# C Objektorientierte Programmierung

## C.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






## C.2 Literatur

ABC83. M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshot and R. Morrison, “An Approach to Persistent Programming”, *The Computer Journal*, Vol. 26, No. 4, pp. 360-365, 1983.

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

Jac92. I. Jacobson. Object-Oriented Software Enineering — A Use Case Driven Approach. Addison-Wesley, 1992.

MaM88. Ole Lehrmann Madsen, Birger Møller-Pedersen, "What object-oriented programming may be — an what it does not have to be", *ECOOP ’88 – European Conference on OO Programming*, pp. 1 - 20, S. Gjessing, K. Nygaard [Eds.]; Springer Verlag, Oslo, Norway, Aug. 1988.

Mey86. Bertrand Meyer, "Genericity versus Inheritance", *Conference on Object-Oriented Programming Systems, Languages, and Applications - OOPSLA ’87*, pp. 391 - 405, Portland (Oreg., USA), published as *SIGPLAN Notices*, Vol. 21, No. 11, Nov. 1986.

Mey88. Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall Inc., Hemel Hempstead, Hertfordshire, 1988.

Oes97. B. Oestereich. Objektorientierte Softwareentwicklung: Analyse und Design. Oldenbourg, 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.

BRJ98. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual.* Addison Wesley, 1998.

RJB98. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language User Guide.* Addison Wesley, 1998.

JBR98. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, 1998.

# C.3 Motivation für das objektorientierte Paradigma

## 1 Ziele

- Mithalten mit der steigenden Komplexität großer Softwaresysteme

  Booch [Boo94] spricht von *industrial-strength Software* und meint damit komplexe Softwaresysteme, wie sie heute in der Industrie (Prozeßsteuerung, Bank- und Versicherungswesen, Buchungssysteme, etc.) in großem Umfang eingesetzt sind. Solche Softwaresysteme sind so komplex, daß es in der Regel für einen Entwickler nicht möglich ist, alle Details des Designs zu überblicken. Die Softwaresysteme haben eine sehr lange Lebensdauer, haben viele Nutzer und werden von vielen unterschiedlichen Personen gewartet und erweitert.

  Mit der ständig steigenden Leistungsfähigkeit (Geschwindigkeit, Speicherumfang, Parallelverarbeitung) von Rechnern nimmt auch die Komplexität der damit bearbeitbaren Probleme zu. Die ursprünglichen Softwareentwicklungsmethoden reichen zur Bewältigung dieser Komplexität nicht mehr aus. Objektorientierung ist eine besser Art Software zu entwickeln und zu strukturieren.

  → Softwarekrise: Hardware wird immer leistungsfähiger, Software wird größer, die aufgrund der Konkurrenzsituation zur Verfügung stehenden Entwicklungszeiträume werden gleichzeitig kürzer, die Kosten für Wartung und Weiterentwicklung steigen dramatisch. Letzlich stehen nicht mehr genug gute Softwareentwickler für die Entwicklung und Pflege der benötigten Software zur Verfügung.

  → Lösung:

- Produktivität des Programmierers steigern
  - ◆ Entwurfsmuster für häufig wiederkehrende Probleme

    → Design-Patterns
  - ◆ Wiederverwendung existierender Software(teile)

    Durch mehrfache Instantiierung allgemein einsetzbarer Softwaremodule (Klassenbibliotheken) sowie durch Anpassung existierender Module an erweiterte oder abgewandelte Anforderungen (Vererbung).

    Frameworks geben komplette Anwendungsgerüste vor, die durch Anpassung einzelner Komponenten zu einer maßgeschneiderten Anwendung geformt werden können.
  - ◆ bessere Erweiterbarkeit von Software

    Durch Modularisierung und klare Definition von Schnittstellen zwischen den einzelnen Softwarekomponenten
  - ◆ bessere Kontrolle über Komplexität und Kosten von Software-Wartung

    Modulgrenzen und -schnittstellen legen klare Grenzen fest.
- Übergang von einer maschinen-nahen Denkweise zu Abstraktionen der Problemstellung

  → Besseres Verständnis für die Problemstellung
  - Terminologie der Problemstellung findet sich in der Software-Lösung wieder

    → Lösung wird leichter verständlich






# C.4 Software-Design Methoden

## 1 Einordnung nach Booch (aus [Boo94])

Grundprinzip Dekomposition:
Zerlegen eines komplexen Problems in überschaubare Teilprobleme

- Top-down structured design (composite design)
  - Traditionelle Software-Design-Methode
- Data-driven design
  - Orientiert sich an der Abbildung von Eingabedaten auf Ausgabedaten
- Object-oriented design
  - "Moderne" Methode - findet seit einigen Jahren großen Zulauf
  - Bisherigen Erfahrungen zeigen vor allem Vorteile bei dem Design sehr großer Softwaresysteme

## 2 Klassen von Programmiersprachen

… zumindest die wichtigsten

- Prozedural / imperativ
  - z. B. Algol, Pascal, C – setzen direkt die Konzepte des Top-down structured design um
- Funktional
  - Lisp und seine "Nachfahren"
- Objektorientiert
  - Simula, Smalltalk, Eiffel, Java (und viele weitere)
  - C++ ist hybrid: prozedural auf der einen, voll objektorientiert auf der anderen Seite

# 3 Top-down Structured Design (Composite Design)

➔ wesentlich durch die traditionellen Programmiersprachen (wie Fortran oder Cobol) beeinflusst

- Einheit für Dekomposition: **Unterprogramm**
  - Resultierendes Programm hat die Form eines Baums in dem Unterprogramme ihre Aufgaben durch Aufrufe weiterer Unterprogramme erledigen

- **Algorithmische Dekomposition** zur Zerlegung größerer Probleme
  - Zentrale Sicht: Ausführung einer Problemlösung
    - Modularisierung: Identifizieren von Teilproblemen / Ausführungsschritten
    - Reihenfolge, in der Aktionen (=Teilproblemlösungen) auszuführen sind, wird hervorgehoben

- Eignung für die Strukturierung heutiger, sehr großer Softwaresysteme wird bezweifelt
  - Heutige Rechner können Softwaresysteme bewältigen, deren Umfang und Komplexität die der 60er und 70er um Größenordnungen übersteigt
  - Der Wert strukturierter Programmierung besteht nach wie vor, aber *Structured programming appears to fall apart when applications exceed 100 000 lines of code or so* [Boo94]
  - Wesentliche Ursache ist, dass nicht die zu bearbeitenden Daten, sondern die auszuführenden Aktionen im Mittelpunkt der Strukturierung stehen. Je größer ein Softwaresystem ist, desto größer wird die Datenmenge und die Menge der Prozeduren, die darauf arbeiten. Eine feste Zuordnung zwischen überschaubaren Teilmengen der Daten und Prozeduren bereits beim Softwaredesign wird durch algorithmische Dekomposition alleine aber nicht erreicht.

