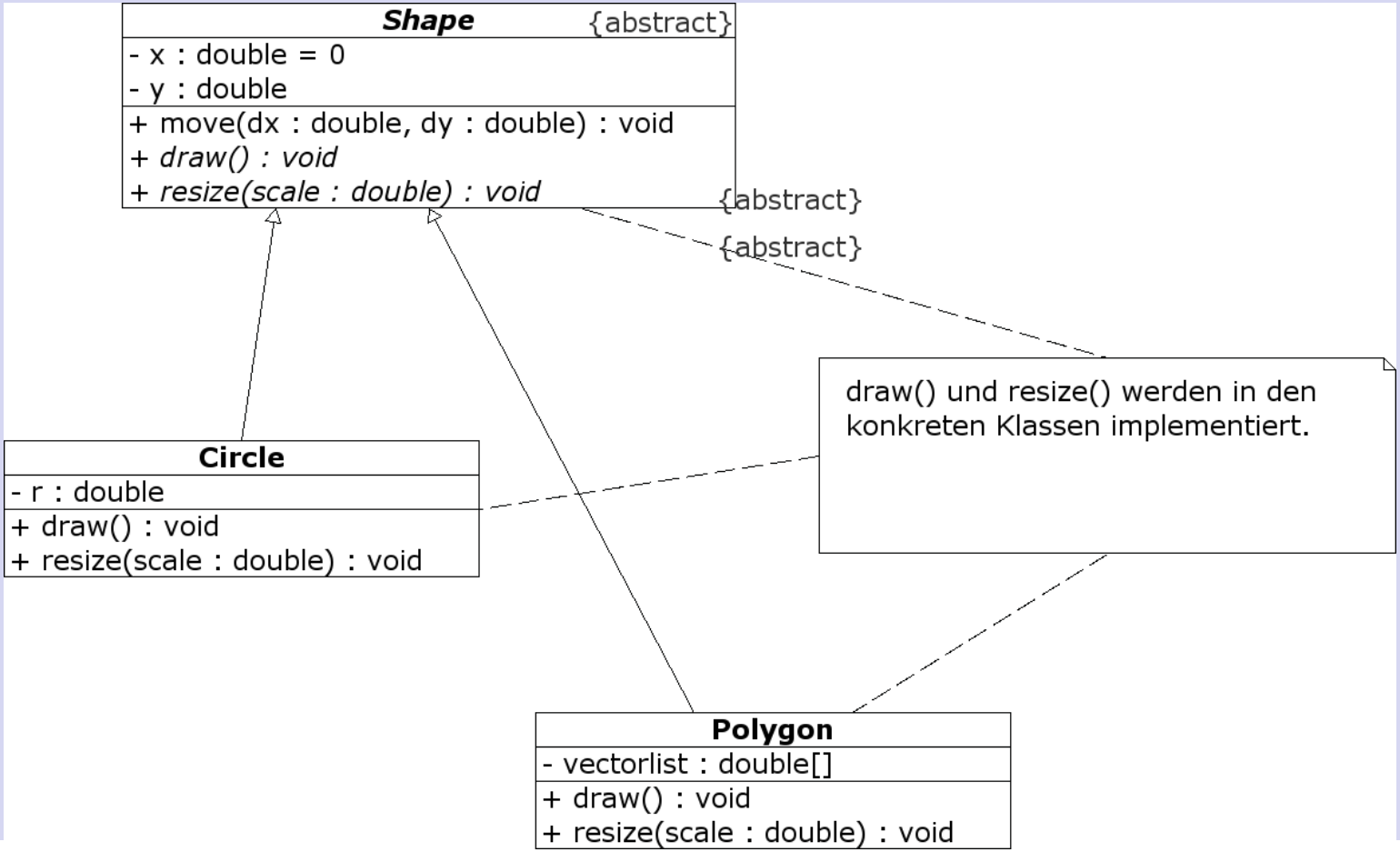
UML Composition



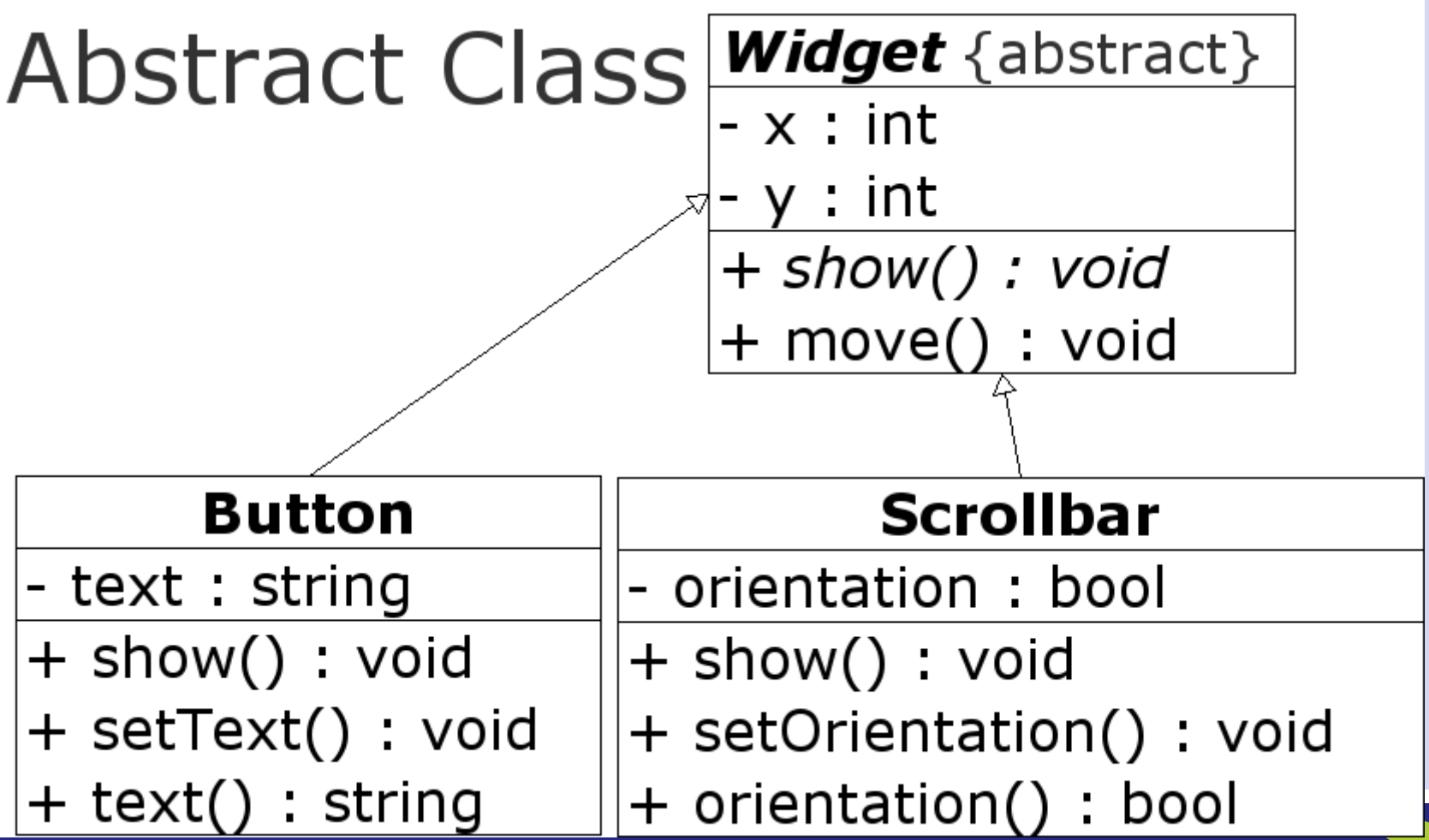
UML Abstract Classes

- Werden nie direkt erzeugt
- Dürfen alles enthalten was eine Klasse auch enthält
- Dürfen zusätzlich Operationen enthalten, die keine Methode (Implementierung) besitzen
- Werden mit kursiv/italic geschriebenem Klassennamen oder mit {abstract} gekennzeichnet

UML Abstract Classes



UML Abstract Classes



UML Abstract Classes (Beispiel)

```
class Widget {
public:
  Widget();
  virtual ~Widget();
  virtual move(int dx,int dy);
  virtual show()=0;
private:
  int x;
  int y;
};
```

UML Abstract Classes (Beispiel) cont.

```
class Button : public Widget {
public:
  Button(string _text);
  virtual ~Button();
  virtual show();
  void setText(string _text);
  string text() { return text; }
private:
  string text;
};
```

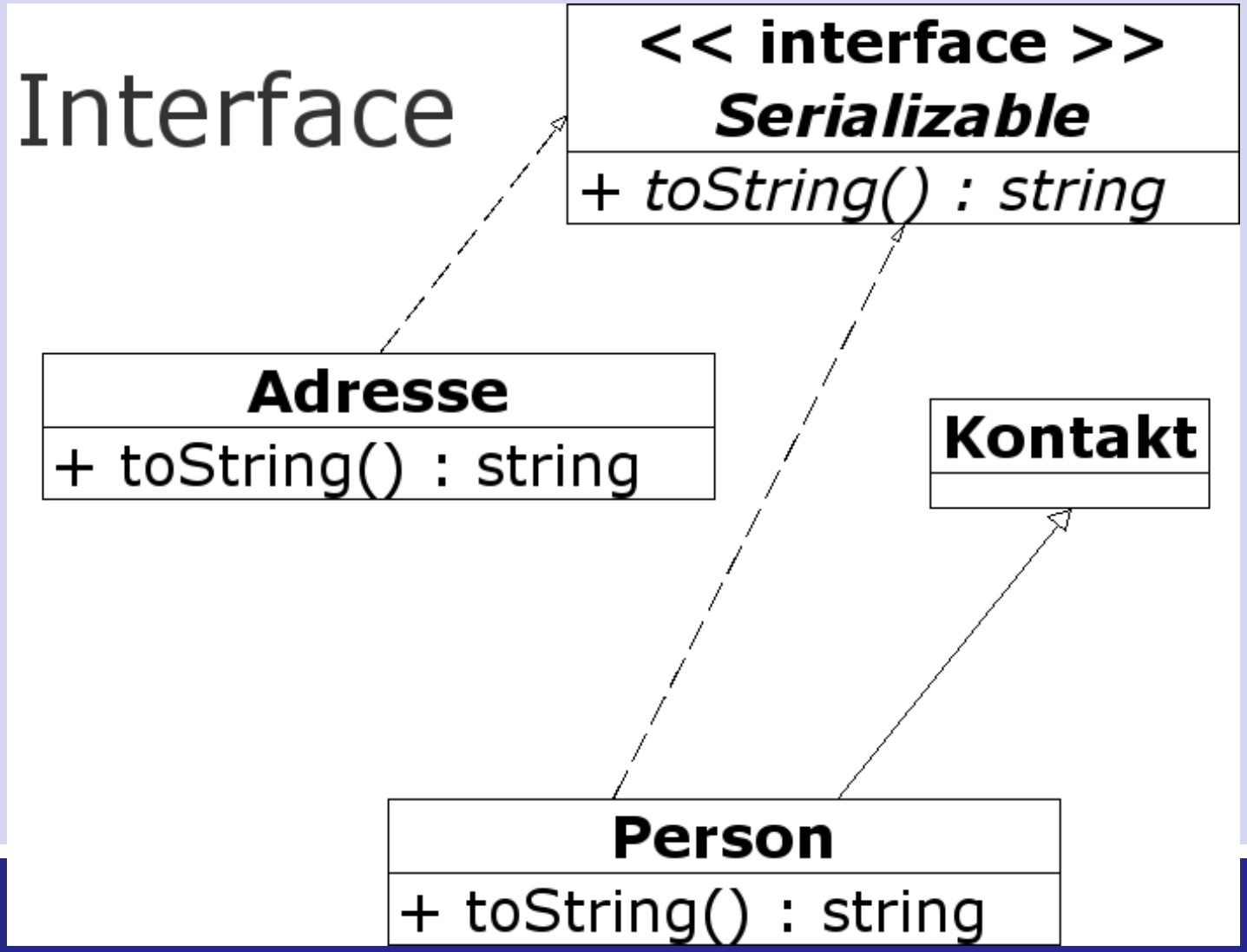
UML Abstract Classes (Beispiel) cont.

```
class Scrollbar : public Widget {
public:
  Scrollbar();
  virtual show();
  void setOrientation(bool _orientation);
  bool orientation() { return orientation; }
private:
  bool orientation;
};
```

UML Interfaces

- Werden nie erzeugt
- Beschreiben einen Aspekt des Verhaltens
- Oft im Zusammenhang mit „Ich kann“ oder „able“ z.B. „storable“ oder „serializable“
- Können keine Attribute enthalten
- Enthalten nur Operationen „ohne Realisierung“
- Werden in C++ durch Klassen abgebildet, die keine Felder und nur pure virtuelle Methoden enthalten.
- Klassen leiten von Ihnen ab, und implementieren diese Methoden

UML Interfaces Klassendiagramm



UML Interfaces (Beispiel)

```
class Serializable {
public:
  virtual string toString()=0;
};
```

UML Interfaces (Beispiel)

```
class Adresse: public Serializable {
public:
   string name;
   string strasse;
   string ort;
   string plz;
   string land;
   virtual string toString();
};
```

UML Interfaces (Beispiel) cont.

```
class Person:public Serializable,public Kontakt {
public:
  string vorname;
  string anrede;
  Adresse *adresse;
  Telefon *telefon;
  virtual string toString();
};
```

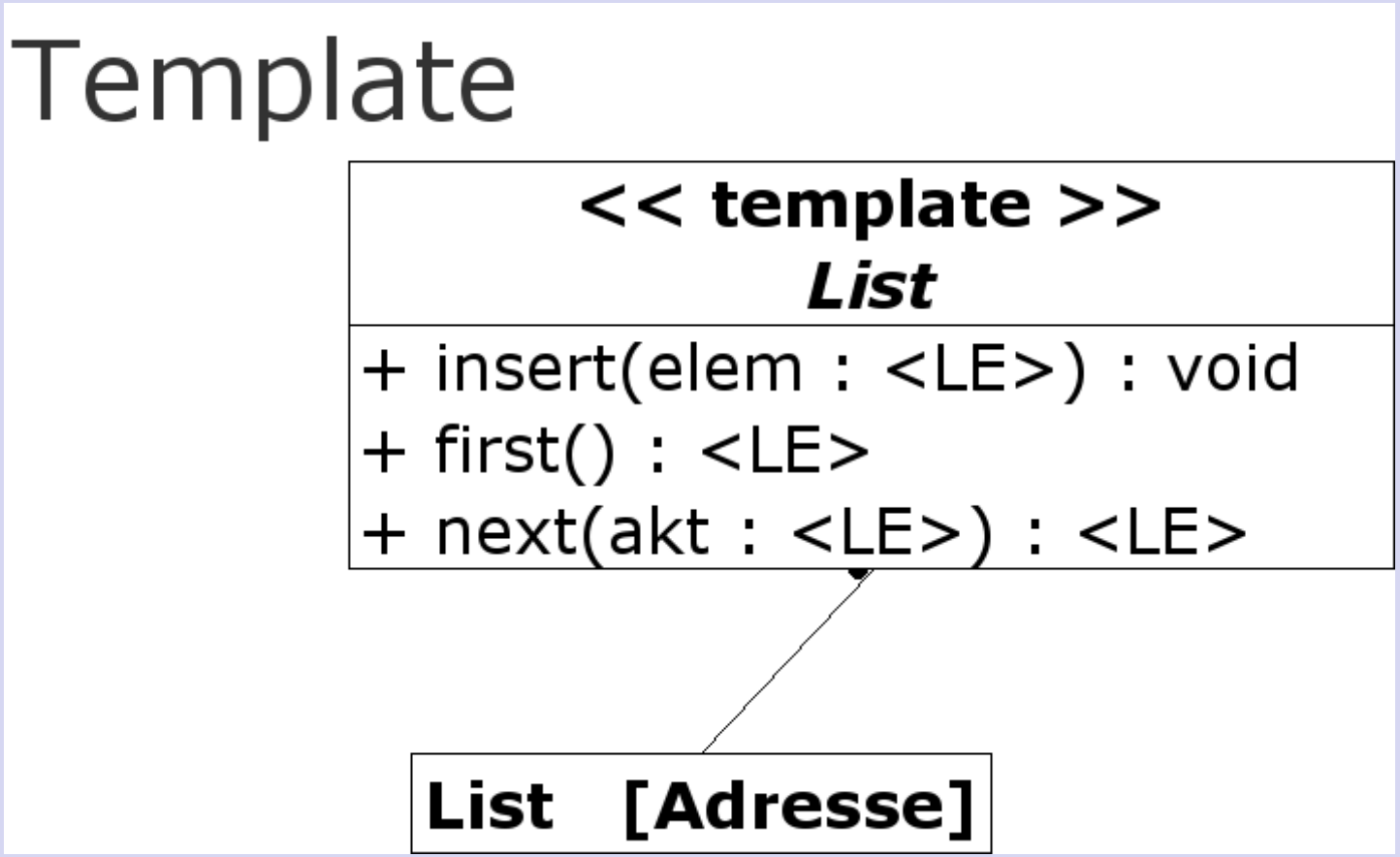
UML Interfaces (Beispiel) cont.

```
string Person::toString() {
  string result=´´;
  result.append(vorname);
  result.append(anrede);
  if(adresse) {
    result.append(adresse->toString());
  }
  if(telefon) {
    result.append(telefon->toString());
  }
  return result;
}
```

