3 Objektorientierte Programmierung

3.1 Überblick

- Motivation für das objektorientierte Paradigma
- Software-Design Methoden
- Grundbegriffe der objektorientierten Programmierung
- Fundamentale Konzepte des objektorientierten Paradigmas
- Objektorientierte Analyse und Design
- Design Patterns

3.2 Literatur

Boo94. Grady Booch, *Object-Oriented Analysis and Design (with Applications)*, Benjamin/ Cummings, Redwood (CA), 1994.

CW85. Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", *Computing Surveys*, Vol. 17, No. 4, Dec. 1985.

GHJ+97. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, 10th print, Addison-Wesley, 1997

Str93. Bjarne Stroustrup, "A History of C++", ACM SIGPLAN Notices, Vol. 28, No. 3, pp.271 - 297, Mar. 1993.

Str00. Bjarne Stroustrup. *The C++ Programming Language* (Special Edition). Addison Wesley. Reading Mass. USA. 2000.

Weg87. Peter Wegner, “Dimensions of Object-Based Language Design”, *OOPSLA '87 – Conference Proceedings,* pp. 1-6, San Diego (CA, USA), published as *SIGPLAN Notices*, Vol. 22, No. 12, Dec. 1987.

Weg90. Peter Wegner, “Concepts and Paradigms of Object-Oriented Programming”, *ACM OOPS Messenger*, No. 1, pp. 8-84, Jul. 1990.

3.3 Motivation für das objektorientierte Paradigma

1 Ziele

- Mithalten mit der steigenden Komplexität großer Softwaresysteme
- Produktivität des Programmierers steigern
  - Entwurfsmuster für häufig wiederkehrende Probleme
  - Wiederverwendung existierender Software(teile)
  - bessere Erweiterbarkeit von Software
  - bessere Kontrolle über Komplexität und Kosten von Software-Wartung
- Übergang von einer maschinen-nahen Denkweise zu Abstraktionen der Problemstellung

3.4 Software-Design Methoden

1 Einordnung nach Booch (aus [Boo94])

- Top-down structured design (composite design)
- Data-driven design
- Object-oriented design

2 Klassen von Programmiersprachen

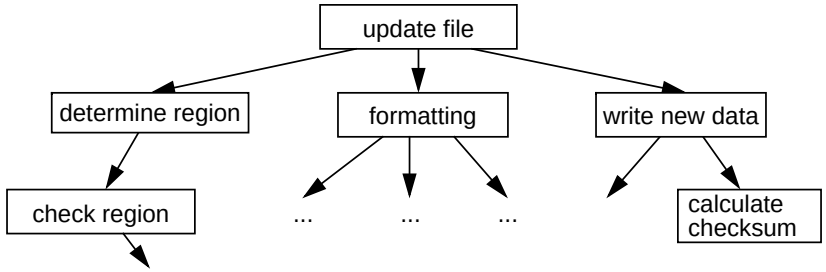
- Prozedural / imperativ
- Funktional
- Objektorientiert

3 Top-down Structured Design (Composite Design)

- Einheit für Dekomposition: **Unterprogramm**
- **Algorithmische Dekomposition** zur Zerlegung größerer Probleme
- Eignung für die Strukturierung heutiger, sehr großer Softwaresysteme wird bezweifelt
- *Top-down strutured design* erfasst nicht:
  - Datenabstraktion & *Information Hiding*
  - Nebenläufigkeit
- Probleme bei sehr komplexen Aufgabenstellung und objektbasierten oder objektorientierten Programmiersprachen
- Häufig eingesetzte Design-Methode
- Prozedurale Sprachen ideal für die Implementierung geeignet