- *Top-down strutured design* erfasst nicht:
  - Datenabstraktion & *Information Hiding*
    - Beim Top-down strutured design operieren Teilproblemlösungen auf einem globalen Datenbestand — lokale Daten werden vor allem zur Zwischenspeicherung verwendet
  - Nebenläufigkeit

- Probleme bei sehr komplexen Aufgabenstellung und objektbasierten oder objektorientierten Programmiersprachen
  - *Structured design* resultiert in einer Problemstrukturierung, die nicht auf die Mechanismen in objektbasierten oder objektorientierten Sprachen abgestimmt ist

- Häufig eingesetzte Design-Methode
  - Grundlagen von *structured design* beruhen auf den Arbeiten von Wirth, Dahl, Dijkstra und Hoare

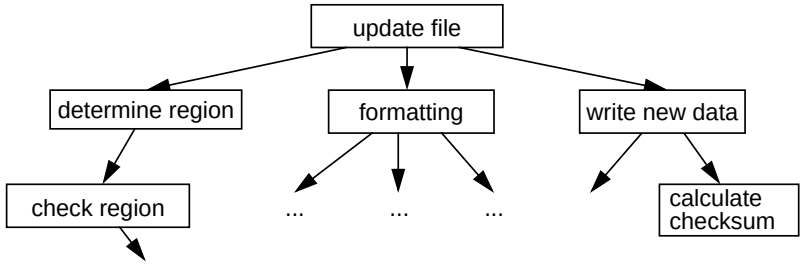
- Prozedurale Sprachen ideal für die Implementierung geeignet






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

- Beispiel




Die Problemstellung ist, eine Datei mit neuen Daten zu aktualisieren.

Die Aufgabe wird in mehrere Teilaufgaben zerlegt, die ihrerseits - wenn sie komplexer sind - weiter zerlegt werden:

- der Bereich für die Daten in der Datei muss festgelegt werden
  - Unteraufgabe hiervon ist, den Datenbereich zu überprüfen
- die Daten müssen vor der Ausgabe formatiert werden
- die neuen Daten müssen schließlich ausgegeben werden
  - Unteraufgabe ist hierbei, eine Checksumme zu berechnen

# 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.*

siehe auch [Mey88]



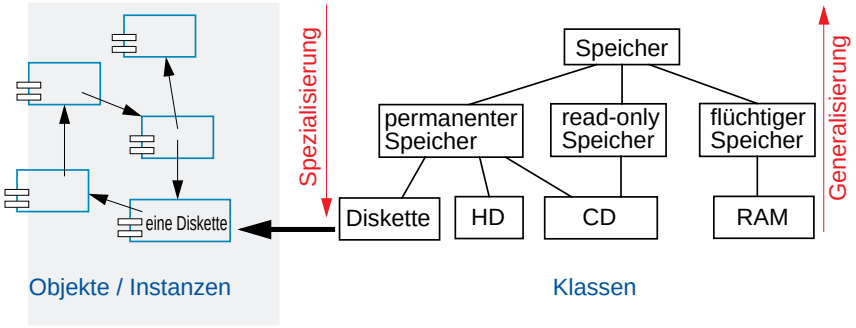



# 4 Objektorientiertes Design (2)

- Softwaresystem wird als Sammlung kooperierender Objekte modelliert
  - Es stehen nicht auszuführende Aktionen im Mittelpunkt, sondern Einheiten, die einen Zustand besitzen, ihre Dienste anderen Einheiten anbieten und zur Erledigung ihrer Aufgaben andere Einheiten in Anspruch nehmen.
  - Die Definition der Einheiten orientiert sich an den Einheiten des zu lösenden Problems

- einzelne Objekte sind Instanz einer Klasse in einer Hierarchie von Klassen
  - Eine Klasse beschreibt die Eigenschaften von Objekten = Instanzen dieser Klasse
  - Hierarchie entsteht durch die Beschreibung von Ober- und Unterklassen
    - Oberklasse: beschreibt allgemeine Eigenschaften von Objekten
    - Unterklasse: beschreibt Spezialisierung gegenüber der Oberklasse
      Instanzen der Unterklasse haben die durch die Oberklasse beschriebenen Eigenschaften, ergänzt oder abgeändert durch die in der Unterklasse beschriebene Spezialisierung

- Beispiel einer Klassenhierarchie:




  - Die Oberklasse **Speicher** beschreibt die allgemeinen Eigenschaften eines Speichers
  - Die Unterklassen **permanenter Speicher**, **flüchtiger Speicher** und **read-only Speicher** spezialisieren diese Eigenschaften — ein permanenter Speicher hat z. B. die Eigenschaft, dass sein Inhalt über einen Systemstart hinweg erhalten bleibt, während ein flüchtiger Speicher dies nicht garantiert. Ein read-only-Speicher würde bei einer Schreiboperation einen Fehler melden.
  - **Diskette**, **Hard Disk** und **CD** sind ihrerseits Unterklassen der (relativ hierzu) Oberklasse **permanenter Speicher** - **CD** ist aber auch gleichzeitig Unterklasse von **read-only Speicher** (d.h. sie hat sowohl Eigenschaften von permanentem, als auch von read-only Speicher).

Von den Klassen und der Klassenhierarchie streng getrennt zu betrachten sind die einzelnen Objekte.

  - Von einer Klasse (z.B. Diskette) kann es mehrere Instanzen (= Objekte) geben.
  - Auch bei Objekten kann es Hierarchien geben (wenn ein Objekt aus anderen quasi "zusammengebaut" ist) - das ist dann aber etwas ganz anderes als die Klassenhierarchie.

## 4 Objektorientiertes Design (3)

- Konzepte in der Struktur moderner Programmiersprachen reflektiert
  - ➤ Smalltalk
  - ➤ C++
  - ➤ Eiffel
  - ➤ Java
  - ➤ Ada
- Grundlage: objektorientierte Dekomposition
  - Dekomposition, orientiert an den Einheiten des *Problem domain* — nicht an den Einheiten oder Ausführungsschritten der Problemlösung (wie bei algorith. Dekomposition)
  - ➔ essentieller Unterschied zu algorithmischer Dekomposition und damit dem *Top-down structured design*
    - andere Art, über Dekomposition nachzudenken
    - Software-Architektur wird völlig anders
  - ➔ Software-Designer muß den *Problem domain* verstehen und überblicken — bei traditioneller Software-Entwicklung kommt diese (eigentlich selbstverständliche) Voraussetzung häufig zu kurz.
- Vorteile:

  **+ Wiederverwendung gemeinsamer Mechanismen**
  - Oberklassen beschreiben gemeinsame Eigenschaften, die in Unterklassen übernommen werden können.