UML Parametric Classes (Templates)

- Sind Klassen von Klassen
- Werden durch Angabe konkreter Parameter zu Klassen (auch Interfaces, abstrakte Klassen)
- Werden in C++ als Templates realisiert
- Der Compiler erzeugt die entsprechenden Klassen automatisch beim Übersetzen
- Werden oft in Containern eingesetzt (STL)
- Sollten vorsichtig eingesetzt werden, da viel Code entstehen kann
- Faustregel: inline class

UML Parametric Classes (Templates)
Klassendiagramm



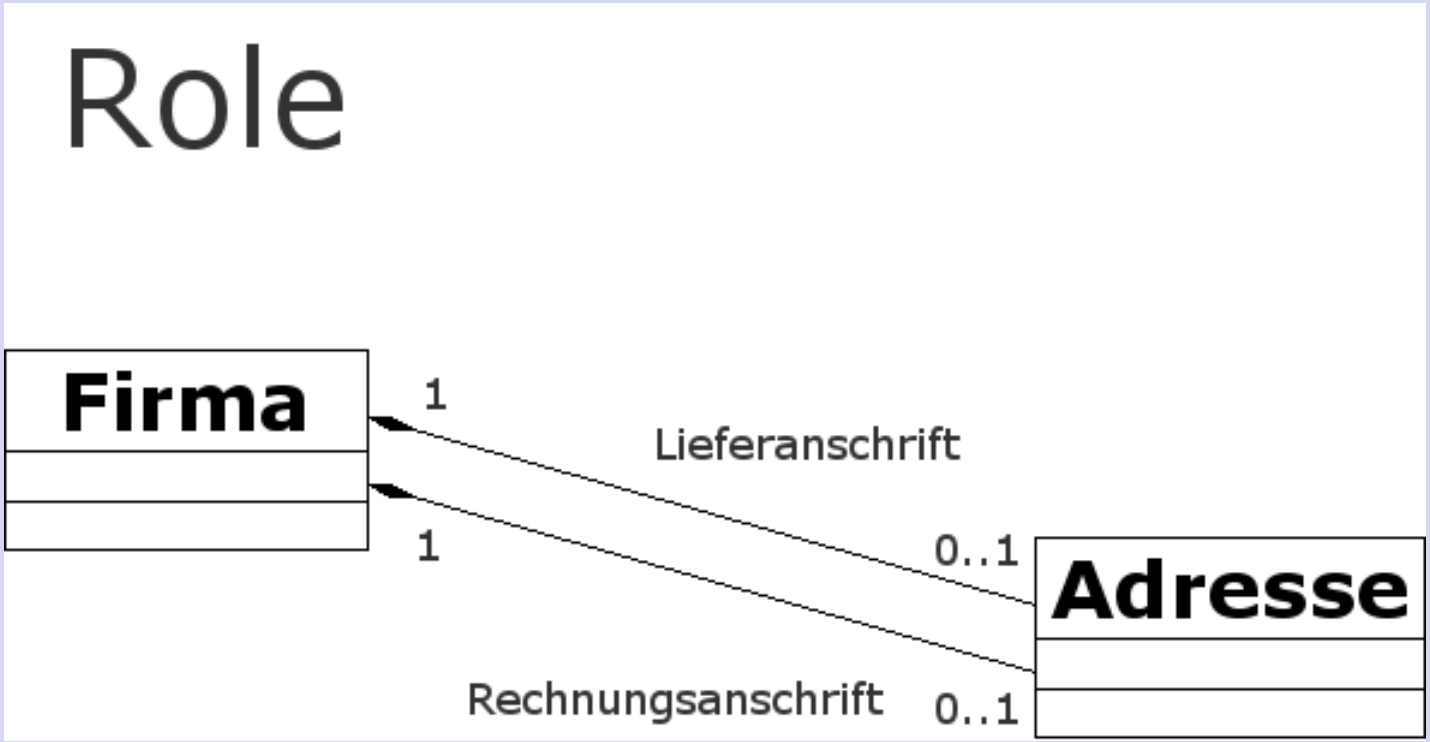
UML Templates (Instanciation)

```
#include "list"
#include "string"
class Adresse {
public:
  Adresse(string _name):name(_name) {
    adressList.push_front(this);
  }
  ~Adresse() { adressList.remove(this); }
  string name;
  typedef list<Adress*> AdressList;
  static AdressList adressList;
};
```

UML Roles

- Gibt Assoziationen eine Namen, der sich vom Namen der „anderen Seite“ unterscheidet
- Gibt normalerweise den Namen des Attributs zur Realisierung der Assoziation an
- Mehrere Assoziationen zwischen den gleichen Klassen heissen unterschiedlich
- Wenn die Rolle variabel ist: siehe qualifizierte Assoziation und Assoziationsklasse
- Beispiel: Person kann Ansprechpartner sein (PIM)

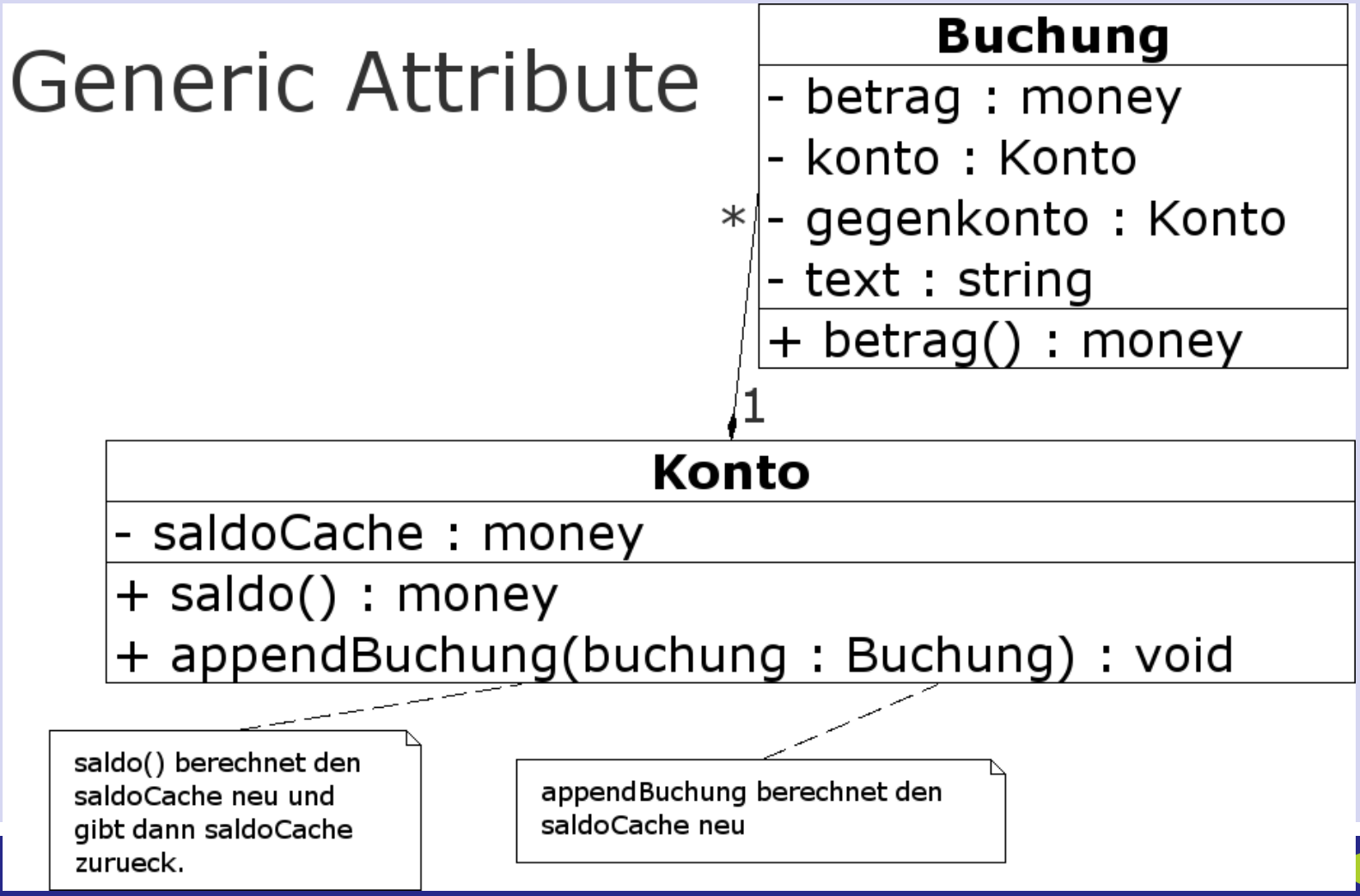
UML Roles Klassendiagramm



UML Generic Attributes

- Oft werden Attribute aus anderen berechnet
- Sie dienen dann als „Cache“ und sollten auch so benannt werden
- Generische Attribute sollten nur über Operationen lesbar und nie schreibbar sein
- Werden beim Ändern von Attributen mit geändert
- Müssen eine Funktion zum Neuberechnen besitzen
- Beispiel: Konten mit Saldo

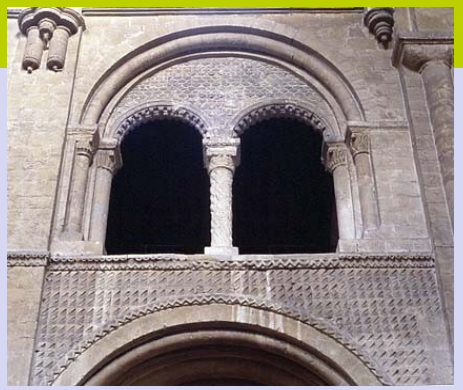
UML Generic Attributes Klassendiagramm



UML Klassendiagramm - Weitere Konzepte

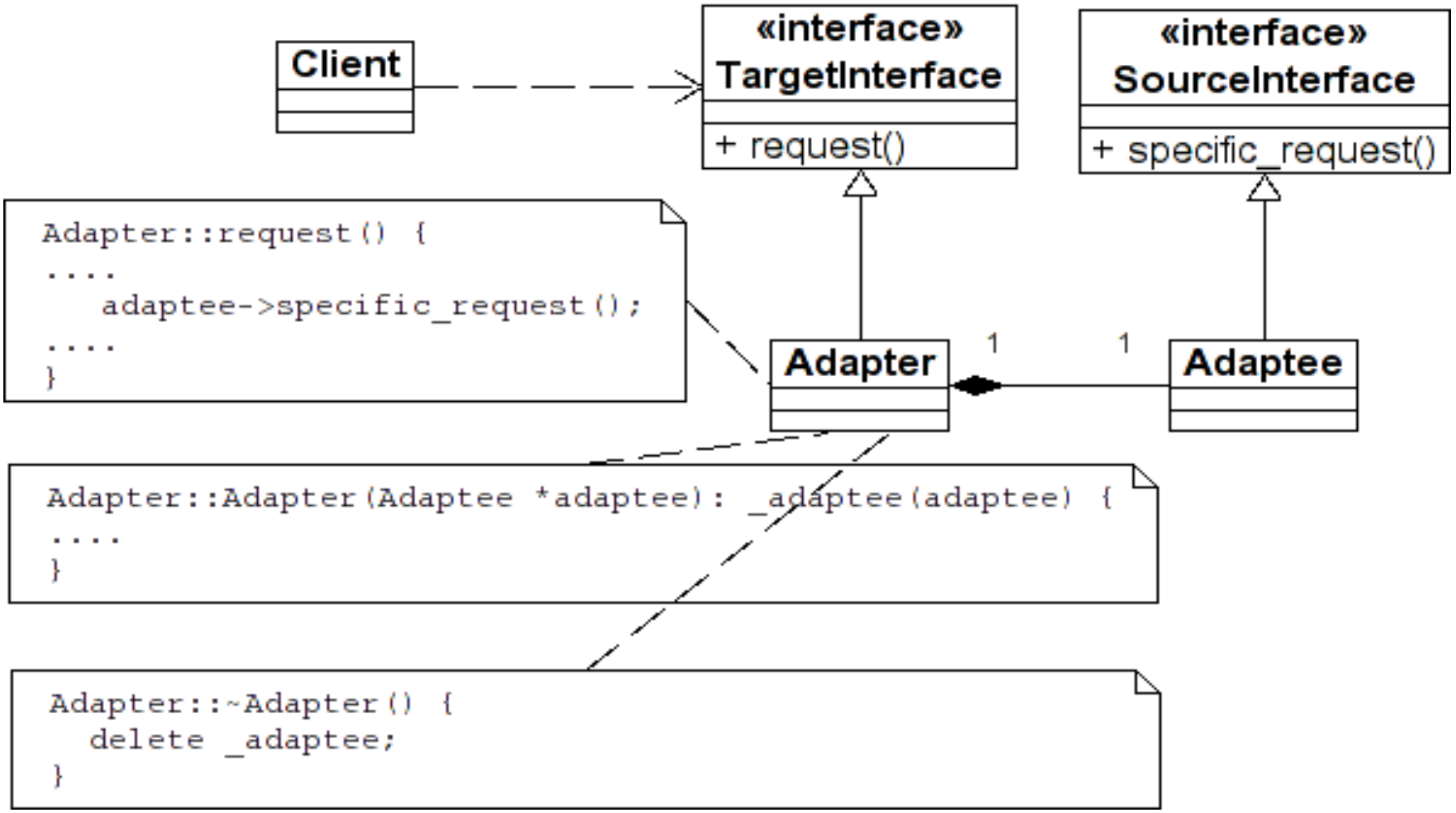
- UML Generic Associations
- UML Navigability
- UML Stereotypes
- UML Constraints
- UML Qualified Association
- UML Associationclass