3 Top-Down Structured Design (Composite Design) (2)

- Beispiel

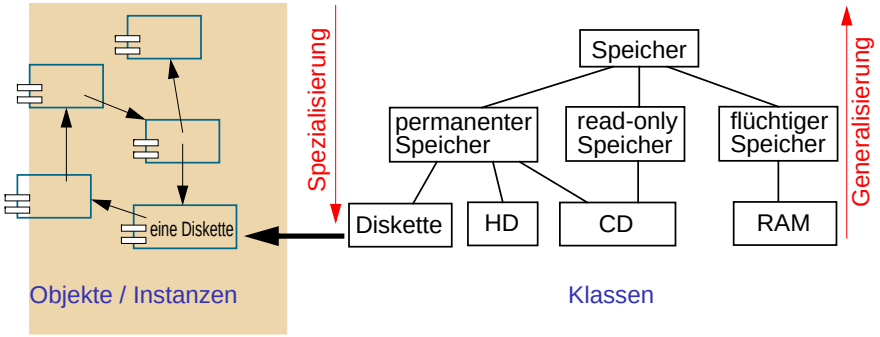


4 Objektorientiertes Design

Bertrand Meyer:
Rechner führen Operationen auf bestimmten Objekten aus;
um flexiblere und wiederverwendbare Systeme zu erhalten,
ist es daher sinnvoller, die Software-Struktur
an diesen Objekten statt an den Operationen zu orientieren.

4 Objektorientiertes Design (2)

- Softwaresystem wird als Sammlung kooperierender Objekte modelliert
- einzelne Objekte sind Instanz einer Klasse in einer Hierarchie von Klassen
- Beispiel einer Klassenhierarchie:



4 Objektorientiertes Design (3)

- Konzepte in der Struktur moderner Programmiersprachen reflektiert
  - ➤ Smalltalk
  - ➤ C++
  - ➤ Eiffel
  - ➤ Java
  - ➤ Ada
- Grundlage: objektorientierte Dekomposition
- Vorteile:
  - + Wiederverwendung gemeinsamer Mechanismen
    - ➔ Software wird kleiner
  - + Software leichter zu ändern und weiterzuentwickeln
  - + Ergebnisse weniger komplex
  - + Besseres Verständnis des Auftraggebers für die Problemlösung

3.5 Objektorientierte Programmierung

1 Definition (Grady Booch)

OOP ist eine Methode der Implementierung, in der Programme in Form von

Mengen kooperierender Objekte

organisiert sind, wobei jedes Objekt

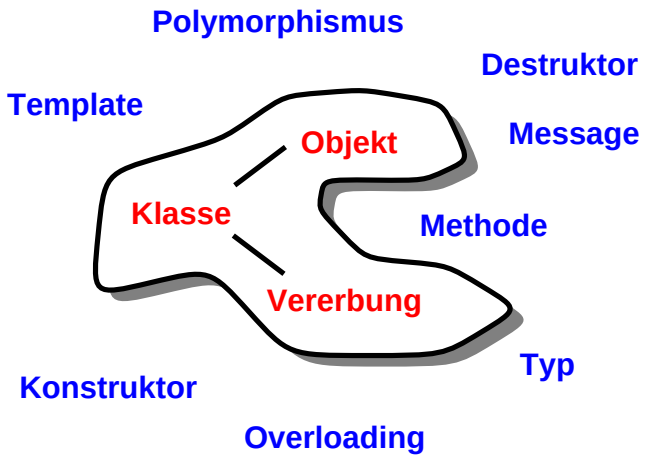
Instanz einer Klasse

ist und die Klassen Bestandteil einer über

Vererbungsbeziehungen

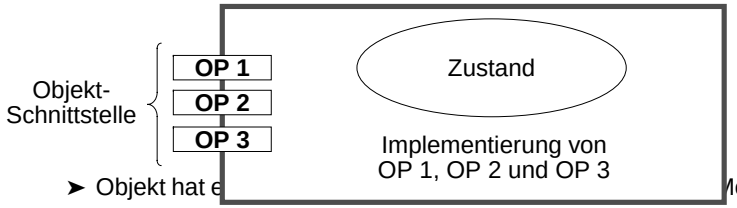
definierten Hierarchie von Klassen sind.