  ➔ Software wird kleiner

  **+ Software leichter zu ändern und weiterzuentwickeln**
  - Änderungen sind immer örtlich begrenzt (auf Klasse und evtl. deren Unterklassen)
  - keine globalen Datenstrukturen
  - Wiederverwendung durch gemeinsame Oberklassen garantiert gemeinsame Eigenschaften
    - Änderung in der Oberklasse wirkt sich konsistent auf alle Unterklassen aus
    - keine Fehler bei Reimplementierungen
  - Weiterentwicklung durch Bildung von Unterklassen

  **+ Ergebnisse weniger komplex**
  - Inkrementelle Entwicklung aus kleinen, überschaubaren Systemen
  - OOD legt die Aufteilung großer Zustandsräume nahe (*separation of concerns*)

  **+ Besseres Verständnis des Auftraggebers für die Problemlösung**
  - Die Begriffswelt des Auftraggebers findet sich in der Softwarelösung wieder






# C.5 Objektorientierte Programmierung

## 1 Definition (Grady Booch [Boo94])

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

**Mengen kooperierender Objekte**

organisiert sind, wobei jedes Objekt

**Instanz einer Klasse**

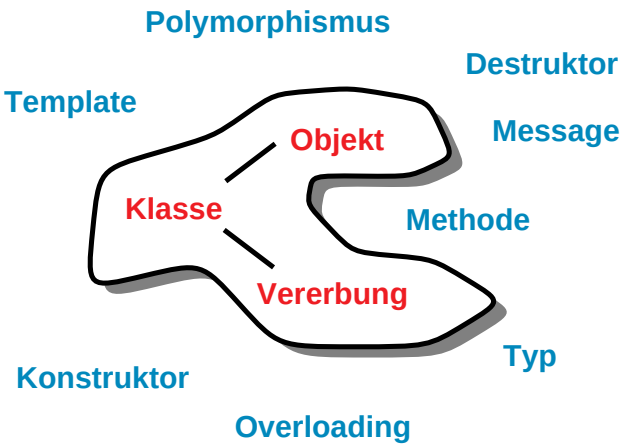
ist und die Klassen Bestandteil einer über

**Vererbungsbeziehungen**

definierten Hierarchie von Klassen sind.

- Diese Definition enthält die wesentlichen Grundbegriffe objektorientierter Programmierung:

# 2 Grundbegriffe




→ Objekt
- Objektorientierte Programmierung verwendet Objekte und nicht Algorithmen als die grundlegenden Bausteine der Problemlösung

→ Klasse
- Jedes Objekt ist Instanz einer Klasse

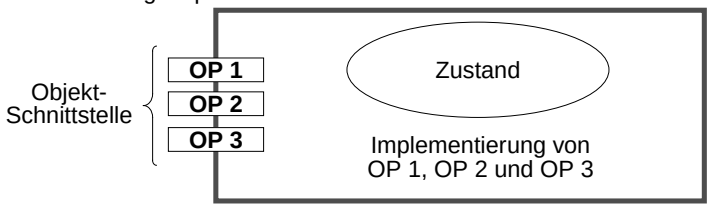
→ Vererbung
- Klassen stehen über Vererbungsbeziehungen in Beziehung miteinander






# 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




  - Zusammenfassung von Datenstrukturen und Operationen auf diesen Datenstrukturen zu einer Programmiereinheit
  - Klassisches Beispiel: Ein Stack, dessen Zustand als verkettete Liste implementiert ist und der die Operationen *push* und *pop* anbietet
  - Vorteile des Objekt-Konzepts:
    - – Logisch zusammengehörige Programmteile stehen auch im Programmtext beieinander
    - – Das Ergebnis von Operationen hängt nicht nur von den Aufrufparametern (wie bei Funktionen) ab, sondern auch vom Zustand des Objekts (→ Operationen mit "Gedächtnis")
    - – Teile des Objekts können vor Zugriff von "außen" geschützt werden — der interne Aufbau des Objekts kann "geheim" bleiben (**Information Hiding**)
  - ➤ ein Objekt hat eine klar definierte Schnittstelle (Operationen = Methoden)
    - Die Methoden eines Objekts definieren gleichzeitig das **Verhalten** des Objekts

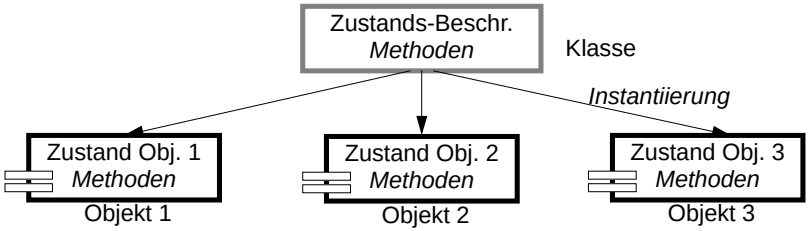
→ **Objektbasierte Programmiersprachen**
- Eine Programmiersprachen ist objekt-basiert, wenn sie Objekte als Sprachkonstrukt unterstützt [Weg87]
- Probleme:
  - – Benötigt man von einem Objekt mehrere Exemplare (z. B. mehrere Stacks, um unterschiedliche Informationen abzulegen), muß das Objekt entsprechend oft implementiert werden

  → Abhilfe: Schablone zur Erzeugung von Objekten

# 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**

- Auch wenn häufig nur ganz pragmatisch die programmiertechnische Sicht im Vordergrund steht (eine Klasse ist etwas, an dem man *new* aufrufen kann), ist vor allem die andere Sichtweise von großer Bedeutung, weil sie ein wesentliches Strukturmerkmal der Software in den Mittelpunkt rückt.
- Die Möglichkeit Klassen zu beschreiben und Instanzen zu erzeugen bildet auch den ersten großen Unterschied von prozeduralen zu objektorientierten Programmiersprachen.
- Alle Instanzen können den in der Klasse implementierten Code für die Methoden gemeinsam nutzen, haben aber jeweils eine eigene Version des Objektzustands (der Variablen), gemäß der Strukturbeschreibung in der Klasse.