Adapter Pattern

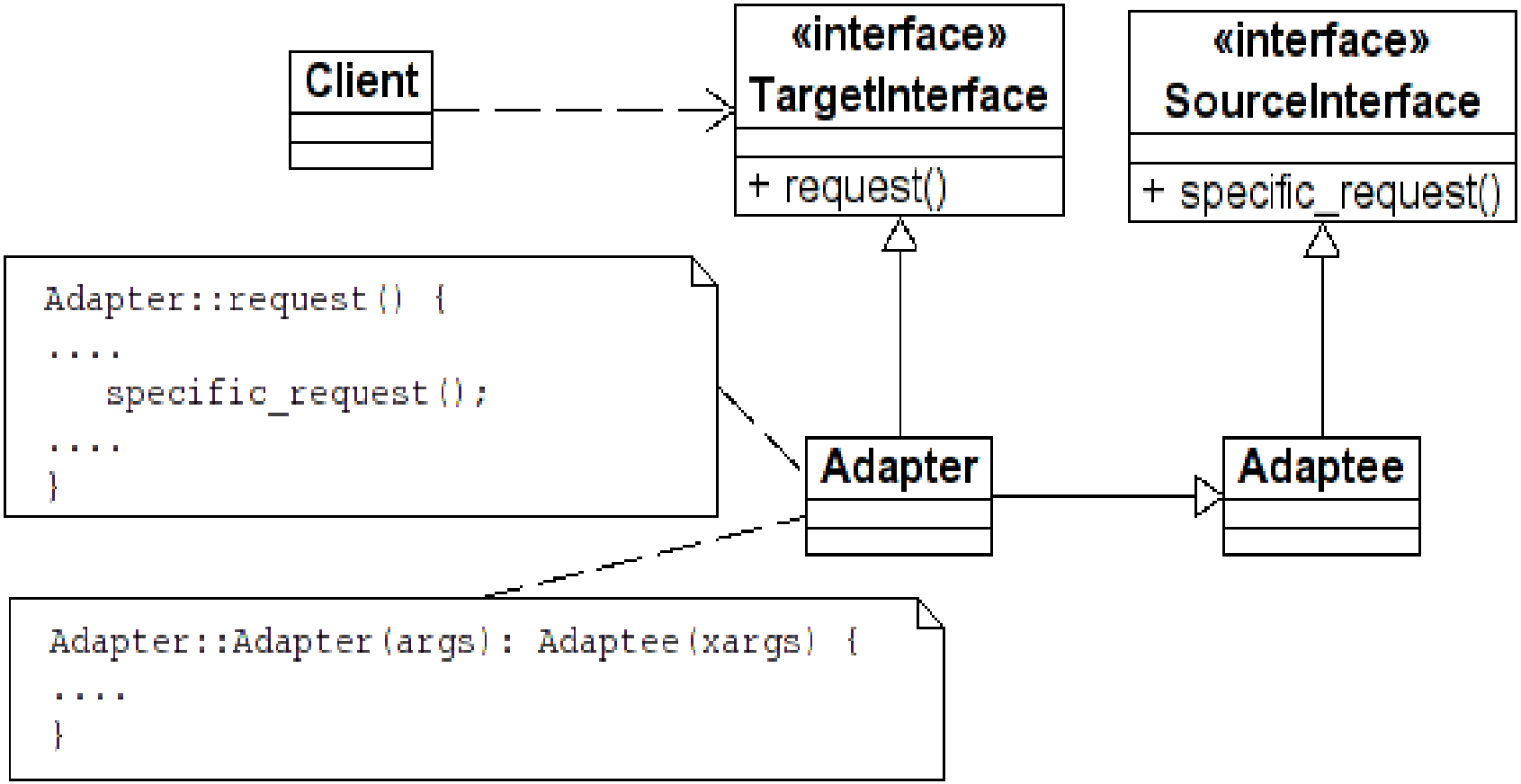


- Problem: Ein bestimmtes Interface wird erwartet. Die Funktionalität wird aber von einer Klasse angeboten, die über ein anderes Interface verfügt
- Idee: Es wird ein Adapter gebaut, der das Zielinterface implementiert. Er besitzt (als Kompositum oder Superklasse) ein Adaptee. Alle Operationen des Zielinterfaces werden als Operationen des Quellinterface ausgedrückt.
- Teilnehmer: Client, Adapter, Adaptee, Quell- und Zielinterface

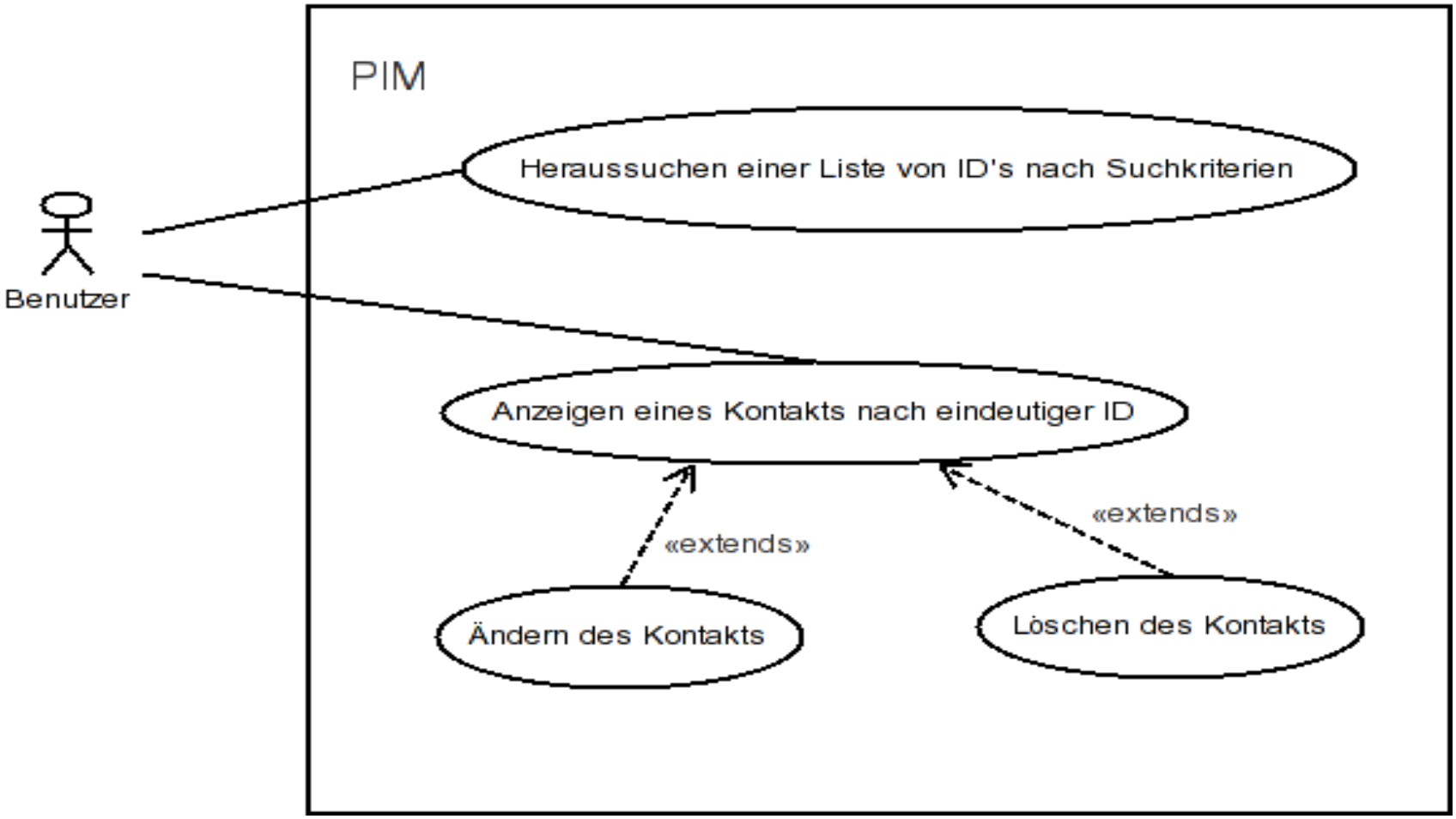
Adapter (Composition)

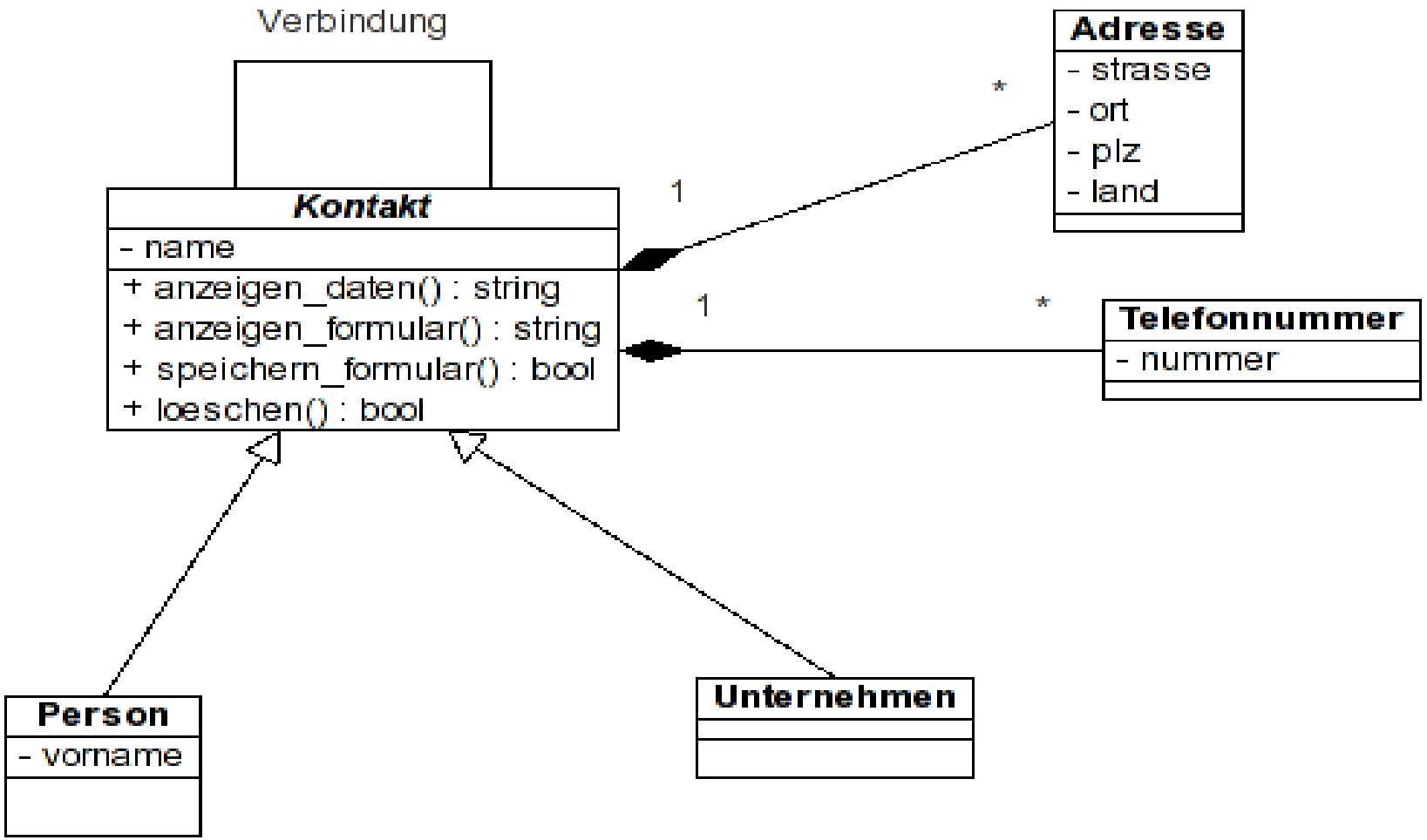


Adapter (Inheritance)



Suche, Anzeige und Bearbeiten von Kontakten




Verbindung
Adresse
- strasse
- ort
- plz
- land
*
1
Kontakt
- name
+ anzeigen_daten() : string
+ anzeigen_formular() : string
+ speichern_formular() : bool
+ loeschen() : bool
1
*
Telefonnummer
- nummer
Person
- vorname
Unternehmen

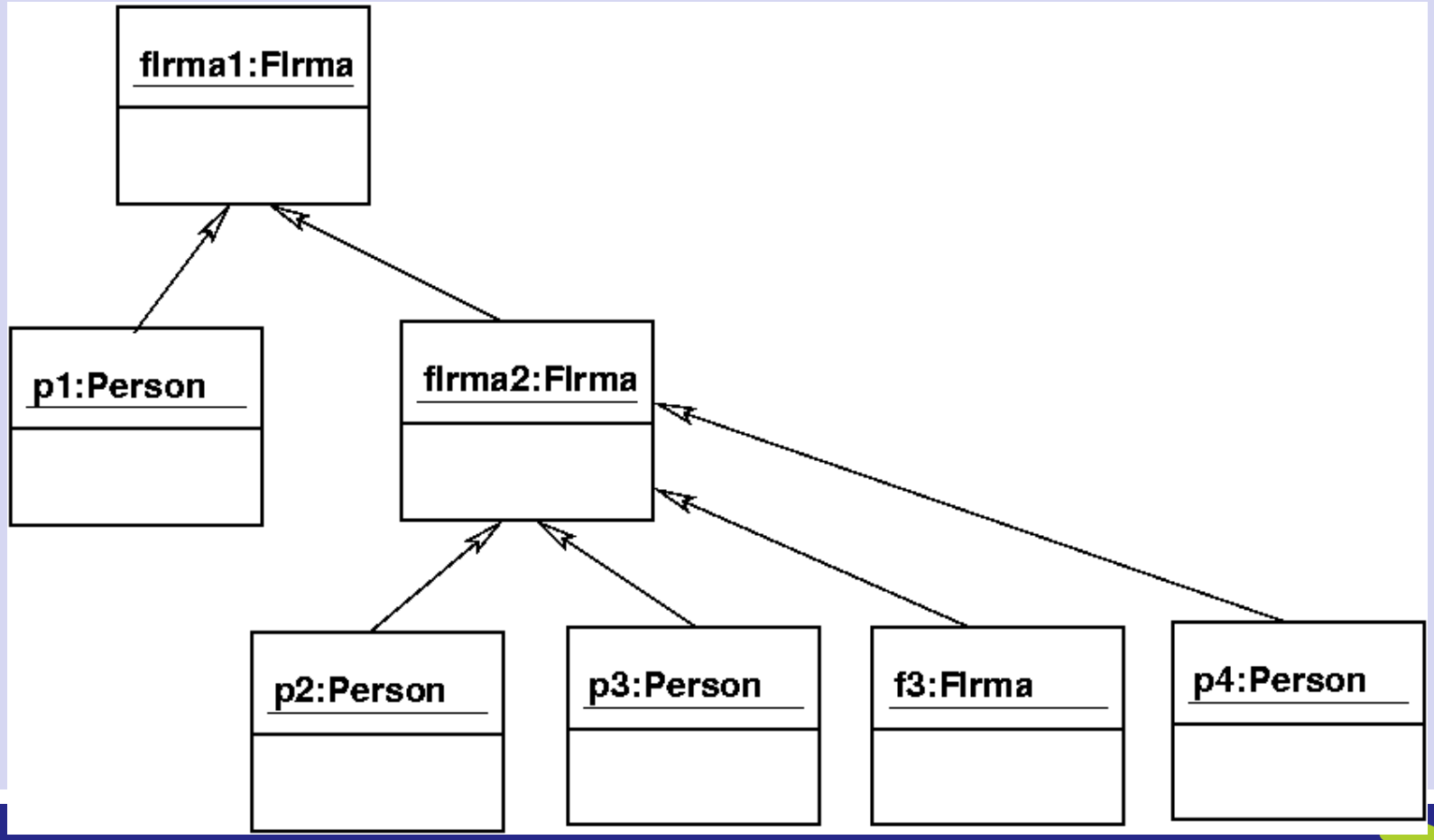
Objektorientierte Softwareentwicklung

Vorlesung 5
UML Diagramme (Paket, Objekt)

UML Object Diagram

- Objektdiagramme zeigen Objekte statt Klassen oder Packages
- Sie bilden den Zustand eines Programmes ab und zeigen die Kommunikation zwischen den einzelnen Objekten
- Sie werden zum Zeigen komplexer Relationen – insbesondere rekursiver Relationen benutzt
- Es wird die gleiche Symbolik wie im Klassendiagramm verwendet, die Instanzen werden durch unterstrichene „Name:Klasse“ gezeigt.