2 Grundbegriffe



3 Objekte & Methoden

- Sicht des Software-Entwicklers:
  - ◆ ein Objekt ist ein "Ding" aus der Problemstellung
    - ➤ es hat einen Zustand
    - ➤ es hat Verhalten
    - ➤ es hat eine eindeutige Indentität
- Programmiertechnische Sicht
  - ➤ eine gekapselte Einheit von Daten und Funktionen auf diesen Daten

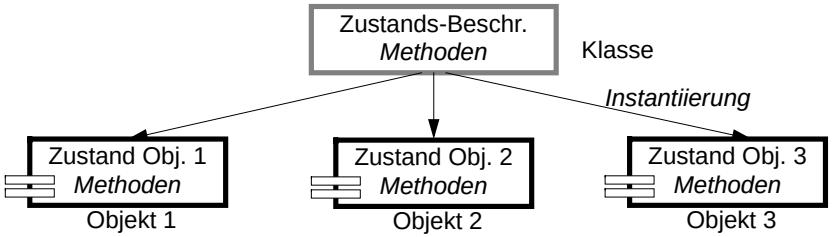


  - ➤ Objekt hat e ethoden)

➔ **Objektbasierte Programmiersprachen**

4 Klassen

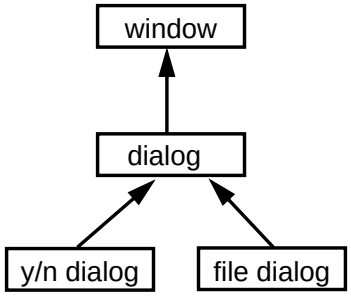
- Sicht des Software-Entwicklers
  - ◆ eine Klasse ist eine Menge von Objekten mit gleicher Struktur und gleichem Verhalten
- Programmiertechnische Sicht
  - ◆ Klasse = Schablone für Objekte
    - ➤ jedes Objekt ist **Instanz** einer Klasse
    - ➤ Objekterzeugung = **Instantiierung**



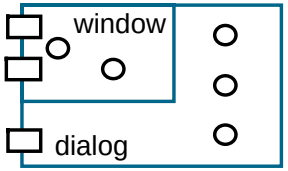
➔ **Klassenbasierte Programmiersprachen**
= Objekte & Klassen

5 Vererbung

- Beziehung zwischen Klassen, in der eine Klasse Struktur und/oder Verhalten übernimmt, das in einer anderen Klasse oder in mehreren anderen Klassen definiert wurde



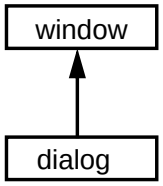
Vererbungshierarchie



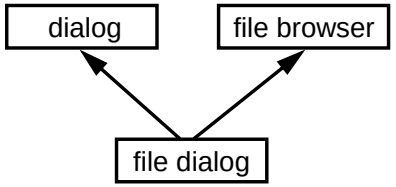
Instanz einer erbenden Klasse

5 Vererbung (2)

- ★ **Begriffe**
- **Unterklasse**
- **Oberklasse**
- **Einfache Vererbung**
- **Mehrfache Vererbung**



einfache Vererbung



mehrfache Vererbung

5 Vererbung (3)

- ★ **Sicht des Software-Entwicklers**
- Spezialisierung / Generalisierung von Klassen
- Gemeinsame Aspekte von Klassen werden in Oberklasen zusammengefasst
- Abstraktionshierarchie:
  - ◆ von allgemeineren Klassen zu spezialisierteren und umgekehrt
- Dokumentation der Beziehung zwischen Klassen