- **Instanzvariablen** = Variablen eines Objekts (= Zustand)

→ **Klassenbasierte Programmiersprachen**
= Objekte & Klassen

Definitionen:

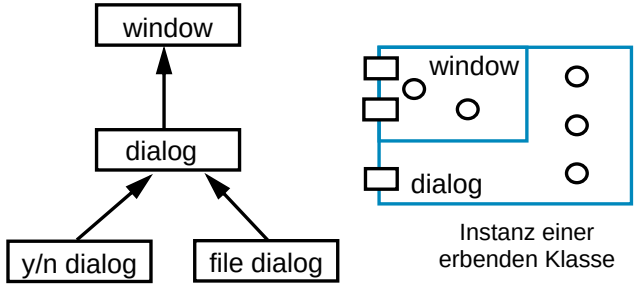
- Eine Programmiersprachen ist klassen-basiert, wenn jedes Objekt eine Klasse besitzt [Weg87]
- Programmiersprachen, die die Formulierung von Klassen und die Instantiierung von Objekten aus Klassen erlauben






# 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

- Vererbung (engl. ***Inheritance***) ist eine Technik, die es erlaubt, neue Klassen auf bereits existierenden Klassen aufzubauen, statt sie vollständig neu zu programmieren. Die neu entstehende Klassen wird als **Unterklasse** (***Subclass, derived class***) bezeichnet, die Klasse, auf der aufgebaut wird, heißt **Oberklasse** (**Basisklasse**, ***Superclass***).
- Im Rahmen der Vererbung erbt die Unterklasse die Methoden und Variablen der Oberklasse. Außerdem kann die Unterklasse Variablen und Methoden hinzufügen oder umdefinieren. Prinzipiell könnte eine Unterklasse Methoden auch weglassen - die meisten Programmiersprachen sehen dies aber nicht vor.

Beispiel:

- In dem Beispiel gibt es eine allgemeine Klasse window, die all die Dinge enthält, die allen Fenstern gemeinsam sind (z. B. Position, Größe, Rahmen sowie Methoden zum Verschieben, Größe verändern, Auf- und Zumachen).
- Die Unterklasse dialog enthält zusätzliche Möglichkeiten um Dialog mit einem Benutzer zu führen (sie weiß z. B. wenn der Fokus auf ihr liegt, kennt das Eingabe-Device, etc.)
- Ein yes/no-Dialog gibt nur die Möglichkeit, Buttons anzuklicken, während ein File-Dialog die Struktur eines Dateibaums anzeigen kann und die Möglichkeit zur Texteingabe anbietet.
- Letzlich sind aber alle Dialoge auch Fenster, haben eine Größe, Position, etc.

Vorteile:

- Die Implementierung der Oberklasse wird nicht verändert
- Gleiche Funktionalität muss nicht mehrfach implementiert werden
- Veränderungen der Oberklasse wirken auf alle Unterklassen
- Beziehungen zwischen Klassen werden dokumentiert

# 5 Vererbung (2)

## ★ Begriffe

### Unterklasse

- Die neu entstehende Klassen wird als **Unterklasse** (***subclass, derived class***) bezeichnet.

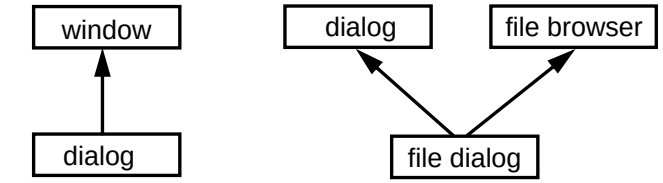
### Oberklasse

- Die Klasse, von der die Unterklasse abgeleitet wurde, heißt **Oberklasse** oder auch **Basisklasse** (***superclass, base class***).

### Einfache Vererbung

- Bei einfacher Vererbung (***single inheritance***) hat eine Unterklasse nur genau eine Oberklasse.
- Java sieht z. B. nur einfache Vererbung bei Klassen vor.

### Mehrfache Vererbung




einfache Vererbung mehrfache Vererbung

- Bei mehrfacher Vererbung (***multiple inheritance***) kann eine Klasse von mehreren anderen Klassen direkt erben
  ➔ die Vererbungsbeziehungen bilden einen gerichteten Graph
- Probleme:
  - Namensgleichheit bei Komponenten der verschiedenen Basisklassen
  - Vererbung von Klassen auf unterschiedlichen Pfaden
- Anwendung:
  - weniger zur Wiederverwendung von Implementierung
  - primär zur Herstellung von Typkonformität (siehe Kapitel über Typisierung)
- C++ erlaubt z. B. 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

    je nachdem was man zuerst hat
    - Generalisierung: wenn man bei der Problemanalyse die konkreteren Dinge zuerst gefunden hat und dann gemeinsame Eigenschaften entdeckt und diese zusammenfassen möchte
    - Spezialisierung: wenn es unterschiedliche Ausprägungen "ähnlicher" Dinge gibt
- 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)

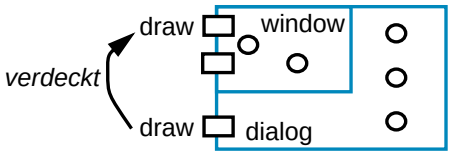





# 5 Vererbung (5)

★ **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?
    - Die reimplementierte Methode der Unterklasse will evtl. in ihrer Implementierung auf die "Originalmethode" der Oberklasse zurückgreifen. Zur Referenzierung von Methoden der Oberklasse wird häufig ein spezielles Schlüsselwort — z. B. *super* — verwendet. In C++ wird dies durch den Namen der Oberklasse und den Scope-Operator :: erreicht.

# 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
    - Daten und Methoden der Kategorien *public* und *protected* haben diese Zugriffseigenschaften auch in der Unterklasse, d. h.
      - auf *public*-Daten und -Methoden von Instanzen der Unterklasse, die aus der Basisklasse übernommen wurden, kann aus Funktionen und Methoden anderer Klassen zugegriffen werden
        - ➔ die Unterklasse stellt einen **Subtyp** des Typs der Basisklasse dar (vgl. Abschnitt über Typisierung)!
      - wird diese Unterklasse im Rahmen weiterer Vererbung wieder als Basisklasse genutzt, können Methoden der weiteren Unterklasse ("Unter-Unterklasse") auf *public*- und *protected*-Daten und -Methoden der ursprünglichen Basisklasse zugreifen, soweit diese in der ersten Unterklasse nicht überlagert wurden.
  - ➤ private-Basisklasse
    - ➔ die *Schnittstelle* der Basisklasse wird *nicht* vererbt
    - Daten und Methoden der Kategorien *public* und *protected* werden mit der Zugriffseigenschaft *private* in die Unterklasse übernommen, d. h.
      - *public*-Daten und -Methoden der Basisklasse sind bei Instanzen der Unterklasse *private* und damit nicht außerhalb sichtbar.
      - Dadurch sind alle Methoden der Basisklasse an Instanzen der Unterklasse nicht aufrufbar, es sei denn, sie wurden explizit für die Unterklasse neu definiert (eine solche Methode der Unterklasse hat dabei auch die Möglichkeit, auf die entsprechende Methode gleichen Names der Basis-Klasse zuzugreifen, da für sie die *private*-Einschränkung ja nicht gilt).
        - ➔ der Typ der Unterklasse ist mit dem Typ der Basisklasse nicht in jedem Fall verträglich, ist also **kein Subtyp** (vgl. Abschnitt über Typisierung)!
      - *public*- und *protected*-Daten und -Methoden der Basisklasse sind für Unterklassen der Unterklasse nicht sichtbar.
- *private*-Daten und -Methoden der Basisklasse nicht sichtbar