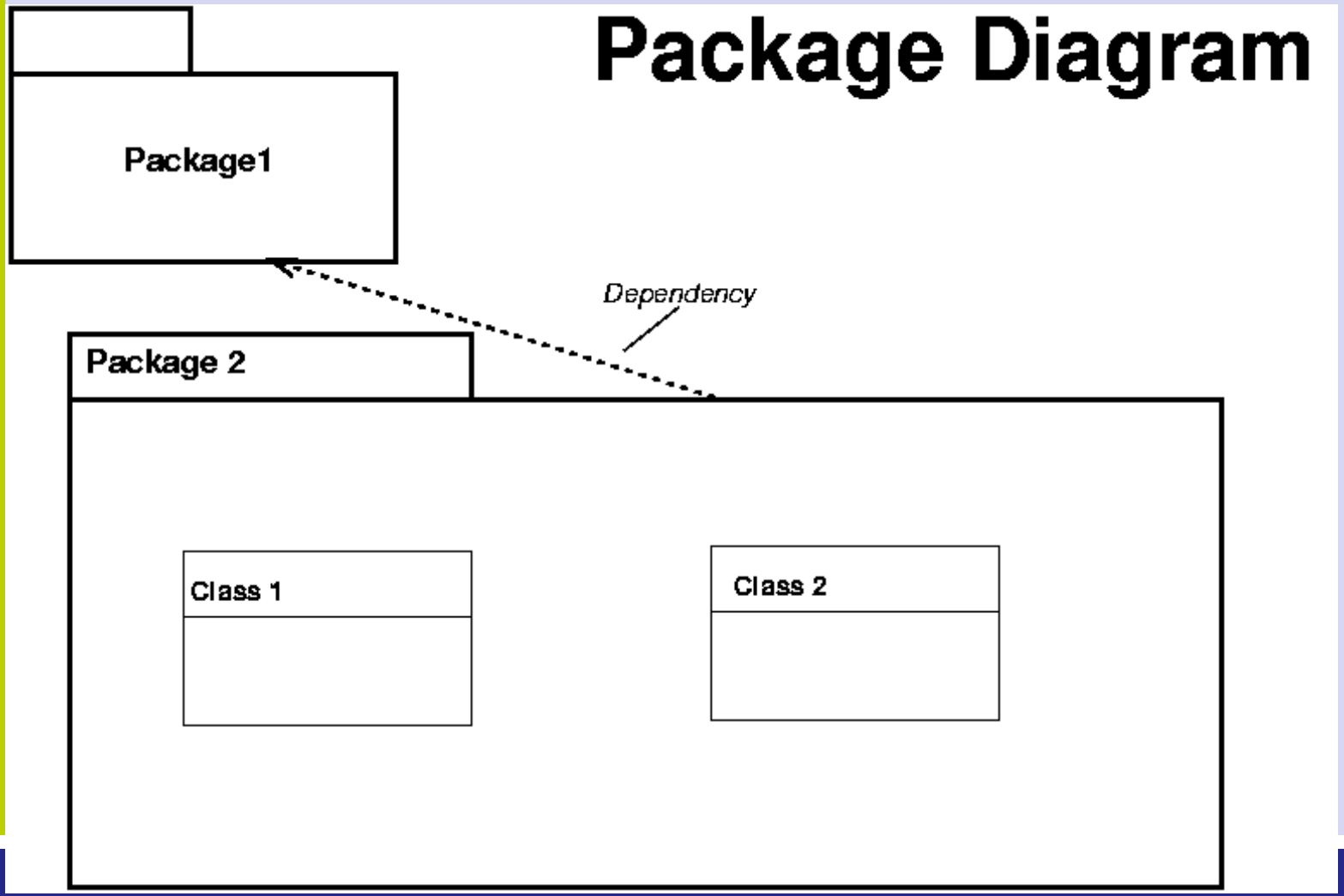
UML Object Diagram (Beispiel)



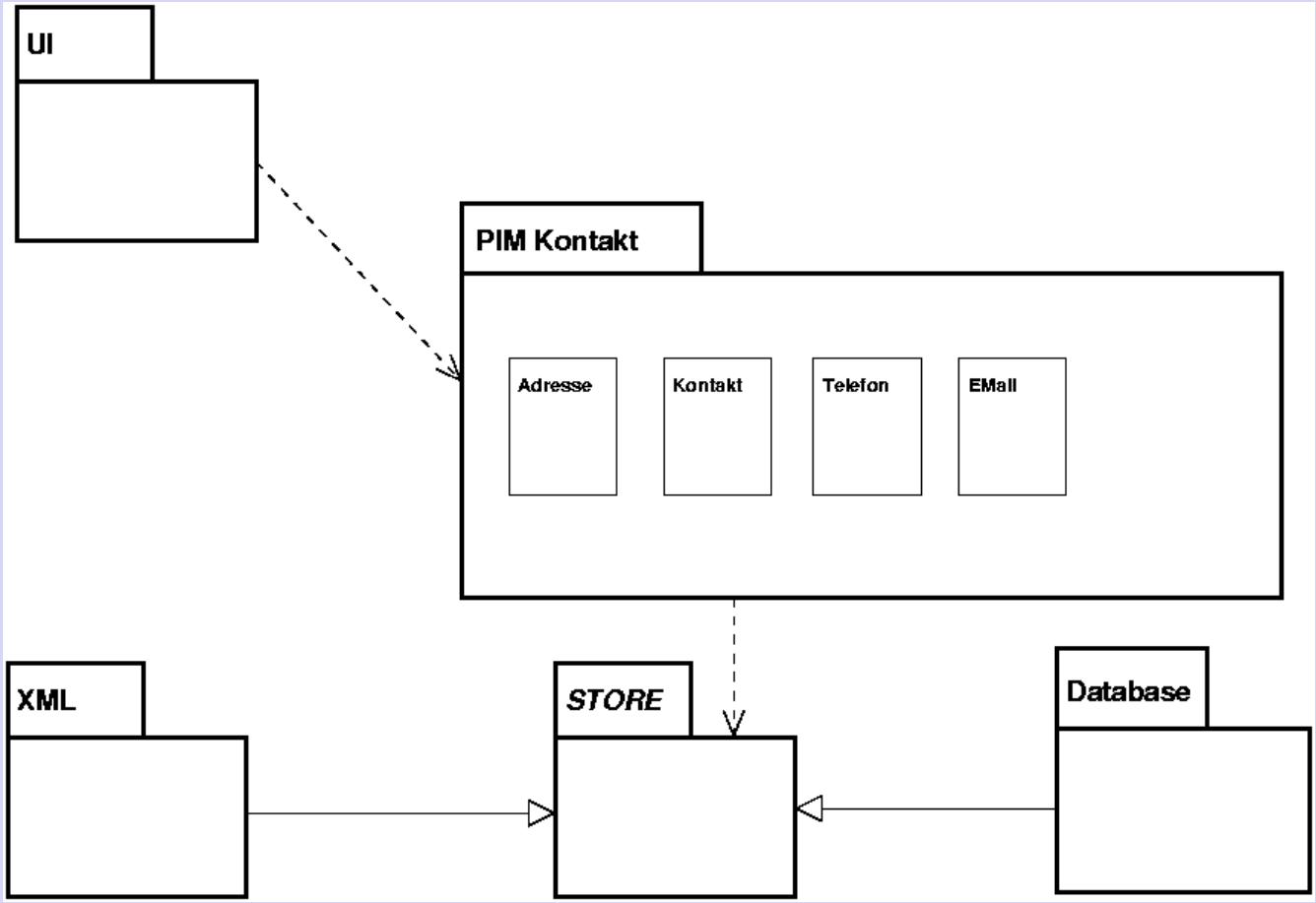
UML Package Diagram

- Gibt die Beziehungen von Klassengruppen an
- Veranschaulicht die Abhängigkeiten bei der Entwicklung
- Hilft bei der Planung von Schnittstellen
- Hilft verschiedene Projektteile zusammen zu entwerfen
- Läßt sich auf verschiedenen Hierarchiestufen einsetzen (auch verschachtelt)
- Abhängigkeiten werden in C++ vor allem durch die Verwendung von anderen Klassen durch “#include” ausgedrückt.
- Abhängigkeit ist nicht transitiv (im Sinne von Packages)
- Leider sind Includes transitiv

UML Package Diagram (Bestandteile)



UML Package Diagram (Beispiel)



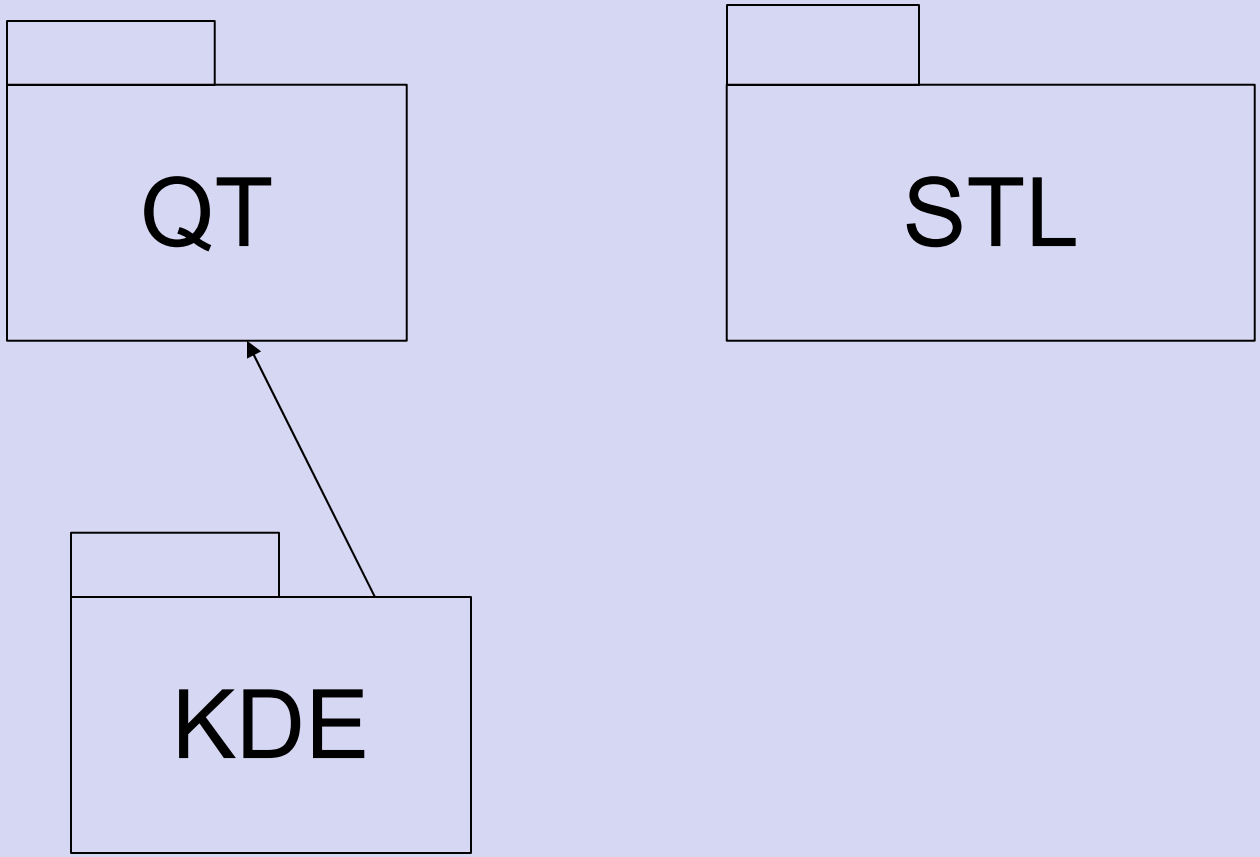
Packete (Probleme)

- Sollten mit minimalen Abhängigkeiten entworfen werden.
- Problem: Namensräume
- Problem: Basistypen
- Problem: Includes (C++)
- Problem: Library-Abhängigkeiten
- Problem: Fehlerbehandlung
- Problem: zu viele Schnittstellen (operationen, attribute)

Packete (Lösungen)

- Minimale Interfaces!
- Minimales Wissen über Interna des Paketes
- Anpassungsinterfaces (möglichst ohne Impedanzverluste)
- Auf der Ebene von Interfaces: Statefull
- Auf der Ebene von Objekten: Stateless
- Benutzen von Namespaces / Source erforderlich
- Excurs: Linkage von C++
- Pattern: Abstract Factory
- Pattern: Facade

Packete (Beispiele)



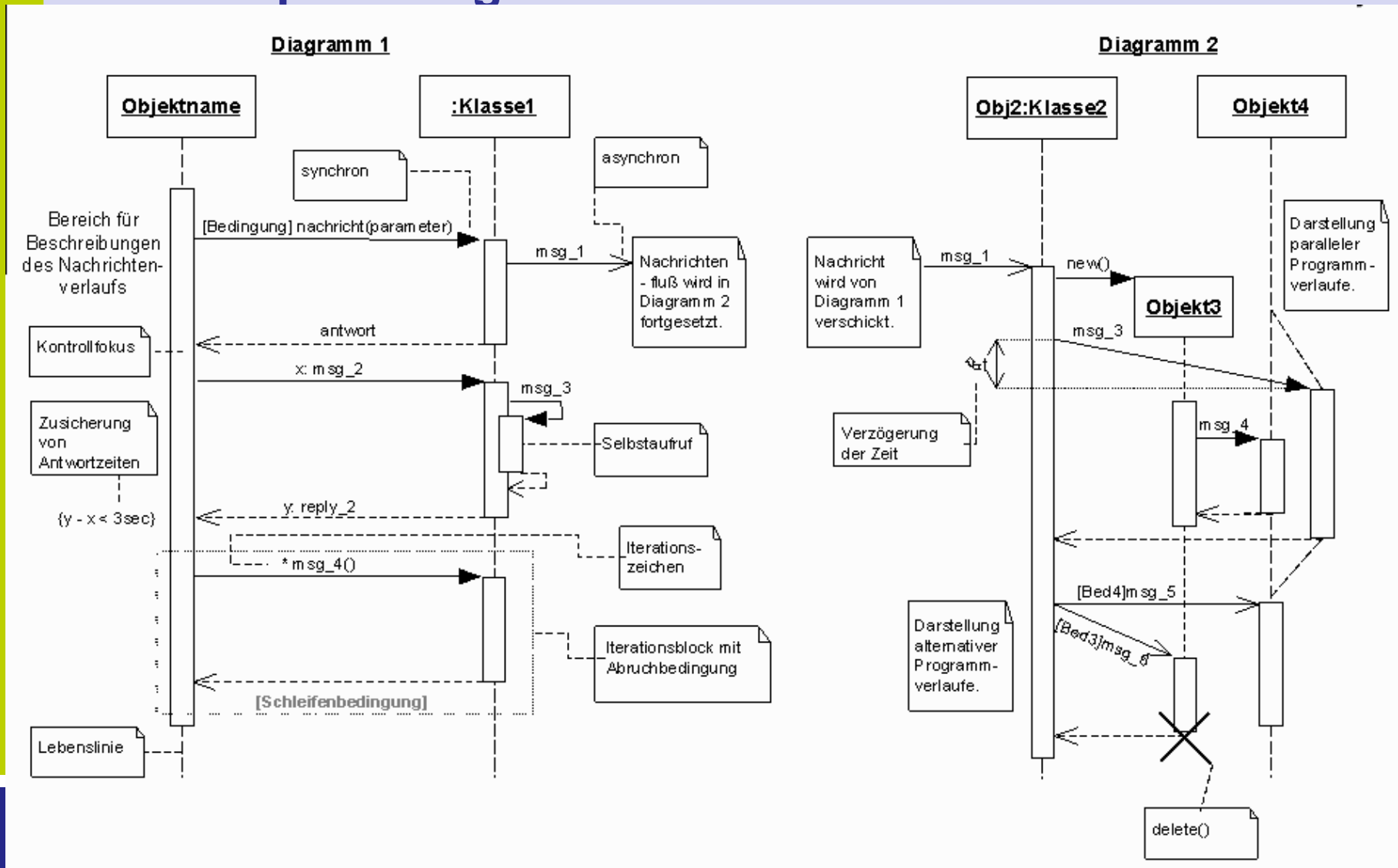
Objektorientierte Softwareentwicklung

Vorlesung 6
UML Diagramme, Diagramme, ..