5 Vererbung (4)

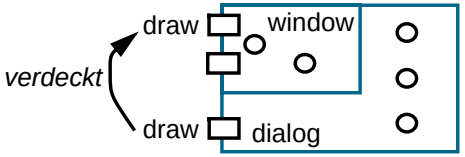
★ **Programmiertechnische Sicht**

- Erweiterung einer existierenden Klassenimplementierung um
  - ➤ zusätzliche Methoden
  - ➤ weitere Daten
- Wiederverwendung von Code:
  es ist keine Reimplementierung der geerbten Daten und Methoden erforderlich
- Reimplementierung einer Methode ist möglich, wenn die Methode der Oberklasse für die Unterklasse nicht passt
- Methoden der Oberklasse können an einem Objekt der Unterklasse aufgerufen werden
- Modifikationen der Oberklasse wirken auf alle Unterklassen (zentrale Softwarepflege)

5 Vererbung (5)

★ **Reimplementierung**

- Reimplementierung einer Methode in der Unterklasse:
  - ➤ verdeckt die Methode der Oberklasse



  - ➤ Default-Verhalten: Aufruf der Methode der Unterklasse
  - ➤ Aufruf der der reimplementierten Methode der Oberklasse?

6 Vererbung in C++

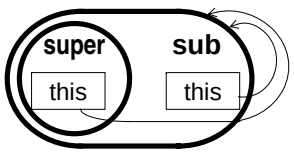
- Unterklasse erbt Variablen und Methoden von der Basisklasse
- Unterklasse kann Basisklasse modifizieren
  - ➤ zusätzliche Methoden und Variablen
  - ➤ veränderte Methoden
- Methoden der Unterklasse haben Zugriff auf *public*- und *protected* - Bereich der Basisklasse
  - ➤ public-Basisklasse
    - ➔ die *Schnittstelle* der Basisklasse wird vererbt
  - ➤ private-Basisklasse
    - ➔ die *Schnittstelle* der Basisklasse wird *nicht* vererbt
- *private*-Daten und -Methoden der Basisklasse nicht sichtbar

7 Dynamisches Binden

- Entscheidung welche Methode ausgeführt wird erfolgt zur Laufzeit (dynamisch)

```
Window *w = new BorderedWindow();
w->display();
```

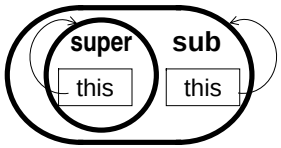
- Gilt auch, wenn ein Objekt eine Methode an sich selbst aufruft!
  - ◆ Beispiel:
    - – **`move()`** ruft am Ende **`display()`** auf, um das Fenster neu zu zeichnen
    - – **`BorderedWindow`** erbt **`move()`** von **`Window`**
    - – ein Aufruf von **`move()`** an einer Instanz von **`BorderedWindow`** ruft am Ende **`display()`** von **`BorderedWindow`** auf



der Zeiger this referenziert immer das "ganze Objekt" und nicht nur den Teil der Oberklasse

7 Dynamisches Binden (2)

- Ohne dynamisches Binden keine “echte Vererbung”
  - ➔ Selbstreferenz (Zeiger *this*) wird nicht richtig angepasst



8 Statisches Binden

- Entscheidung welche Implementierung einer Methode genommen wird fällt zur Übersetzungszeit
- In C++ werden nur “virtual” -Methoden dynamisch gebunden
  - ◆ alle anderen Methoden werden generell statisch gebunden
- In Java werden alle Methoden dynamisch gebunden
  - ◆ statisches Binden kann durch das Schlüsselwort **final** in der Methode-deklaration erzwungen werden
  - ◆ solche Methoden können in Unterklassen nicht reimplementiert werden

```
public final void incr() { value += step; }
```

  - ➔ Compiler kann statisch binden

3.6 Fundamentale Konzepte des objektorientierten Paradigmas

- Abstraktion
  - ◆ Kapselung
  - ◆ Datenabstraktion in OOP
- Modularisierung
- Hierarchie
- Typisierung
  - ◆ Typhierarchie
  - ◆ Polymorphismus
  - ◆ Generizität
- Nebenläufigkeit
- Persistenz