# 7 Dynamisches Binden

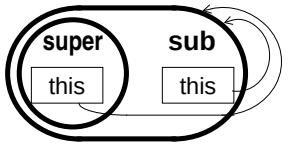
- Entscheidung welche Methode ausgeführt wird erfolgt zur Laufzeit (dynamisch)

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

Aufrufe virtueller Methoden in C++ werden immer dynamisch gebunden (*late binding*)

- bei Zeiger auf Objekt wird Methode durch Instanz bestimmt
  - Wird über einen Objektzeiger eine Methode aufgerufen, so wird die Methode des Objekts, auf das der Zeiger gerade zeigt, ausgeführt.
  - Ist es ein Objekt einer Unterklasse und wurde eine Methode der Basisklasse in der Unterklasse umdefiniert, so wird der in Unterklasse implementierte Code ausgeführt.
- Wird an einen Zeiger auf ein Objekt einer Basisklasse eine Instanz einer Unterklasse zugewiesen, so sind nach wie vor nur die in der Basisklasse deklarierten Methoden aufrufbar
  - in der Unterklasse neu hinzugekommene Methoden sind im Typ des Zeigers (= Typ der Basisklasse) nicht angegeben und daher nicht bekannt.

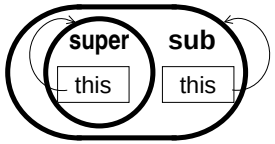
- Gilt auch, wenn ein Objekt eine Methode an sich selbst aufruft!
  - ◆ Beispiel:
    - **`move()`** ruft am Ende **`display()`** auf, um das Fenster neu zu zeichnen
    - **`BorderedWindow`** erbt **`move()`** von **`Window`**
    - ein Aufruf von **`move()`** an einer Instanz von **`BorderedWindow`**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




Ohne dynamisches Binden entsteht kein konsistentes Objekt der Unterklasse. Methoden der Oberklasse würden nach wie vor nur Implementierungen der Oberklasse aufrufen, selbst wenn die Unterklasse eine Reimplementierung hat, während reimplementierte Methoden der Unterklasse, die reimplementierten Methoden aufrufen würden.

Beispiel:

```
class ober {
        void m1() { … m2(); …}
        void m2() {… }
        void m3() {… m1(); m2(); }
}
```

```
class unter : public ober
{
        void m2() {… }
        void m3() {… m2(); m1(); }
}
```

Bei einem Aufuf der Methode m3 eines Oberklassenobjekts wird zweimal ober::m2 aufgerufen - einmal indirekt über m1, einmal direkt aus m3.

Bei einem Aufruf von m3 an einem Unterklassenobjekt würde ohne dynamisches Binden aus der reimplementierten Methode zuerst das reimplementierte unter::m2 direkt aufgerufen werden, dann würde das geerbte ober::m1 aufgerufen und von dort das "alte" ober::m2 - weil die reimplementierte Fassung von m2 aus der geerbten Methode m1 nicht gefunden würde.






# 8 Statisches Binden

- Entscheidung welche Implementierung einer Methode genommen wird fällt zur Übersetzungszeit

  bei Zeiger auf Objekt wird auszuführende Methode durch Typ des Zeigers bestimmt:
  Wird über einen Zeiger auf ein Objekt einer Basisklasse eine Methode aufgerufen, so wird immer der in der Basisklasse definierte Code ausgeführt, unabhängig von dem Objekt, auf das der Zeiger zur Ausführungszeit gerade zeigt.

- In C++ werden nur “virtual” -Methoden dynamisch gebunden
  - ◆ alle anderen Methoden werden generell statisch gebunden
- In Java werden alle Methoden dynamisch gebunden
  - Methoden in Java entsprechen im wesentlichen den virtuellen Methoden in C++
  - ◆ statisches Binden kann durch das Schlüsselwort **final** in der Methodedeklaration erzwungen werden
    - Der Hauptgrund für statisches Binden in C++ ist Effizienz. Mit final-Methoden wurde in Java ein Mechanismus geschaffen, mit dem man in Einzelfällen ebenfalls Effizienz gegenüber "normalen", dynamisch gebundenen Aufrufen gewinnen kann.
  - ◆ solche Methoden können in Unterklassen nicht reimplementiert werden

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

  - ➔ Compiler kann statisch binden
  - Im Gegensatz zu nicht-virtuellen Methoden von C++ kann hier in der Klassenhierarchie in den oberen Ebenen noch dynamisch gebunden werden. Erst ab der Subklasse, die die Methode final deklariert hat abwärts ist die Methode statisch gebunden.
  - Während bei C++ auch eine nicht-virtuelle Methode in einer Subklasse noch reimplementiert werden kann (welche Methode tatsächlich aufgerufen wird hängt dann ja vom Typ des Zeigers ab, über den aufgerufen wird), verhindert der Java-Compiler, daß eine final-Methode nochmals reimplementiert wird. Die Aufrufsemantik kann sich damit nie von dynamisch gebundenen Methoden unterscheiden!

# C.6 Fundamentale Konzepte des objektorientierten Paradigmas

Eine Reihe programmiersprachlicher Konzepte, die unabhängig von den Ideen zur objektorientierten Programmierung entstanden sind, sind für das objektorientierte Paradigma von grundlegender Bedeutung:

- Abstraktion
  - ◆ Kapselung
  - ◆ Datenabstraktion in OOP
- Modularisierung
- Hierarchie

Weitere allgemeine programmiersprachliche Konzepte ergänzen viele objektorientierte Programmierumgebungen. Teilweise erhielten solche Konzepte auch erst durch die Verbindung mit dem objektorientierten Paradigma praktische Bedeutung.

- Typisierung
  - ◆ Typhierarchie
  - ◆ Polymorphismus
  - ◆ Generizität
- Nebenläufigkeit
- Persistenz






## 1 Abstraktion

Grundlegendes Konzept zur Lösung komplexer Probleme

- Für Zusammenhänge relevante Aspekte hervorheben
- Details vernachlässigen
  - Abstraktion hilft dem Software-Entwickler, sich auf das zu lösende Problem zu konzentrieren und sich den Blick auf die eigentlichen Probleme nicht durch Implementierungsdetails zu verstellen!