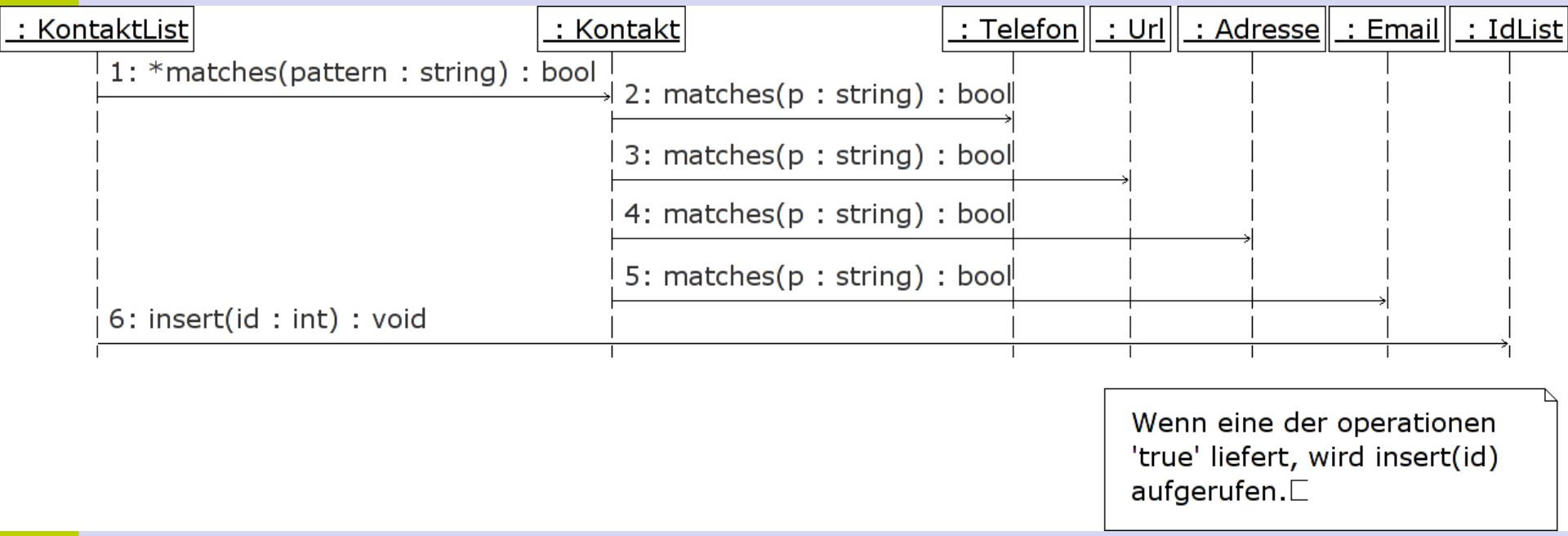
Das Sequenzdiagramm (Sequence Diagram)

- Ist ein “dynamisches Diagramm” d.h. es werden Abläufe während der Benutzung des Systems gezeigt.
- Elemente des Diagramm´s sind:
  - Klassen/Instanzen
  - Messages (manchmal Returns)
  - Creation Point/Destruction Point
- Wird nur bei wirklich komplizierten Erzeugungs- und Destruktionsszenarien benötigt
- Normalerweise wird kein Code generiert
- Dient auch zur Identifikation von Methoden, die gebraucht werden
- Möglichkeit Pakete zu finden durch Schwimmbahnen (Swimlanes)

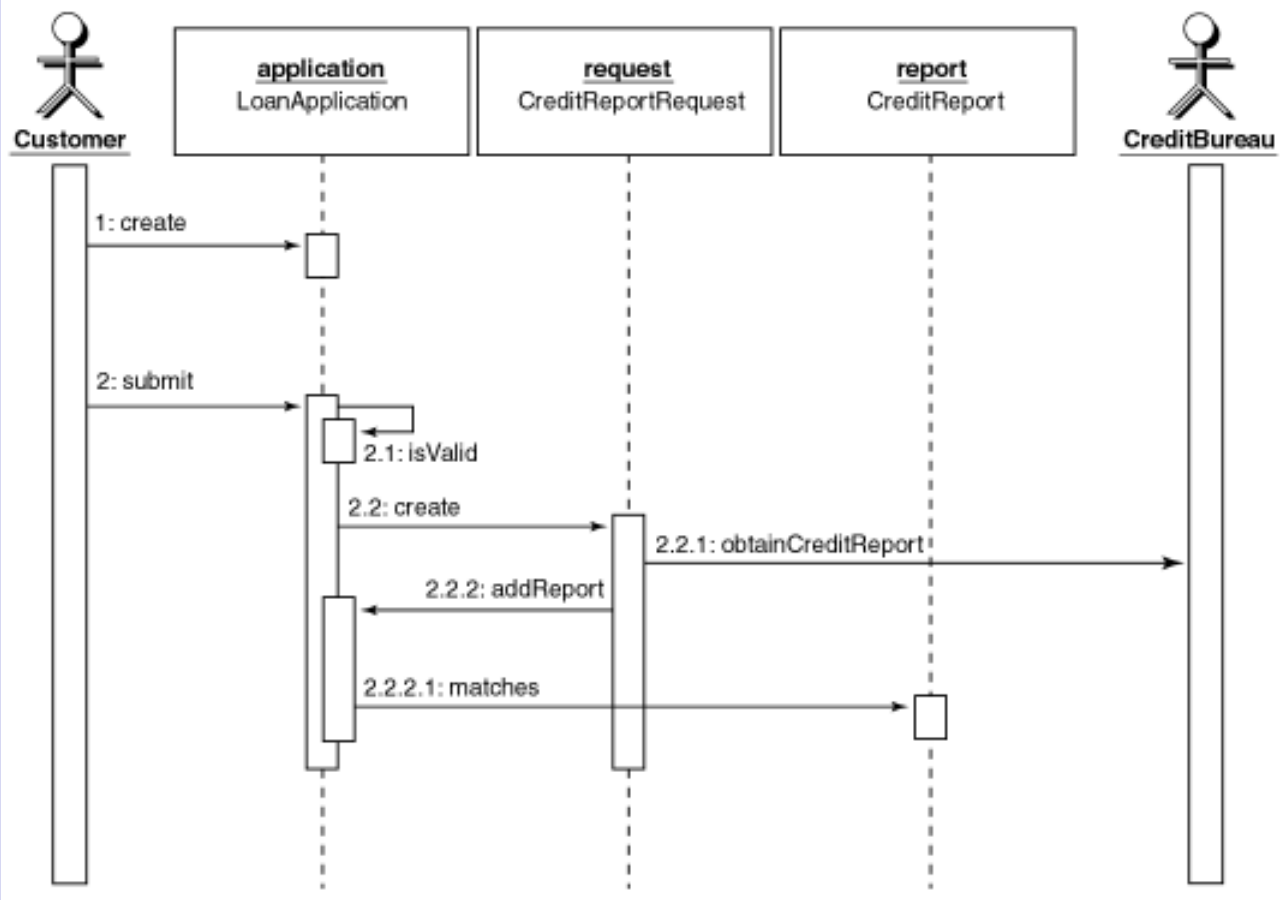
Das Sequenzdiagramm



Das Sequenzdiagramm (Beispiel1)



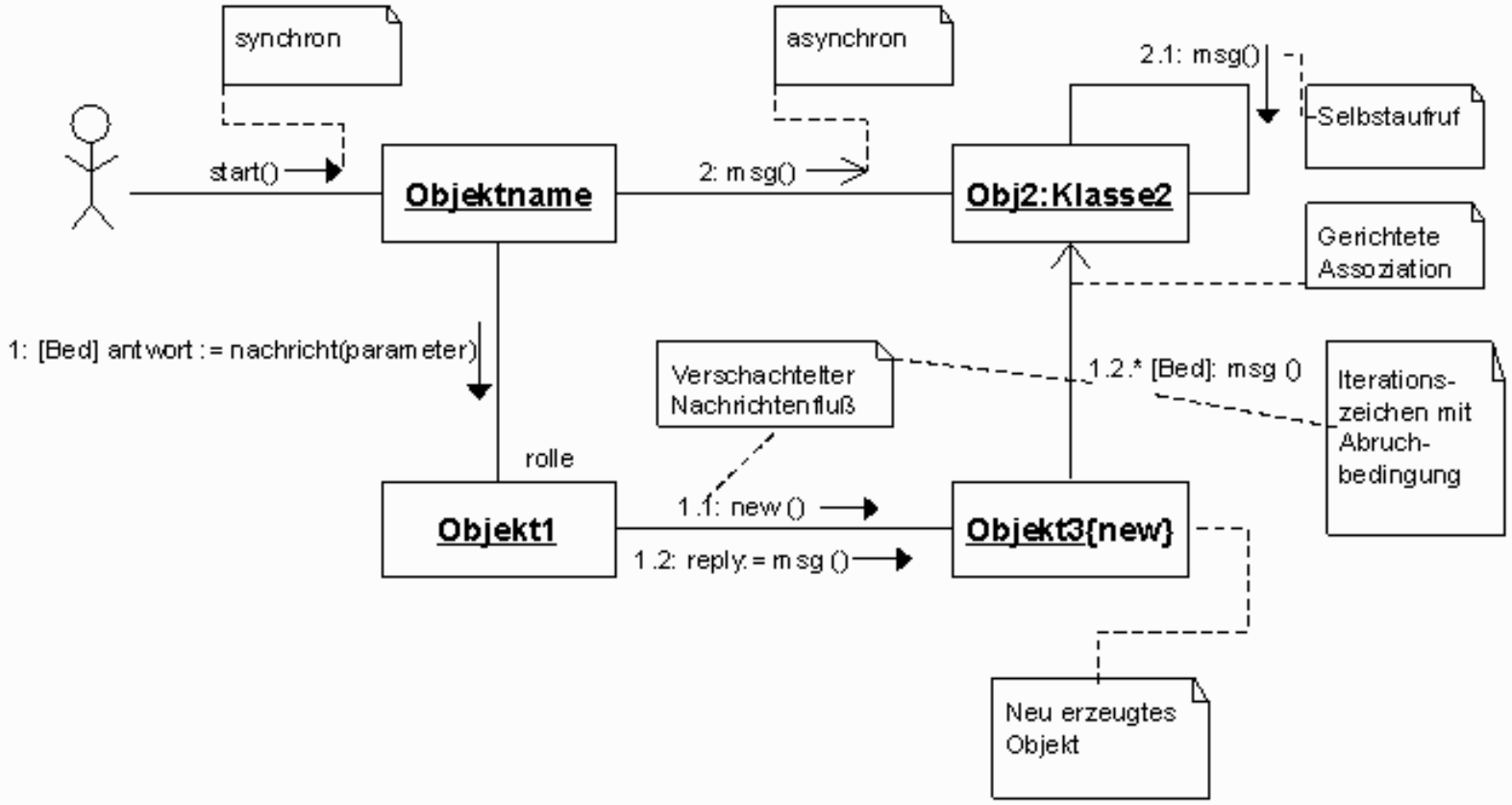
Das Sequenzdiagramm (Beispiel2)



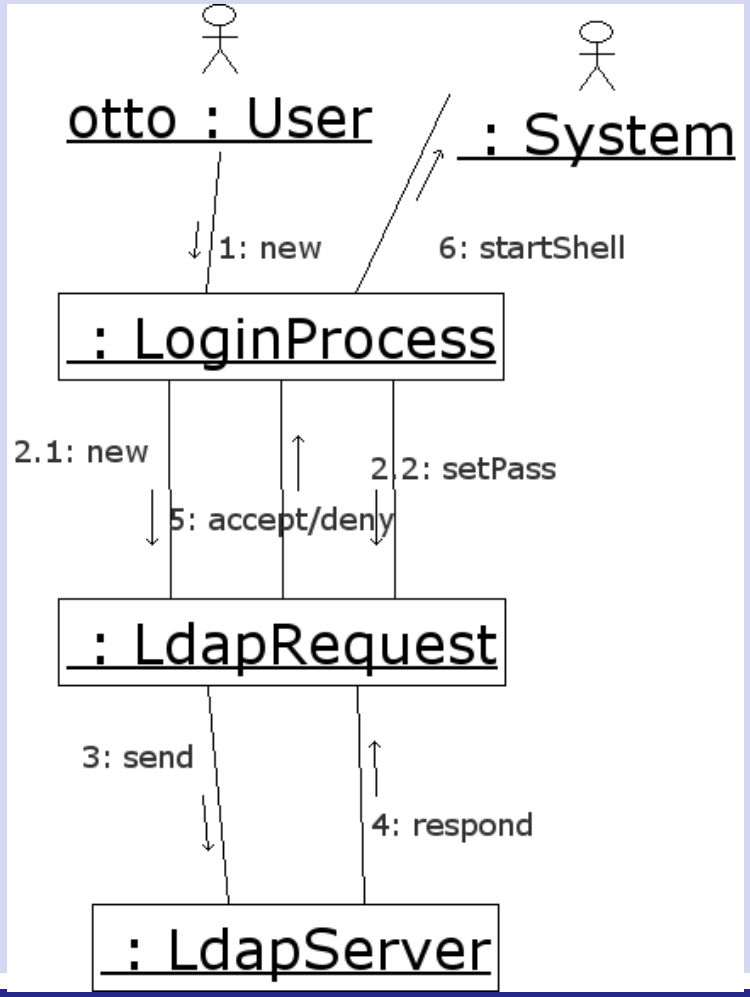
Das Collaborationsdiagramm

- Dynamisches Diagramm
- Zeigt Instanzen(Objekte) und ihre Kommunikation
- Noch abstrakter als das Sequenzdiagramm
- Keine Codegenerierung (nur Hinweise für Methoden)