1 Abstraktion

Grundlegendes Konzept zur Lösung komplexer Probleme

- Für Zusammenhänge relevante Aspekte hervorheben
- Details vernachlässigen

➔ Objektorientierung

- wichtig:
  - ➤ Signatur eines Objekts
  - ➤ Semantik eines Objekts

  } Sicht von aussen

    - ➥ contract model: Sicht von aussen = Vertrag mit anderen Objekten
- unwichtig:
  - ➤ Implementierung eines Objekts

- zuerst die Abstraktion beschreiben, dann Implementierung vornehmen

2 Kapselung

= Information Hiding
Verbergen der Implementierung einer Abstraktion vor den Benutzern der Abstraktion

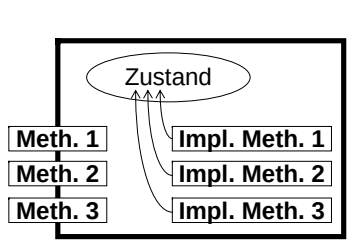
- Gegenstück zu Abstraktion
  - ➤ Abstraktion stellt die äußeren Eigenschaften eines Objekts heraus
  - ➤ Kapselung verbirgt die Interna
- Grundvoraussetzung für Abstraktion

  B. Liskov *Damit Abstraktionen funktionieren, müssen Implementierungen gekapselt sein*
- Kapselung in der Objektorientierung
  - ◆ Darstellung des Objektzustands
  - ◆ Implementierung der Methoden

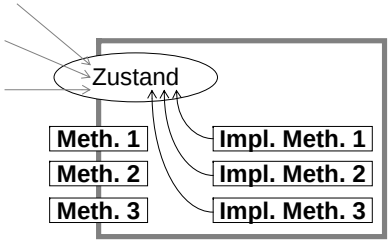
➔ Abstrakter Datentyp, Datenabstraktion

2 Datenabstraktion in OOP

➔ auf Objektzustand kann nur durch die Methoden des Objekts zugegriffen werden



Objekt als Implementierung eines ADT



Objekt ohne Datenabstraktion

- Datenabstraktion in C++ und Java
  - ◆ Sichtbarkeitsbereiche (*Scope*-Regeln)

3 Modularisierung

Aufteilung eines Programms in einzelne Komponenten kann seine Komplexität reduzieren

- ◆ Teilproblem überschaubarer
- ◆ Teilprobleme Entwicklergruppen zuzuordnen
- ◆ Module = separate Softwareentwicklungseinheiten

- **vor allem: Aufteilung erzeugt Grenzen = Schnittstellen**
  - ➔ Schnittstellen müssen klar definiert werden
  - ➔ Schnittstellen müssen dokumentiert werden
- viele Programmiersprachen unterscheiden zwischen Schnittstelle und Implementierung eines Moduls
- *Structured Design*: Gruppierung von Unterprogrammen
- OOD: Gruppierung von Objekten und Klassen

4 Hierarchie

- ◆ Abstraktion & Kapselung helfen Details von Komponenten zu verbergen
- ◆ Modularisierung hilft zusammengehörende Abstraktionen zu bündeln

- Überblick über große Problemstellungen immer noch schwierig
  - ➔ zu viele Abstraktionen
  - ➔ Hilfsmittel zur Organisation von Abstraktionen notwendig
- Abstraktionen bilden oft Hierarchien
  - ➤ gemeinsame Eigenschaften → allgemeinere Abstraktionen
  - ➤ Unterschiede → Spezialisierungen
  - ➔ Hierarchie: Ordnung auf Abstraktionen
- Hierarchie & Objektorientierung
  - ➤ Klassenstruktur: Vererbung → "**Art-von**"-Hierarchie (*kind of / is a*)
  - ➤ Objektstruktur: Aggregation → "**Teil-von**"-Hierarchie (*part-of*)