→ Objektorientierung

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

  } Sicht von aussen

  ➥ contract model: Sicht von aussen = Vertrag mit anderen Objekten
  - Signatur: Beschreibung der Objektschnittstellen — Methoden, Parameter, Ergebnisse
  - Semantik: Auswirkung von Methodenaufrufen

    Eine Klasse realisiert die Implementierung einer Abstraktion. Für den Anwender dieser Klasse — bzw. ihrer Instanzen — ist dabei lediglich der für ihn sichtbare Teil interessant.
    - – Sichtbar sind zum einen die Schnittstellen (=Methoden, Parameter und Resulate), zum anderen die Auswirkungen von Operationen auf das zukünftige Verhalten des Objekts.
    - – Änderungen des Objektzustands, die über die Schnittstellen nicht beobachtet werden können (z. B. ein Stack-Objekt fordert dynamisch Speicher nach, um weitere Daten speichern zu können und gibt ihn auch selbsttätig wieder frei, wenn er nicht mehr benötigt wird), sind für den Benutzer des Objekts irrelevant.
- unwichtig:
  - ➤ Implementierung eines Objekts
    - – Instanzvariablen
    - – Implementierung von Methoden
  - Die Implementierung eines Objekts sollte ausgewechselt werden können, solange sich die Schnittstelle und die Semantik der Operationen nicht ändert. Dafür ist es aber wichtig, daß der Benutzer keine Kenntnisse über Implementierungsdetails erhält, die für seine Anforderungen nicht erforderlich sind. Er könnte sonst Annahmen über das Verhalten des Objekts machen und sich auf diese Annahmen bei seiner Programmierung verlassen, obwohl solch ein Verhalten durch den Implementierer des Objekts nie zugesichert wurde.
- zuerst die Abstraktion beschreiben, dann Implementierung vornehmen
  - = zuerst Problemlösung spezifizieren, dann implementieren!

# 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

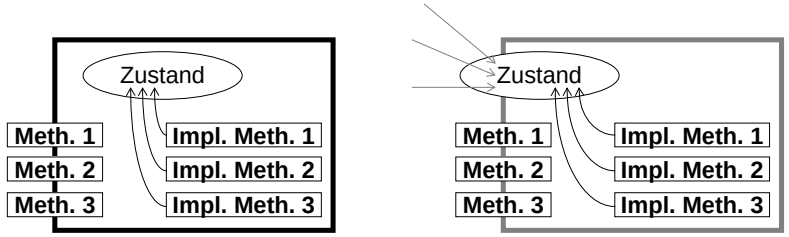
➔ Abstrakter Datentyp, Datenabstraktion






# 2 Datenabstraktion in OOP

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




Objekt als Implementierung eines ADT — Objekt ohne Datenabstraktion

- Viele Programmiersprachen erzwingen Datenabstraktion allerdings nicht (z. B. C++)
- Häufig ist ein Kompromiss zwischen klarer Strukturierung und Disziplin sowie Flexibilität und Effizienz erforderlich.
- Vor allem in der Systemprogrammierung ist Datenabstraktion in der Praxis nicht bis in die untersten Betriebssystemschichten durchzuhalten.
  - – Hinter einem Objekt kann sich "ein Stück Hardware" (z. B. ein Controller oder ein Prozessor) verbergen — Änderungen am Zustand solcher Objekte sind in manchen Fällen trotzdem von "außen" zu beobachten (z. B. weil nach einem Aufruf an ein Objekt, das den Treiber für eine serielle Schnittstelle realisiert, ein Signalpegel auf der Schnittstelle verändert wird und dadurch eine Telefonmodem-Verbindung abgebaut wird).

- Datenabstraktion in C++ und Java
  - ◆ Sichtbarkeitsbereiche (*Scope*-Regeln)
    - Kapselung durch *private*- und *protected*-Bereiche in Klassen
    - Objektschnittstellen im *public-Bereich* einer Klasse
    - *private* -Daten und -Methoden
      - – nur innerhalb von Methoden der gleichen Klasse sichtbar
    - *protected* -Daten und -Methoden (in Java: *private protected*)
      - – nur für Methoden der gleichen Klasse und in Methoden von Unterklassen sichtbar
    - *public* -Daten und -Methoden
      - – für alle Funktionen des Programms und für alle Methoden anderer Klassen sichtbar
        *public* -Methoden bilden die Schnittstelle der Klasse nach außen
        *public* -Daten erlauben eine Verletzung der Objektkapselung
        ➔ vermeiden!

# 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
  - ➔ enge Beziehung zwischen Modularität und Kapselung

  Die Unterstützung von Modularisierung in Programmiersprachen ist allerdings sehr unterschiedlich:
  - In C oder C++ sind Module lediglich getrennt übersetzbare Dateien — Schnittstellen zwischen Modulen können über Include-Dateien konsistent gehalten werden. Die Sprachen sehen allerdings keinerlei Mechanismen vor, die eine konsistente Benutzung der Schnittstellen wirklich garantieren.
  - Andere Sprachen (wie z. B. Modula) kontrollieren dagegen die Konsistenz der Modulschnittstellen direkt.
  - In Java gibt es ein eigenes Modularisierungskonzept: *Packages*
    - – Zusätzliche Schutzattribute: Klassen oder Methoden sind damit nur innerhalb des eigenen Paketes sichtbar
    - – Pakete spannen einen eigenen Namensraum auf
      - ➔ keine Namenskonflikte zwischen unabhängig entwickelten Softwarekomponenten möglich
    - – Das Laufzeitsystem garantiert die Einhaltung der Schutzattribute und die konsistente Nutzung der Modulschnittstellen
- *Structured Design*: Gruppierung von Unterprogrammen
  - Unterprogramme werden meist aufgrund ihrer Interaktionsbeziehungen auf Module verteilt.
- OOD: Gruppierung von Objekten und Klassen
  - Bei der Programmierung komplexer Softwaresysteme entsteht eine große Anzahl von Klassen
    - ➔ viele kleine Einheiten mit eigenen Schnittstellen
    - ➔ Beziehungen zwischen den Einheiten schwer überblickbar
  - Aufgabe der Modularisierung im Rahmen von objektorientiertem Softwareentwurf ist damit, eine Gruppierung von Klassen auf Basis der Problemstruktur vorzunehmen. Diese kann sich von der Struktur, die durch die Interaktionsbeziehungen von Unterprogrammen entstehen würde stark unterscheiden!