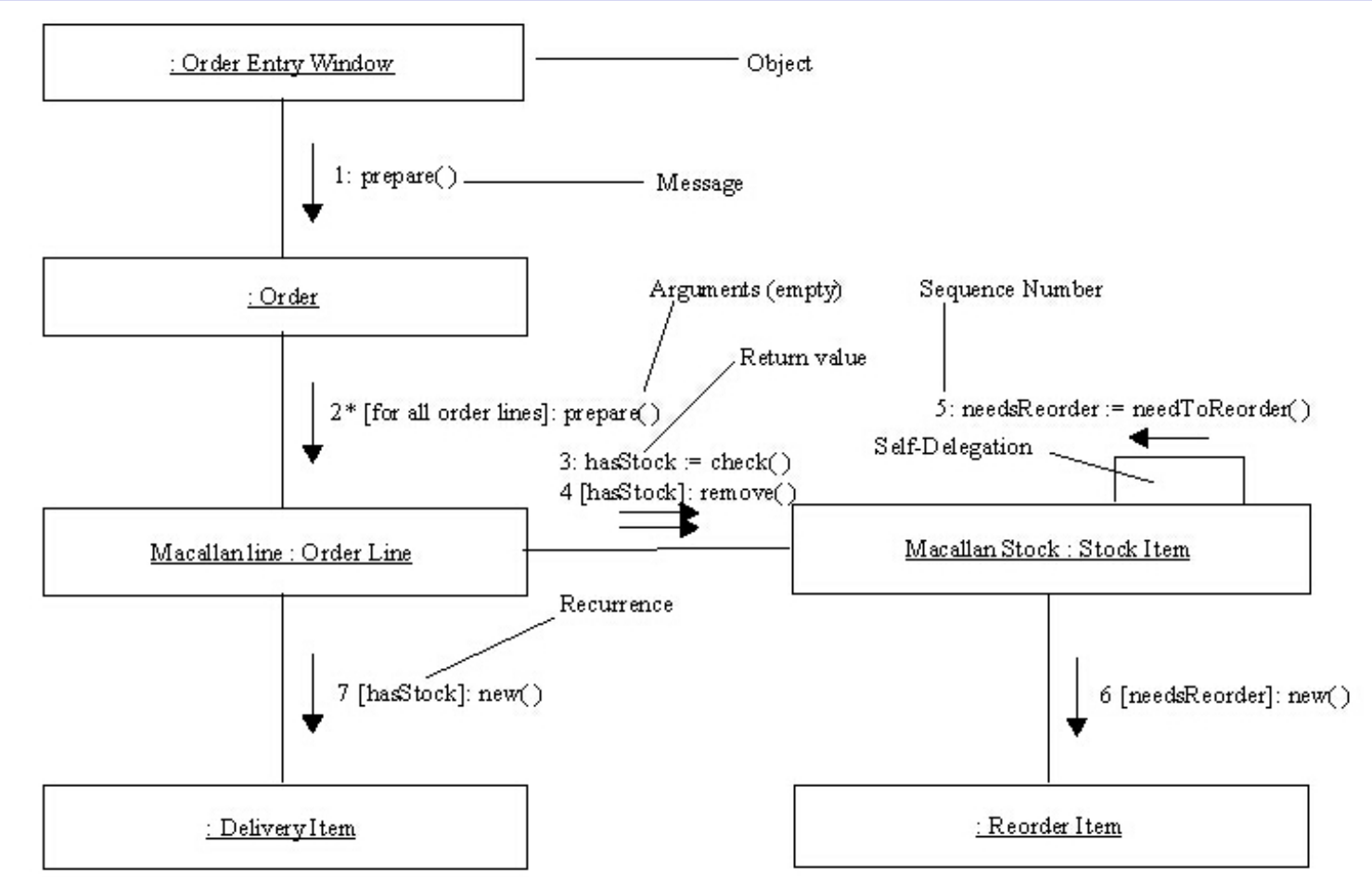
Das Collaborationsdiagramm



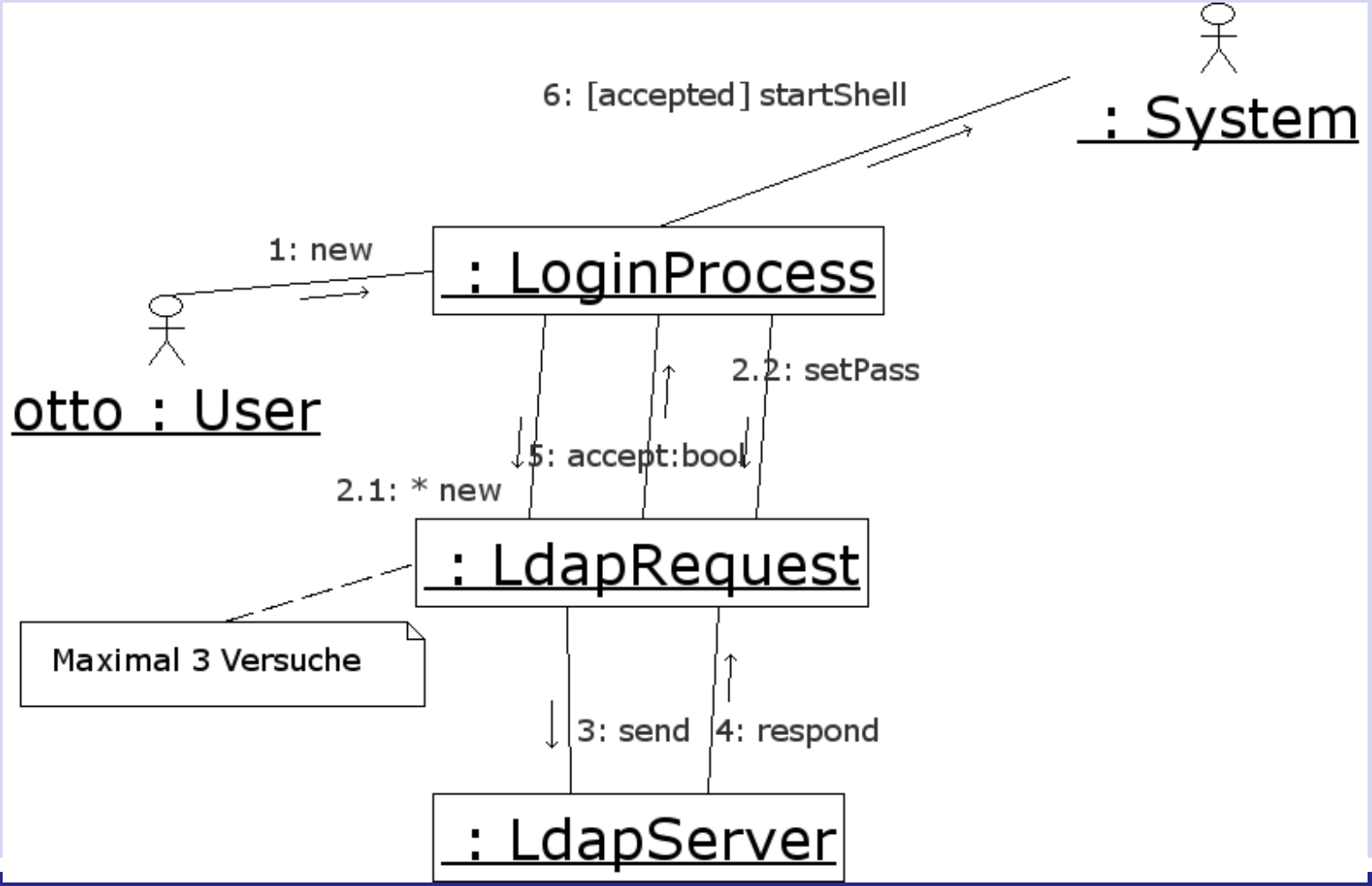
Das Collaborationsdiagramm (Bsp1)



Das Collaborationsdiagramm (Bsp2)

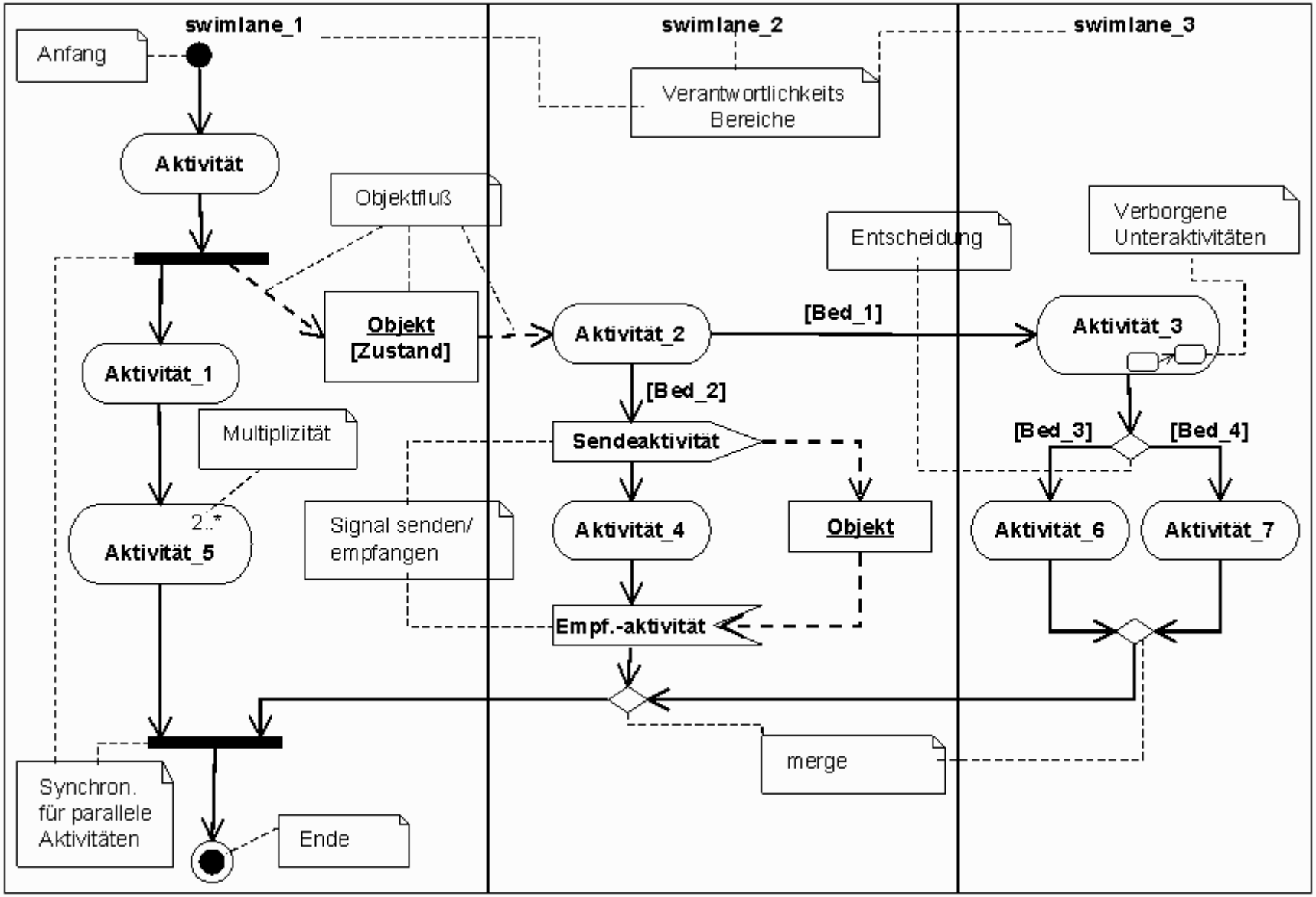


Das Collaborationsdiagramm (Bsp1a)



Das Aktivitätsdiagramm (Activity Diagram)

- Dynamisches Diagramm
- Zeigt den Ablauf einer Methode (meist)
- Entspricht dem guten alten PAP
- Enthält zusätzlich Nebenläufigkeiten
- Einschub: NO THREADS!!!!
- Kaum Möglichkeit zur Generierung von Code
- Dient zur Erklärung von Algorithmen


swimlane_1
swimlane_2
swimlane_3
Anfang
Aktivität
Objektfluß
Verantwortlichkeits Bereiche
Entscheidung
Verborgene Unteraktivitäten
Objekt [Zustand]
Aktivität_2
[Bed_1]
Aktivität_3
Aktivität_1
[Bed_2]
Multiplizität
Sendeaktivität
[Bed_3]
[Bed_4]
2..*
Aktivität_5
Signal senden/ empfangen
Aktivität_4
Objekt
Aktivität_6
Aktivität_7
Empf.-aktivität
merge
Synchron. für parallele Aktivitäten
Ende

Objektorientierte Softwareentwicklung

Vorlesung 7
Refactoring / XML

Schritte

- Ableitung aller relevanten Klassen von einem Interface XMLWritable mit einer Schreibmethode
- Implementierung dieser Funktion (und damit Realisierung des Interfaces in den Klassen)
- Dabei schreibt jede Klasse ihre Attribute als XML-Attribute (Namen!) und ihre Assoziationen als Childs.
- Natürlich ist die Verwendung einer XML-Library möglich, für diese einfache Aufgabe wäre aber wohl auch eine eigene Implementierung denkbar.
- Richtige Quotierung und Beachtung des Encodings

Ausschnitte aus den Vorgaben

- Definition von Adresse (adresse.h)
- Definition von Kontakt (kontakt.h)
- Definition von KontaktList (kontaktlist.h)

adresse.h

```
#include <string>
using namespace std;

class Adresse {
public:
  string name;
  string strasse;
  string ort;
  string plz;
  string land;
  Adresse(string _name="",string _strasse="",string
_ort="",string _plz="",string _land="")
:name(_name),strasse(_strasse),ort(_ort),plz(_plz),land(_la
nd) {}
  bool matches(string pattern) const;
};
```

kontakt.h

```
class Kontakt {
public:
  int id;
  string name;
  map <string,Adresse> adressen;
  map <string,Telefon> telefone;
  map <string,Url>     urls;
  map <string,EMail>   emails;
  map <string,Kontakt*>  verbindungen;
  virtual bool matches(string pattern) const;
  Kontakt(string _name="") { id=++idSerial;
name=_name; }
  virtual ~Kontakt() {}
  void addAdresse(string function,Adresse _adresse);
  void addTelefon(string function,Telefon _telefon);
  void addUrl(string function,Url _url);
  void addEMail(string function,EMail _email);
  void addVerbindung(string function,Kontakt
*_kontakt);
private:
  static int idSerial;
};
```

```
bool Kontakt::matches(string pattern) const {
  for(map<string,Adresse>::const_iterator
i=adressen.begin();i!=adressen.end();++i) {
    if((*i).second.matches(pattern)) return true;
  }
  for(map<string,Telefon>::const_iterator
i=telefone.begin();i!=telefone.end();++i) {
    if((*i).second.matches(pattern)) return true;
  }
  for(map<string,Url>::const_iterator
i=urls.begin();i!=urls.end();++i) {
    if((*i).second.matches(pattern)) return true;
  }
  for(map<string,EMail>::const_iterator
i=emails.begin();i!=emails.end();++i) {
    if((*i).second.matches(pattern)) return true;
  }
  return false;
}
```

kontaktlist.h

```
class KontaktList {
public:
  list <Kontakt*> kontakte;
  void addKontakt(Kontakt*);
  virtual ~KontaktList();
  virtual string& toXML(string &destination);
};
```

kontaktlist.c

```
void KontaktList::addKontakt(Kontakt *kontakt) {
  kontakte.push_front(kontakt);
}

KontaktList::~KontaktList() {
  for(list<Kontakt*>::const_iterator
i=kontakte.begin();i!=kontakte.end();++i) {
    delete(*i);
  }
  kontakte.clear();
}
```

Was man wissen sollte:

- Was ist XML/SAX
- Definition von XMLWritable
- Implementierung von XMLWritable

Interface XMLWritable

```
class XMLWritable {
public:
  virtual void write(SAX *sax)const =0;
};
```

```
class SAX {
public:
  virtual void startDocument()=0;
  virtual void endDocument()=0;
  virtual void appendAttribute(const string
&attributeName,const string &attributeValue)=0;
  virtual void appendAttribute(const string
&attributeName,int attributeValue)=0;
  virtual void startElement(const string &elementName)=0;
  virtual void endElement(const string &elementName)=0;
};
```

Realisierung von XMLWritable

```
class Adresse : public XMLWritable {
public:
  string name;
  string strasse;
  string ort;
  string plz;
  string land;
  Adresse....
  virtual ~Adresse() {}
  bool matches(string pattern) const;
  virtual void write(SAX *sax) const;
};
```

Realisierung von XMLWritable

```
void Adresse::write(SAX *sax) const {
  sax->startElement("Adresse");
  sax->appendAttribute("name",name);
  sax->appendAttribute("strasse",strasse);
  sax->appendAttribute("ort",ort);
  sax->appendAttribute("plz",plz);
  sax->appendAttribute("land",land);
  sax->endElement("Adresse");
}
```

Wie geht es noch besser?