5 Typisierung

- Typkonzept baut auf ADT-Theorie auf
- Typisierung ermöglicht die Überprüfung von Ausdrücken auf Typ-Kompatibilität zur Übersetzungszeit

  ➔ Vermeidung von Fehlern
- Strenge Typisierung
  Konformität aller Typen in einem Ausdruck wird garantiert

  ➔ Statische Typisierung
  Konformität wird komplett zur Übersetzungszeit überprüft
  - ➤ Einschränkung der Flexibilität
    - kompatible Typen
    - dynamisches Binden

  ➔ mehr Flexibilität durch Polymorphismus und Generizität

6 Typhierarchie

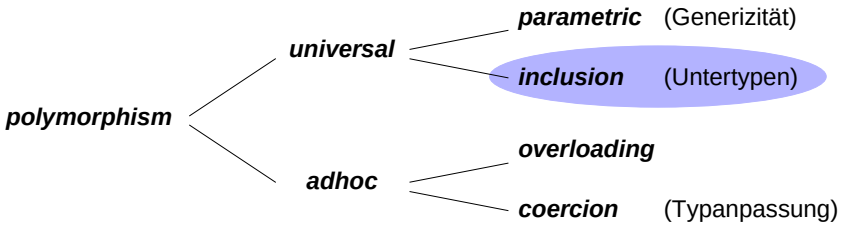
- Häufig 1:1-Relation zwischen Klassen und Typen — aber nicht notwendig
  - ➤ mehrere Klassen können einen Typ implementieren
  - ➤ eine Klasse kann unterschiedliche Typen implementieren
    •
- Hierarchie bei Klassen: Oberklasse ← Unterklasse
  - ◆ Ziel: Wiederverwendung von Implementierung
  - ◆ Unterklasse nicht notwenigerweise typ-konform zu Oberklasse
- Hierarchie bei Typen: Obertyp ← Untertyp
  - ◆ Ziele: – Verhaltens-Vererbung
    – Beschreibung **konformer** Typen (➔ Polymorphismus)
- Typvererbung (*Subtyping*) als Mechanismus zur Ableitung von Typen

  ➔ Zusammenhänge zwischen Typen werden übersichtlich
  - – Identifikation konformer Typen

7 Polymorphismus

- die Fähigkeit, verschiedene Formen anzunehmen
  - ➤ mehr als ein Typ für Werte oder Variablen
  - ➤ verschiedene Typen als Parameter zu Funktionen
  - ➤ verschiedene Typen als Operanden eines Operators
- Beispiel:

  +-Operator arbeitet mit int- und real-Werten
- Klassifikation [CW85]



8 Polymorphismus in C++

- *overloading polymorphism*
  - ➤ Function-name overloading
  - ➤ Operator overloading
- *inclusion polymorphism*
  - ➤ public Vererbung
- *coercion polymorphism*
  - ➤ Cast-Operatoren

8 Polymorphismus in C++ (2)

★ **Inclusion-Polymorphism — *DER Polymorphismus in OOP***

Vererbung + Virtuelle Methoden + Objekt-Referenzen

- Eine Objektreferenz (Zeiger) hat einen Typ (= Klasse)
  - ◆ Instanzen dieser Klasse und ihrer Unterklassen können dem Zeiger zugewiesen werden
  - ◆ bei einem Methodenaufruf wird die tatsächliche Implementierung der Methode nicht von der Klasse des Zeigers, sondern von der Klasse des aktuellen Objekts bestimmt

➔ Man kann jederzeit einer Referenz irgendein Typ-konformes Objekt zuweisen und alles funktioniert

- ◆ man kann es als Parameter einer Methode übergeben
- ◆ der Programmierer der Methode musste den neuen Untertyp nicht kennen — so lange er konform zu dem Obertyp ist, den seine Methode erwartet