# 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*)
    - Vererbung
    - Delegation
  - ➤ Objektstruktur: Aggregation → "**Teil-von**"-Hierarchie (*part-of*)
    - Aggregation — ein Objekt ist aus mehreren Unterobjekten zusammengesetzt
      - – Beispiel: Objekt Auto besteht aus Unterobjekten Motor, Getriebe, Lenkung, etc.

# 5 Typisierung

- Typkonzept baut auf ADT-Theorie auf

  Definitionen aus [Boo94]:

  - Ein Typ ist eine genaue Charakterisierung der gemeinsamen Eigenschaften (bezüglich Struktur und Verhalten) einer Menge von Einheiten
  - Typisierung ist das Erzwingen der Klasse eines Objekts, so dass Objekte unterschiedlicher Typen nicht oder zumindest nur in sehr eingeschränkter Weise untereinander ausgetauscht werden können

  Bei Objekten wird der Typ durch die Signatur und die Semantik der Methodenaufrufe bestimmt.

  Die meisten Programmiersprachen ermöglichen dem Programierer allerdings nur die Formulierung der syntaktischen Eigenschaften eines Typs.

  – Bei vielen objektorientierten Sprachen (z. B. C++, aber nicht Java) erfolgt dies durch die Beschreibung einer Klassenschnittstelle.

  ➔ Die Beschreibung einer Klasse definiert gleichzeitig einen Typ.

- Typisierung ermöglicht die Überprüfung von Ausdrücken auf Typ-Kompatibilität zur Übersetzungszeit

  ➔ Vermeidung von Fehlern

  - Zuweisungen unterschiedlicher Typen oder die Übergabe falscher Typen als Parameter von Funktions- bzw. Methodenaufrufen kann zur Übersetzungszeit festgestellt werden. Auf diese Weise lassen sich viele Fehler von vorneherein vermeiden.

- Strenge Typisierung
  Konformität aller Typen in einem Ausdruck wird garantiert

  ➔ Statische Typisierung
  Konformität wird komplett zur Übersetzungszeit überprüft

  ➤ Einschränkung der Flexibilität

  - kompatible Typen

    Häufig kann der Compiler nicht feststellen, dass zwei unterschiedliche Typen kompatibel sind und damit in einem Ausdruck (z. B. einer Zuweisung) problemlos zusammen verwendbar sind.

  - dynamisches Binden

    Wenn erst zur Laufzeit festgelegt werden soll, welches Objekt (oder auch welche Funktion in nicht-OOP) über eine Variable erreichbar ist, kann die Typ-Verträglichkeit nicht generell zur Übersetzungszeit überprüft werden.

  ➔ mehr Flexibilität durch Polymorphismus und Generizität

  – Polymorphismus erlaubt neben der Verwendung des passenden Typs auch andere, konforme Typen.

  – Generizität ermöglicht eine Parametrierung mit Typen. Dadurch können z. B. Klassen, die für verschiedene Typen einsetzbar wären, aufgrund strenger Typisierung aber nicht flexibel verwendet werden können, durch Typ-Parameter für die Verwendung in Zusammenhang mit einem Typ eingestellt werden.






# 6 Typhierarchie

- Häufig 1:1-Relation zwischen Klassen und Typen — aber nicht notwendig

  ➤ mehrere Klassen können einen Typ implementieren

  - Beispielsweise können unterschiedliche Klassen einen Datentyp "Stack" unterschiedlich implementieren (Daten werden in einem Feld oder in einer verketteten Liste verwaltet), aber die gleichen Methodenaufrufe (bezüglich Signatur und auch Semantik) anbieten.

  ➤ eine Klasse kann unterschiedliche Typen implementieren

  - Eine Klasse kann beispielsweise besondere Methodenaufrufe zur Einstellung von Parametern oder Debug-Möglichkeiten besitzen, deren Verwendung in Teilen einer Anwendungsimplementierung aber nicht erlaubt sein darf (weil die Klasse möglicherweise später durch eine Implementierung ohne diese Methoden ersetzt werden soll). Wenn man in den Teilen der Anwendung einen Typ für die Klasse bekanntgibt, der die besonderen Methodenaufrufe nicht enthält, garantiert das Typsystem, dass sie auch nicht verwendet werden. In den Teilen der Anwendung, die die besonderen Methodenaufrufe benötigen, wird ein anderer Typ für die Klasse deklariert, der die Methoden enthält.

- Hierarchie bei Klassen: Oberklasse ← Unterklasse

  ◆ Ziel: Wiederverwendung von Implementierung

  ◆ Unterklasse nicht notwenigerweise typ-konform zu Oberklasse

  - In den meisten Programmiersprachen (auch in C++ und Eiffel, nicht aber in Java) ist es möglich, Unterklassen zu programmieren, die nicht typ-konform zur Oberklasse sind.
  - Unter dem Gesichtspunkt, durch eine Vererbung bei Klassen Programmcode wiederzuverwenden ist dies eigentlich unproblematisch. Da viele Programmiersprachen aber durch Klassendeklarationen gleichzeitig Typdeklarationen vornehmen, ist es wichtig, unterscheiden zu können, wo durch Klassen-Vererbung ein konformer Untertyp entsteht und wo nicht.

- Hierarchie bei Typen: Obertyp ← Untertyp

  ◆ Ziele: – Verhaltens-Vererbung
  – Beschreibung **konformer** Typen (➔ Polymorphismus)

  - Durch eine Typhierarchie soll ausschließlich festgelegt werden, welche Typen zu welchen anderen Typen konform sind.

- Typvererbung (*Subtyping*) als Mechanismus zur Ableitung von Typen

  - Analog zur Vererbung bei Klassen: der Untertyp erbt von einem Obertyp
  - Unterschied zur Vererbung bei Klassen: der Untertyp ist auf jeden Fall konform zum Obertyp — d. h. er kann zusätzliche Methoden enthalten, enthält aber bezüglich Signatur und Semantik alle Methoden des Obertyps.
  - Mehrfachvererbung bei Typen (ein Subtyp ist konform zu mehreren Obertypen) ist im Gegensatz zu Mehrfachvererbung bei Klassen absolut unproblematisch, da keine Implementierung verebt wird!

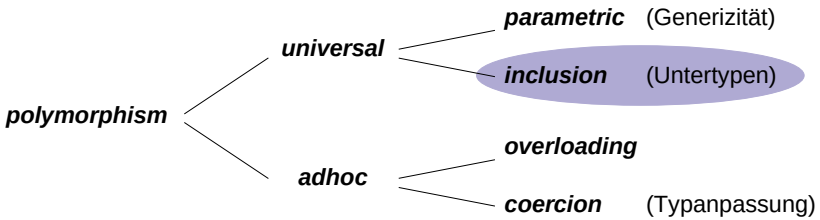
  ➔ Zusammenhänge zwischen Typen werden übersichtlich

  – Identifikation konformer Typen

  - Durch die Typhierarchie ist festgelegt, welche (Unter-)Typen statt eines Typs in einem Ausdruck verwendet werden dürfen.