- Noch abstrakter
- Definition von XMLWritable
- Implementierung von XMLWritable

Interface XMLWritable

```
class XMLWritable {
public:
  virtual void write(SAX *sax,const string
&function="") const;
  virtual void writeAttributes(SAX *sax) const=0;
  virtual void writeChildren(SAX *sax) const {};
  virtual string className() const=0;
};
```

Implementierung XMLWritable

```
void XMLWritable::write(SAX *sax,const string &function)
const {
  sax->startElement(className());
  if(!function.empty()) sax->
      appendAttribute("function",function);
  writeAttributes(sax);
  writeChildren(sax);
  sax->endElement(className());
}
```

Realisierung XMLWritable (simple)

```
class EMail : public XMLWritable {
public:
  string adresse;
  EMail(string _adresse = "" ) :adresse(_adresse) {}
  virtual ~EMail() {}
  bool matches(string pattern) const;
  virtual void writeAttributes(SAX *sax) const;
  virtual string className() const { return "EMail"; }
};



void EMail::writeAttributes(SAX *sax) const {
  sax->appendAttribute("adresse",adresse);
}
```

Realisierung XMLWritable (advanced)

```
void Kontakt::writeAttributes(SAX *sax) const {
  sax->appendAttribute("id",id);
  sax->appendAttribute("name",name);
}

void Kontakt::writeChildren(SAX *sax) const {
  for(map<string,Adresse>::const_iterator
i=adressen.begin();i!=adressen.end();++i) {
    i->second.write(sax,i->first);
  }
  for(map<string,Telefon>::const_iterator
i=telefone.begin();i!=telefone.end();++i) {
    (*i).second.write(sax,i->first);
  }
..........
}
```

Hausaufgabe

- Ergänzen Sie unser PIM-Beispiel mit einer kompletten XML-Ausgabe. (siehe Website)

Objektorientierte Softwareentwicklung

Vorlesung 8
Wo sind eigentlich die Objekte?
Speicher und seine Verwaltung

Das Speicher-Drama

- Die CPU das Betriebssystem und die Programmiersprache sind Bestandteile der Speicherverwaltung (Allokator)
- Die Evolution des Speichers – Virtualisierung überall
- Modelle dienen der Vereinfachung und Abstraktion – und schaffen Probleme
- Wie kann objektorentierte Programmierung zur Lösung des Speicherproblems eingesetzt werden?
- Welche speziellen Probleme werden durch Objektorientierung erzeugt?

Wie Speicher arbeitet

1. Generation: Das Programm adressiert den Speicher direkt
2. Generation: Das Programm adressiert Speicher, der mittels Mappingtabelle auf den physikalischen Speicher verweist. (MMUs)
3. Generation: Das Programm adressiert Speicheradressen, die auf Pageadressen umgerechnet werden, die wiederum auf physikalischen Speicher gemappt werden. (Descriptor Tables/ Virtuelle Speicherverwaltung)
4. Generation: Das Programm adressiert Speicher, der auf virtuelle Speicherbereiche verweist, der auf Adressen umgerechnet wird, der auf Pages verweist.....
5. .....

Wie Programme Speicher brauchen

- 1. Generation von Allokatoren: Fest zugewiesener Speicher – alle Objekte (damals nur Variablen genannt) haben feste Adressen und sind beim Programmieren oder Compilieren bekannt.
- 2. Generation von Allokatoren: Parameter und lokale Variablen werden auf dem Stack angelegt, dadurch wird rekursiver Aufruf von Funktionen möglich. (Pascal)
- 3. Generation von Allokatoren: Es wird ein Heap zur Verfügung gestellt von dem „nach oben“ Speicher dynamisch entnommen werden und mit „mark“ und „release“ wieder freigegeben werden kann

  Einschub: 3a „alloca“ Heap Allokator im Stack

Wie Programme Speicher brauchen (2)

- 4. Generation von Allokatoren: Speicher kann dynamisch beschafft und freigegeben werden. („malloc“, „free“) der Speicher ist dabei unorganisiert („void“). Der Allokator ist dabei meistens eine Emulation eines Random-Allokators auf dem Heap
- 5. Generation von Allokatoren: Speicher wird „typisiert“ beschafft und freigegeben („new“, “delete“). Der Allokator kennt die Art des zu beschaffenden Objekts. Zusätzlich werden Initialisierer und Destruktorfunktionen aufgerufen.
- 6. Generation von Allokatoren: Der Speicher wird zwar explizt beschafft, aber nur implizit freigegeben. Die Freigabe erfolgt durch das Allokationssystem selbst. (Garbage Collektor)

Hohe Flexibilität vs.armes Modell

- Aus der ersten Generation der Computer stammt das heute noch gültige Modell der Speichersegmente: Code (text), Daten (data,bss), Stack, Heap(bss)
- Zusätzlich sind Prozesse gegeneinander geschützt sowie die Prozesse gegen das Betriebssystem abgeschirmt
- Als zusätzliche Modelle stehen „shared memory“ und „mmap“ zur Verfügung.
- Leider sind alle Modelle nicht auf die schnelle Erzeugung und Zerstörung kleiner Speicherobjekte zugeschnitten.
- Die Abbildung von „new“ und „delete“ auf Heap-Allokation und Free-list führt zu Fragmentierung und exzessivem Speicherwachstum

Einordnung von Objekten in Speicherklassen

Objektspeicher können nach mehreren Kriterien in Gruppen eingeteilt werden:

- Lebensdauer („single shot“ wie z.B. IteratorResults bis zu „persistent)
- Größe
- Anzahl der Erzeugungen/Zerstörungen
- Dynamische oder feste Größe
- Unifizierbarkeit
- Sichtbarkeit innerhalb des Programms, Threads der Unit oder des Systems

C++ einfache Stack Allokation

```
class X {
public:
  int i;
};

int main() {
  X x;
  x.i=3;
}
```

C++ (2) Stack Allokation mit Konstruktor

```
class X {
public:
  X() { i=0; }
  int i;
};

int main() {
  X x;
  x.i=3;
}
```

C++ (3) Konstruktor/Parameter

```
class X {
public:
  X(int _i) :i(_i) {}
  int i;
};

int main(int argc,char *argv[]) {
  X x(1);
}
```

C++ (4) Heap Allokation

```
class X {
public:
  X(int _i=0) :i(_i) {}
  int i;
};

int main(int argc,char *argv[]) {
  X *x=new X;
  x->i=7;
  delete x;
}
```

C++ (5) Array Allokation

```
class X {
public:
  X(int _i=0) :i(_i) {}
  int i;
};

int main(int argc,char *argv[]) {
  X *x=new X[10];
  x[3].i=7;
  delete[] x;
}
```

Objekt

Ein Objekt ist die abstrahierte, maschinelle Abbildung eines Teils der modellierten Umwelt oder des Softwaresystems. Es kann Exemplar einer oder mehrerer Klassen sein. Ein Objekt setzt sich zusammen aus:

- Objektidentität
- optionalen Operationen
- optionalen Attributen
- optionalen Assoziationen zu anderen Objekten

Klasse

Eine Klasse ist ein Konzept zur Beschreibung gemeinsamer Eigenschaften (Properties) von Objekten (Abstraktion). Die Klasse ermöglicht die Abgrenzung eines Konzepts. Klassen können Spezialisierungen von anderen Klassen (Vererbung) sein. Das Verhalten der Objekte einer Klasse wird durch Operationen, ihre Struktur durch Attribute und Assoziationen bestimmt. Eine Klasse enthält explizit oder implizit die Vorschrift zur Konstruktion (Konstruktor) und Destruktion (Destruktor) von Objekten. Sie kann zusätzlich durch Stereotypen und Eigenschaftslisten beschrieben werden.

Operation

Die Operation beschreibt das Vermögen eines Objekts auf Botschaften (Messages) zu reagieren. Botschaften besitzen Signaturen, die mindestens aus Name und Parametertypenliste besteht. Die Implementierung einer Operation nennt man Methode. Sie beschreibt das Verhalten des Objekts als Reaktion auf die erhaltene Botschaft. Die Eigenschaften einer Operation sind:
Sichtbarkeit, Name, Parameterliste, Returntyp und Attributliste

Attribut

Die Attribute beschreiben die Struktur des Objekts bzw. aller Objekte einer Klasse. Es setzt sich aus Namen, Typ, Sichtbarkeit und optional Multiplizität, Stereotyp und Attributliste zusammen. Attribute können nur lesbar (readonly) oder auch abgeleitet (generisch) sein. Nach Außen sind Attribute durch die Möglichkeit repräsentiert einen objektspezifischen Wert lesen (Get-Operation) oder schreiben (Set-Operation) zu können. Deshalb kann man Attribute auch als spezielle Operationssets verstehen. Die technische Ausprägung eines Attributs ist oft eine Membervariable.

Klassenattribut Klassenoperation

Klassen sind auch als (Singelton) Metaobjekte definierbar. Dabei existiert explizit oder implizit ein unifiziertes Objekt, das die Klasse repräsentiert. Dieses Metaobjekt verfügt dann auch über Operationen und Attribute, die für alle Exemplare der Klasse gelten. Zum Aufruf einer Klassenoperation oder zum Zugriff auf ein Klassenattribut ist kein Exemplar der Klasse erforderlich. Spezielle Klassenoperationen sind Konstruktoren. Klassenoperationen und Klassenattribute werden oft für Bookkeeping verwendet.

Generalisierung

stellt eine gerichtete Beziehung zwischen verschiedenen Konzepten dar. Sie drückt eine n:1 spezieller: allgemeiner Beziehung aus und stellt eine Form der Abstraktion dar. Generalisierung entspricht oft einer Verallgemeinerung. Technisch wird die sie durch die Einführung von Superklassen, abstrakten Basisklassen oder Interfaces realisiert.

Spezialisierung

ist eine gerichtete Beziehung zwischen einem oder mehreren generelleren und einem spezielleren Konzept. (1..*:1) Normalerweise kann ein spezielleres Konzept ohne Bedenken als allgemeineres verwendet werden aber nicht umgekehrt. Bei Klassen sprechen wir von der allgemeineren (generelleren) Superklasse und der spezielleren Subklasse. Ein Konzept kann mehrere Konzepte spezialisieren (z.B. Mehrfachvererbung), aber nur ein Konzept verallgemeinern.

C++ (6) Destruktor

```
class X {
public:
   X(int _i=0) :i(_i) { printf("neues X\n"); }
   ~X() { printf("ein X freigegeben\n"); }
   int i;
};

static X y;
int main(int argc,char *argv[]) {
X x;
   x=y;
   return 0;
}
```

C++ (7) Copy Construktor (implizit)

```
class X {
public:
   X(int _i=0) :i(_i) { printf("neues X\n"); }
   ~X() { printf("ein X freigegeben\n"); }
   int i;
};

static X y;
int main(int argc,char *argv[]) {
X x(y);
   return 0;
}
```

C++ (7) Copy Construktor (explizit)

```
class X {
public:
   X(int _i=0) :i(_i) {}
   X(const X&_x) :i(_x.i) {} // Copy Construktor
   int getI() const { return i; }
privat:
   int i;
};

static X y;
int main(int argc,char *argv[]) {
X x(y);
   return 0;
}
```

C++ (8) Flat Copy

```
class X {
public:
   X(int _i=0) :i(_i) {}
   int i() const { return i; } // andere Variante
privat:
   int i;
};

static X y;
int main(int argc,char *argv[]) {
X x;
   x=y;
   return 0;
}
```

C++ (9) operator=

```
class X {
public:
   X(int _i=0) :i(_i) {}
   operator=(const _x&) { i=_x.i; }
   int i;
};

static X y;
int main(int argc,char *argv[]) {
X x;
   x=y;
   return 0;
}
```

C++ (10) operator new

```
class X {
public:
   int i;
  void *operator new(size_t size) {
    if(fcount) return felems[--fcount];
    return ::operator new(size);
  }
  void operator delete(void *ptr,size_t size) {
    if(fcount<(sizeof(felems)/sizeof(felems[0])) {
      felems[fcount++]=ptr;
    } else {
      ::operator delete(ptr,size);
    }
  }
private:
static void* felems[10];
static int fcount;
};
```

Andere Aspekte der Speicherverwaltung

- Strings als Hauptproblem und verschiedene Lösungen
- Refcountdesign / Unifikation
- Deep vs. Flat Copy
- Garbage Collection
- “Mapped Files” and “Shared Memory”
- “by value” oder “by reference”
- Memory Leakage und “Checker”

Hausaufgabe (8)

- Ergänzen Sie in unserem PIM die Klassen für Termine und Aufgaben.
- Ergänzen Sie das Testprogramm entsprechend
- Verwenden Sie als Basis das Programmpaket A8 von der Website und das Klassendiagramm aus der Vorlesung.