8 Polymorphismus in C++ (3)

★ **Virtuelle Methoden & Inclusion Polymorphism — Beispiel:**

```
class geo_obj {                          // general superclass
  public:
    virtual void draw();
};

class circle : public geo_obj {          // subclass
  public:
    void draw();
}

class square : public geo_obj {          // subclass
  public:
    void draw();
}

main () {
  geo_obj *ptr;
  ptr = new circle;
  ptr->draw();
...
```


9 Typen und C++: Abstrakte Klassen

- Basisklasse deklariert Methoden und Parameter, definiert sie aber nicht
  - – *pure virtual function*

  ➔ Basisklasse beschreibt einen Typ
- Subklassen definieren unterschiedliche Implementierungen der Methoden

  ➔ jede Subklasse stellt eine Implementierung des Typs dar
- Basisklasse ist nicht instantiierbar
- Beispiel:

```
class geo_obj {                          // abstract class
  public:
    virtual void draw() = 0;             // pure virtual function
};
class circle : public geo_obj {          // subclass
  public:
    void draw() { ... }
}
```


10 Typen und Java: Interfaces

- 2 Möglichkeiten einen Typ zu deklarieren:
  - ◆ im Rahmen einer Klassendefinition

    ➔ Klassenvererbung führt automatisch zu Typvererbung
  - ◆ durch eine Interface-Deklaration

    ➔ separate Typbeschreibung
- Beispiel:

```
public interface Printable {
    public void Print();
}

public class MyPoint extends Object implements Printable {
    ....
    public void Print() {
        System.out.println("x="+x+" y="+y);
    }
}
```

10 Typen und Java: Interfaces (2)

- Vererbung & Mehrfachvererbung auf Interfaces
- eine Klasse kann mehrere Typen implementieren
- Typkonformität ist transitiv
- Exceptions sind auch Bestandteil der Typ-Schnittstelle
- Beispiele:

```
interface Streamable extends FileIO, Printable {
    // additional Methods
    public void test() throws TestException;
}

class Test implements Streamable, Testinterface {
    ...
}
```


11 Generizität (*Genericity*)

- Möglichkeit, den Typ von programmiersprachlichen Einheiten durch Parameter festzulegen
- OOP: generische Klassen

  generische Klasse ➔ Instantiierung (+Parametrierung) ➔ tatsächliche Klasse
  tatsächliche Klasse ➔ Instantiierung von Objekten
- Beispiel:
  - allgemeine Klasse Stack
    - int-Stack
    - real-Stack
    - string-Stack
- Generizität kann weitgehend mit Vererbung nachgebildet werden
- realisiert in Ada, Eiffel, C++ (ab. V 3.0, *Templates*) und Java (ab Java 5)

12 Generizität und C++: Templates

- Ziel: Klassendefinition ohne Festlegung auf bestimmte Typen
  - dynamische Typprüfung zur Laufzeit
  - statische Typprüfung zur Übersetzungszeit + parametrierbare Klassen
- Template = parametrierbare Klasse
- Beispiel:

```
template <class T> class stack {
  private:
    int index;
    T *array;
  public:
    void stack(int n)
              { index = 0; array = new T[n]; }
    void push(T elem)
              { array[index++] = elem; }
    T pop(void)
              { return(array[--index]); }
};
```


13 Generizität und Java (ab Java 5)

- auf den ersten Blick ähnlich zu C++-Templates,
  aber viele Unterschiede im Detail
  - ➤ nur eine generische Klasse für alle Objekte
  - ➤ Bounds erlauben die Einschränkung von Typ-Variablen auf bestimmte Subtypen
- Typ-Parametrierung auch auf Interfaces möglich
- Beispiel:

```
class AnimalStack <T extends Animal> {
    private int index;
    private T[] array;

    public void AnimalStack(int n)
              { index = 0; array = (T[])new Object[n];}
    public void push(T elem)
              { array[index++] = elem; }
    public T pop()
              { return(array[--index]); }
};
```

14 Nebenläufigkeit (Concurrency)

Nebenläufigkeit: mehrere Aktivitätsträger werden parallel von mehreren Prozessoren bearbeitet oder von einem Prozessor simuliert

- Nebenläufigkeit orthogonal zu Objektorientierung
  aber: Komplexität einer Problemlösung wird durch Nebenläufigkeit erhöht
- Granularität: Nebenläufigkeit / Objekte (Kapseln)
  - ➤ Nebenläufigkeit feiner granular
    - ➔ NL auch objektintern
  - ➤ Nebenläufigkeit grober granular
    - ➔ NL nur objektextern
- Integration der Kontrolle von Nebenläufigkeit in objektorientierte Sprachen
  - ➤ orthogonale Sprachen
  - ➤ nicht-orthogonale Sprachen

15 Nebenläufigkeit und Java

- Thread-Konzept und Koordinierungsmechanismen sind in Java integriert
  - ➥ nicht-orthogonale, nicht-uniforme Sprache bzgl. Nebenläufigkeit
- Erzeugung von Threads über Thread-Klassen
- Beispiel

```
class MyClass implements Runnable {
        public void run()  {
                System.out.println("Hello\n");
        }
}

....
MyClass o1 = new MyClass();    // create object
Thread t1 = new Thread(o1);    // create thread to run in o1

t1.start();                    // start thread

Thread t2 = new Thread(o1);    // create second thread to run in o1

t2.start();                    // start second thread
```


15 Nebenläufigkeit und Java (2)

- ★ **Koordinierungsmechanismen**
- Monitore: exclusive Ausführung von Methoden eines Objekts
  - ◆ Beispiel:

```
class Bankkonto {
    int value;
    public synchronized void AddAmmount(int v) {
        value=value+v;
    }
    public synchronized void RemoveAmmount(int v) {
        value=value-v;
    }
}
...
Bankkonto b=....
b.AddAmmount(100);
```

  - ◆ Conditions: gezieltes Freigeben des Monitors und Warten auf ein Ereignis

15 Nebenläufigkeit und Java (3)

- ★ Koordinierungsmechanismen in Java 5
- mehrere APIs zur Unterstützung von Nebenläufigkeit
- Atomare Operationen
  - ➤ z.B. getAndSet oder compareAndSet auf Boolean, Integer oder Long
  - ➤ Paket java.util.concurrent.atomic
- Locks und Condition Variablen
  - ➤ orthogonale Implementierung zusätzlich zu synchronized/wait/notify
  - ➤ Paket java.util.concurrent.locks
- Umfangreiches Paket java.util.concurrent
  - ➤ thread-sichere Container, Queues, Collections
  - ➤ Executor-Framework, Thread-Pools, Futures, Scheduling
  - ➤ Semaphore, Latches, Barrieren

16 Persistenz

★ Motivation für Persistenz

■ “aktive” Daten → Programmiersystem

■ “passive” Daten → DBMS oder Dateisystem

➔ 2 Sichten auf Daten

➔ Nachteile für den Anwender

- ➤ Konvertierung notwendig
- ➤ Datenschutz des Programmiermodells geht verloren

★ allgemeine Definition:
Persistenz ist die Eigenschaft von Daten, das Ende einer Umgebung, in der sie entstanden oder benutzt wurden, zu überdauern

16 Persistenz (2)

★ **Persistenz in objektorientierten Systemen**

■ Objekte überleben das Ende der Umgebung (Aktivitätsträger, Anwendungsausführung), in der sie instantiiert oder benutzt wurden

■ In OO-Betriebssystemen mächtiger Basis-Mechanismus

- ➤ Datenspeicherung
- ➤ Datentransport

■ Beispiele

- ➤ Dateisysteme
- ➤ Datenbanksysteme
- ➤ persistente Kommunikationsobjekte

■ Eigenschaften des objektorientierten Programmiermodells gehen automatisch auf Mechanismen über, die damit erstellt wurden