Objektorientierte Softwareentwicklung

Vorlesung 9

C++ und Konkrete Klassen

Konkrete Klassen

- Implementieren einfache Typen
- Sollen sich möglichst nahtlos in das bestehende Typsystem einbetten
- Können wie eingebaute Typen verwendet werden
- Tragen extrem zur Lesbarkeit von Programmen bei
- Verschleiern die Anzahl und Komplexität von Operationen und sollten deshalb mit Vorsicht implementiert und benutzt werden
- Dienen der Implementierung eines “kleinen” Konzepts
- Von konkreten Klassen soll gewöhnlich nicht abgeleitet werden
- Beispiele: STL string, complex

Konkrete Klassen (2)

- Es war ein wesentliches Designziel bei der Entwicklung von C++, die Erstellung konkreter Klassen zu ermöglichen.
- Dieses Konzept wird von anderen OO-Sprachen oft nicht unterstützt. (Java)
- Anmerkung: konkrete Klassen sind keineswegs das Gegenteil abstrakter Klassen!
- Beispiel: Decimal

Beispiel Decimal

- Eine Klasse soll das Rechen- und Rundungsverhalten einer Dezimalzahl implementieren.
- Diese Klasse soll sich ohne Probleme zusammen mit den anderen Datentypen verwenden lassen

Die Klasse sollte folgende Operationen unterstützen:

```
Decimal a,b(2.2);
Decimal c=4;
a=b;
a=a+b;
a=a+2.43;
a=2.43+a+sin(a);
```

constructor und copy construktor

```
class Decimal {
  static const int maxDigits=32;
  int exponent;
  bool sign;
  int digitCount;
  char digits[maxDigits];
public:
  Decimal()
  :exponent(0),sign(false),digitCount(0)
  {}
  Decimal(const Decimal &);
};
```

copy construktor (implementation)

```
Decimal::Decimal(const Decimal &src)
:exponent(src.exponent),sign(src.sign),
 digitCout(src.digitCount)
{
  for(int i=0;i<digitCount;i++) {
    digits[i]=src.digits[i];
  }
}
```

Einschub: Nochmal Values und References

- Beim Aufruf von Funktionen/Methoden und bei der Rückgabe von Werten kann man zwischen „By value“ und „By Referenz“ unterscheiden
- Bei der Aufbewahrung von Objekten kann sind diese Unterscheidungen ebenso möglich
- Speziell bei der Rückgabe von Referenzen ist höchste Vorsicht geboten: Existiert das Objekt nach der Rückgabe überhaupt noch?
- Bei konkreten Klassen wird vorzugsweise by Value gearbeitet, Ausnahme ist die konstante Referenz, die eine Kopie vermeidet.

operator=

```
class Decimal {
...
public:
  Decimal &operator=(const Decimal& src);
};

Decimal &Decimal::operator=(const Decimal &src){
  if(&src==this) return;
  exponent=src.exponent;
  sign=src.sign;
  digitcount=src.digitCount;
  for(int i=0;i<digitCount;i++)
    digits[i]=src.digits[i];
  return *this;
}
```

operator+

```
class Decimal {
...
public:
  Decimal operator+(const Decimal& src) const;
};

Decimal Decimal::operator+(const Decimal &src)
   const {
  Decimal result;
  //Viele interessante Dinge
  return result;
}
```

andere operatoren

```
class Decimal {
...
public:
 Decimal operator+(const Decimal& src) const;
 Decimal operator-(const Decimal& src) const;
 Decimal operator%(const Decimal& src) const;
 Decimal operator/(const Decimal& src) const;
 Decimal operator*(const Decimal& src) const;
 Decimal operator-() const;
  ....
};
```

globale operatoren

```
// Problem : Auswahl der Operatoren nach
// linkem Argument deshalb:
// fuer 1+Decimal(2)
Decimal operator+(int,const Decimal&rhs){
  return Decimal(1)+rhs;
}
```

Umwandlungsfunktionen

```
//Automatisches Konvertieren in andere
//Typen
..
operator double() const;
...

//Verwendung automatisch
Decimal a(2.2);
printf(„%e\n“,sin(a));
```

Objektorientierte Softwareentwicklung

Vorlesung 10

Muster in der OO-Entwicklung - Pattern

Pattern

- Beobachtung: Routine führt zu immer wieder verwendeten Mustern
- Metabetrachtung: Welche dieser Muster sind gut. (abstrakt, decken das Problem optimal ab)
- Sammlung solcher Pattern in Templates oder Büchern

Muster auf verschiedenen Ebenen

- “Kontrollstrukturen” (wie while, if, switch, for, ...) sind ein Muster zur übersichtlichen Beschreibung von Aktivitäten
- „Semaphor“ ist ein Muster zur sicheren Resourcenverwaltung
- „Entitymodell“ ist ein Muster zur Beschreibung von persistenten Datenspeichern
- “Objektorientierung” ist ein Muster zur Reduktion der Komplexität in Programmen
- „Gesetz“ ist ein Muster zur Regelung gesellschaftlicher Probleme.

Eigenschaften von Mustern

1. Muster haben eine Aufgabe (Wofür, Problem)
2. Muster besitzen eine Namen (Was)
3. Muster haben ein Anwendungsgebiet (Wo)
4. Muster haben eine bestimmte Struktur (der Trick, Lösung)
   a) Teilnehmer
   b) Zusammenwirken
5. Muster haben Ergebnisse
   a) positive – zusätzlich zur Lösung des Problems
   b) negative

Ein Beispielmuster – “Semaphore”

1. Aufgabe: Eine Resource soll so verwaltet werden, dass nur jeweils genau eine Verwendung möglich ist.
2. Name: „Semaphore“
3. Anwendungsgebiet: Eine Resource ohne Verwaltung wird von einer exclusiven Resource verwaltet.
4. Struktur Teilnehmer: nicht exclusive Resource, exclusive Resource

   Struktur Zusammenwirken: jeweils genau eine exclusive Resource wird einer zu verwaltenden Resource zugeteilt. Bevor die verwaltete Resource verwendet wird, muss die exclusive an einem allen bekannten Platz beschafft werden. Nach der Verwendung wird die exclusive Resource zurückgelegt.
5. Ergebnisse: Zusätzlicher Aufwand für die Verwaltung einer anderen Resource (u.U. Problemtransformation) , Wartezeiten müssen geschickt überbrückt werden. Verlust der exclusiven Resource blockiert das System

Ein Beispielmuster – “Singelton”

1. Aufgabe: Von einer Klasse soll genau ein Objekt existieren und global zur Verfügung stehen
2. Name: „Singelton“
3. Anwendungsgebiet: OO-Programmierung bei der einzelne Klassen nur „einmalig“ instanziiert werden dürfen
4. Struktur Teilnehmer: Singlelton-Klasse, User-Klasse

   Struktur Zusammenwirken: Objekte des Singelton-Typs können nicht von anderen Objekten erzeugt werden (privater Konstruktor). Statt dessen gibt es eine Funktion mit Klassen-Scope, die eine Referenz auf ein Klassenelement zurückgibt.
6. Ergebnisse: Die Semantik der Konstruktion wird verändert, globale Variablen werden vermieden, erlaubt Erweiterung zu mehreren Instanzen und verschieden Implementierungen

„Singelton“ (Deklaration)

```
seriffont.h
class SerifFont :public Font {
private:
  SerifFont():Font(“Courier”,12) {}
public:
  static SerifFont *instance() {
    if(!theInstance) theInstance=new SerifFont;
    return theInstance;
  }
private:
  static SerifFont *theInstance;
};
```

„Singelton“ (Implementation)

seriffont.c:

```
#include <seriffont.h>
SerifFont *SerifFont::theInstance;
```

„Singelton“ (Usage)

```
#include <seriffont.h>
...
Font *f1=serifFont::instance();
DrawText(handle,f1,“Hallo“);
....
```

„Singelton“ (Template)

```
template <class T> class singelton :public T {
private:
  singleton<T>():T() {}
public:
  static singleton<T>*instance() {
    if(!theInstance) theInstance=new
   singleton<T>;
    return theInstance;
  }
private:
  static singleton<T>*theInstance;
};
```

„Singelton“ (Template/Usage)

```
...
  class T1 {
  public:
    int i;
    T1() :i(1) {}
  };
...
// Implementation of static Member
singleton<T1>* singleton<T1>::theInstance;
...
// Real Usage
printf(“%d\n”,singleton<T1>::instance->i);
```

Ein Beispielmuster – “Wrapper”

1. Aufgabe: Ein nicht-objektorientiertes System soll mit einem OO-System gekoppelt werden
2. Name: „Wrapper“
3. Anwendungsgebiet: Verwendung von C-Libraries in OO-Systemen, Ankopplung von anderen Systemen oder Sprachen
4. Struktur Teilnehmer: Klasse als Wrapper, externe Library

   Struktur Zusammenwirken: Die „Klasse“ in der Library wird auf eine Klasse in der OO-Sprache abgebildet (1:1) dabei kapselt der Wrapper die Funktionen und bildet einen Namensraum für die Verwendung
5. Ergebnisse: Kontrolle über die Verwendung der Library, eventuelle Ersetzbarkeit, Impedanz-Mismatch

Hausaufgabe (10)

- Nennen Sie je ein Pattern aus dem Bereich der Mathematik, der Informatik und der Mechanik und begründen Sie Ihre Entscheidung
- Welche Pattern aus dem Buch „Design Pattern“ sind vom generelleren Pattern „Publisher/Subscriber“ spezialisiert?

Creational Patterns

Abstract Factory: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder : Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method : Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype : Specify the kinds of objects to create using a prototypical instance, and (create new objects by copying this prototype.

Singleton : Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Adapter : Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge : Decouple an abstraction from its implementation so that the two can vary independently.

Composite : Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structural Patterns (2)

Decorator : Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade : Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight : Use sharing to support large numbers of fine-grained objects efficiently.

Proxy : Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Chain of Responsibility : Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command : Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter : Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator : Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator : Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Behavioral Patterns (2)

Memento : Without violating encapsulation, capture and externalize an object's intemal state so that the object can be restored to this state later.

Observer : Define a one-to-many dependency between objects so that when one object changes state, al1 its dependents are notified and updated automatical1y.

State : Al1ow an object to alter its behavior when its intemal state changes. The object wil1 appear to change its class.

Strategy : Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Behavioral Patterns (3)

Template Method : Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor : Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Objektorientierte Softwareentwicklung

Vorlesung 11
Generizität, Metaprogramming und Scripting

Generizität

- Allgemein: Anzahl der Programmzeilen ist ein wichtiges Maß für die Qualität! (und zwar je weniger desto besser)
- Kann auf viele verschiedene Arten erreicht werden z.B:
  - Templates (in der Sprache)
  - Generatoren (im Werkzeug)
  - Design (im Entwurf)

Software Reuse

- simple Library (API)
- simple Copy and Edit
- Generatoren erster Ordnung (Wizzards)
- Generatoren zweiter Ordnung (Forth and Back)
- Generatoren dritter Ordnung (Round Trip)
- Generative Elemente in der Programmierung
- Generative Elemente in der Programmiersprache
- Reuse genereller Ideen (Pattern)

Metainformationen

- Unabdingbar für Generative Programmierung sind Metainformationen!
- Metainformation gibt es auf der Ebene:
  1. Entwurf: Repository
  2. Compiletime: Sprachmittel
  3. Laufzeit: Introspection, Invokation

Metainformationen (2)

Metainformationen beinhalten z.B.

- Entwurfsinformationen
  - Informationen zu Klassen
    - Attribute: Namen, Typen, Rechte
    - Methoden: Namen, Parameter, Typen, Rechte
    - Weitergehende Entwurfsinformationen (UML)
- Laufzeitinformationen
  - Informationen zu Klassen
    - Werte von Attributen
    - Adressen von Methoden
    - Call Stack, Root Objects

Metaprogrammierung

- Dient dem Debugging/Logging
- Ermöglicht neue Programmierebenen
- Ermöglicht „generisches Memento“
- Vereinfacht die Implementation von Scripting
- Muss auch auf seiner Ebene OO-Prinzipien einhalten!

Scripting

- Ermöglicht schnelle Programmierung und das Verbinden von Komponenten
- Pattern: Bridge
- Ermöglicht Trennung von „Core“ Part und Oberfläche
- Konfiguration ist Programmierung
- Nachteile: neigt zum Verselbstständigen, oft keine OO im Scripting möglich, weiteres Programmierelement

Multiprocessing und OO

- Arbeit teilen
- Viele Architekturen
- Threads oder Prozesse
- Interprozeßkommunikation
- Verriegelung und Verklemmung
- keine Beweisbarkeit für „Nichtverklemmung“
- Sicher _NUR_ bei bestimmten Pattern:
  1. Pipe
  2. Supervisor Worker
  3. GenderChanger
  4. OnIdle (unter Vorbehalt)

Error Handling und Exceptions

- Was ist ein Fehler oder eine Ausnahme, was ist eigentlich normal?
- Was muss Fehlerbehandlung leisten:
  - Aufräumen
  - sichere Erkennung beim Aufrufer
  - Weitergabe bei Nichtbehandlung
  - Trennung von Code und Fehlerbehandlung (Vor- und Nachteil)
- Exceptions als zusätzliches Flow-Konstrukt
- Fangen von Ausnahmen
- Regel: Ausnahmebehandlung NUR zur Behandlung von nicht normalen Abläufen!

Einschub: Programmieren und Formatieren

Strukturierung ist dein Freund

Code entwickelt sich. Wenn er sich entwickelt strukturiere früh und konsequent. Sobald du ein Muster in deinem zu lösenden Problem entdeckst, fasse es in eine Struktur in deinem Code.

(Ralf)

Strukturierung:

1. Einteilung in Files/Klassen
2. Einteilung in Subprojekte/Directories
3. Erkennung von Gemeinsamkeiten und Abstraktion

Einschub: Programmieren und Formatieren

- Namen sind wichtig: (nicht Schall und Rauch)

  Man kann sehr viel schneller Schreiben als Denken. Wähle keine kryptischen Namen. Bezeichner, über die man Nachdenken muß wenn man sie wieder liest kosten viel mehr Zeit als gleich den richtigen Begriff zu suchen. Bilde Namen immer nach den gleichen Regeln. Wenn Funktionen, die eine bestimmte Aufgabe haben immer gleich geschrieben werden, wird es dir auch in großen Projekten nicht schwer fallen einen lange nicht verwendeten Funktionsnamen ohne in der Dokumentation nachzuschauen richtig zu verwenden. (Ralf)

- Fazit: Konventionen erleichtern das Leben

Einschub: Programmieren und Formatieren

- Formatierung ist wichtig:

Gut formatierte Quellen helfen dir den Code zu verstehen. Gewöhne dir einen Stil an nach dem du deine Quellen formatierst und versuche dich konsequent daran zu halten. Versuche in deinem Stil mit so wenig Regeln wie möglich auszukommen. Falls du ein Beispiel für sinnvolle Formatierung brauchst sehe dir Java Quellen an. Wenn du an einem Fremdprojekt arbeitest übernimm dafür den Formatierungsstil dieses Projekts. Tabulatoren sind dein Feind - gehe davon aus, daß an einem anderen Editor als deinem eigenen die Tabs anders eingestellt sind als bei dir.

(Ralf)

- Lesbarkeit und Einrückung

Einschub: Programmieren und Formatieren

- „Tricky Programming“ ist dein Feind:

  Versuche nicht clever zu Programmieren sondern lesbar. Wenn du über deinen Code nachdenken mußt, wenn du ihn einen Monat später wieder liest, dann ist er schlecht geschrieben oder es gehört ein Kommentar dran - meist ist er schlecht geschrieben.

  (Ralf)

- *Schaue dir verschiedenen Code von anderen Leuten an und versuche ihn zu verstehen. Anschliessend programmiere und dokumentiere so, wie du es gerne vorgefunden hättest. (Alex)*

Mengen und Maße

- Mengengerüst aufstellen
- Bedeutung wird durch technische Entwicklung innerhalb eines Projektes meist kleiner
- Es findet sich immer ein Limit!
- *Die am schwersten zu behebenden Fehler in einem Projekt sind Fragilität gefolgt von fehlender Skalierbarkeit. (Alex)*

Einschub: Optimierung

- Sollte am Algorithmus und im Modell erfolgen.
- Beachte Roundtrips/Latenzen:

  kleine Latenz kann man nicht kaufen. (1/7 Sekunde = 1* um die Erde)
- Skalierung beachten
- Zu frühe Optimierung ist die Mutter allen Übels:

  Programme haben meist _sehr_ wenig Code, der wirklich oft durchlaufen wird. Es macht keinen Sinn an anderen Stellen Laufzeitverhalten über Programmstruktur zu stellen. Man braucht _sehr_ viel Übung um Performanceprobleme durch Codereview zu finden. Profiler sind dein Freund. (Ralf)

Einschub: Datenbank und OO

- (fast) Alle Programme sind DB-Programme
- Relationale Datenbanken
- Abbildung von Objekten auf Relationen
- Bemühungen der DB-Hersteller zur OO-Integration
- Tools (Generatoren) erleichtern die Arbeit
- OO-DB´s
- Klassendiagramm->ER-Schema und vice versa

Einschub:Aspektorientierung

- Mehrere Aspekte im Design oft ohne Berührung
- Typischerweise als “Rules”
- Oft über Generatoren oder Präprozessoren eingebaut
- orthogonalisiert die Programmierung in einer weiteren Ebene

Einschub:Extreme Programming

- Testzentrisches Programmieren
- Erst wird eine Testszenario erarbeitet dann das (oft bestehende) Programm stückweise umgebaut
- Gut geeignet zum Aufräumen

Einschub: Design by Contract

- Programmierung mit Zusicherung
- Test durch Assertions
- Precoditions
- Postconditions
- Invariants

Schlussbemerkungen

- Bitte auf die Klausur vorbereiten! (siehe Web)
- Vielen Dank!
- Anregungen sind willkommen!