# 7 Polymorphismus

- die Fähigkeit, verschiedene Formen anzunehmen [Mey88]
  - ➤ 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]




*universal polymorphism:*

- funktioniert prinzipiell mit beliebig vielen Typen (alle Typen haben aber eine gemeinsame Grundstruktur — sind beispielsweise Subtypen eines gemeinsamen Obertyps, z. B. Zahlen in verschiedenen Darstellungen und Ausprägungen).

*parametric polymorphism:*

- Eine polymorphe Funktion besitzt einen expliziten Parameter, in dem der Typ des Parameters für jeden Aufruf mit übergeben wird (spezielle Ausprägung dieses Konzepts in Adas *generic functions* zu finden).

***inclusion polymorphism**:*

- Bei Subtyping: ein Untertyp kann jederzeit statt des Typs verwendet werden.
  **In objektorientierten Sprachen von großer Bedeutung**: häufig mit Vererbung gekoppelt — wenn eine Unterklasse einen Untertyp implementiert, können Instanzen der Unterklasse statt Instanzen der Basisklasse verwendet werden.

*ad hoc polymorphism:*

- funktioniert nur auf einer endlichen Menge — möglicherweise völlig voneinander unabhängiger Typen.

*overloading:*

- Mehrere verschiedene Funktionen werden unter dem gleichen Namen definiert und es wird anhand des Aufruf-Kontexts (Anzahl und Typ der Aufrufparameter) entschieden, welche Implementierung aufgerufen wird. (Beispiele siehe C++)

*coercion:*

- Vor dem Aufruf der Funktion werden Parameter, deren Typ nicht mit der Funktionsdefinition übereinstimmt, automatisch in den passenden Typ konvertiert.
  (Beispiel für *Coercion*: in C wird bei der Addition eines int- und eines real-Wertes automatisch eine Konvertierung beider Operanden nach double durchgeführt).






# 8 Polymorphismus in C++

Einige Konzepte in C++ können zur Realisierung von Polymorphismus eingesetzt werden. Dabei sind drei Kategorien zu unterscheiden:

- *overloading polymorphism*
  - ➤ Function-name overloading
  - ➤ Operator overloading
    - Funktionen oder Operatoren eines Namens können für unterschiedliche Parameter- bzw. Operandentypen definiert werden.
- *inclusion polymorphism*
  - ➤ public Vererbung
    - Da in C++ Subklassen gleichzeitig Subtypen des Datentyps der Basisklasse bilden, sind sie typkonform zur Basisklasse. Instanzen der Subklasse können daher statt Instanzen der Basisklasse verwendet werden. In der Subklasse können dabei Methoden der Basisklasse anders implementiert sein.
- *coercion polymorphism*
  - ➤ Cast-Operatoren
    - Für die Verwendung eines "unpassenden" Typs als Parameter oder in einem Ausdruck können für Einzelfälle Regeln beschrieben werden, wie der unpassende Typ in einen passenden Typ konvertiert wird. Diese Konvertierung erfolgt dann automatisch.

# 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();
...
```

- Der Aufruf **`ptr->draw()`** bewirkt den Aufruf der **`draw`**-Methode der Subklasse **`circle`**, obwohl **`ptr`** ein Zeiger auf ein Objekt der Klasse **`geo_obj`** ist!
- Wäre **`draw`** in **`geo_obj`** nicht als virtual deklariert worden, hätte der Aufruf **`ptr->draw()`** die Methode der Klasse **`geo_obj`** angesprochen, obwohl eine Instanz der Unterklasse **`circle`** an **`ptr`** gebunden war.
- Die **`draw`**-Methoden der Unterklasse und auch die **`draw`**-Methode der Oberklasse **`geo_obj`** müssen in obigem Beispiel natürlich noch außerhalb der Klassen definiert werden.
- Auf die Definition der **`draw`**-Methode der Oberklasse könnte auch verzichtet werden, **`geo_obj`** wäre dann eine abstrakte Klasse (siehe Abschnitt über abstrakte Klassen).
  - es könnten dann allerdings keine Instanzen von **`geo_obj`** erzeugt werden, da die Klasse nicht vollständig definiert ist
  - **`geo_obj`** wäre dann nur als Basisklasse im Rahmen von Vererbung zu gebrauchen — in obigem Beispiel durchaus sinnvoll, weil eine **`draw`**-Methode, die nicht näher beschriebene geometrische Objekte zeichnen kann, wohl ohnehin keinen Sinn macht.

Virtuelle Methoden einer Subklasse, die eine virtuelle Methode der Basiklasse neu implementieren, sind automatisch wieder virtuell (z. B. bei weiterer Vererbung).

→ **`void draw();`** in der Klasse **`circle`** ist damit korrekt, es muß nicht **`virtual`** davor angegeben werden!

# 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() { ... }
}
```

- Die Klasse **`geo_obj`** beschreibt lediglich die Signatur von geometrischen Objekten, enthält aber keinerlei Implementierungen.
- Unterklassen von **`geo_obj`** stellen dann konkrete Implementierungen verschiedener geometrischer Objekte dar, die alle konform zu dem Datentyp **`geo_obj`** sind (und damit z. B. an einen Zeiger auf **`geo_obj`** zuweisbar sind).

- Grundsätzlich können auch einzelne (virtuelle) Methoden, die für alle Subklassen gleich sein sollen, in der abstrakten Basisklasse definiert werden und andere, die erst in den Subklassen implementiert werden sollen, als *pure virtual functions* angegeben werden. Die Basisklasse ist dann nach wie vor eine abstrakte Klasse, da sie nicht vollständig implementiert ist, sie entspricht aber nicht mehr einer reinen Typdefinition, sondern ist eine Mischung aus Typdefinition und -implementierung.
- Genauso können Subklassen von abstrakten Klassen auf die Definition einer *pure virtual function* verzichten (sie müssen deren Deklaration aber explizit wiederholen!). Solche Subklassen sind dann nach wie vor abstrakte Klassen und können nur im Rahmen von weiterer Vererbung eingesetzt werden.






# 10 Typen und Java: Interfaces

- 2 Möglichkeiten einen Typ zu deklarieren:
  - ◆ im Rahmen einer Klassendefinition
    - Java kennt auch abstrakte Klassen wie in C++
    - Nachteile der impliziten Typ-Deklarationen durch Klassen:
      - Es ist nur einfache Vererbung möglich. Ein durch eine Klasse definierter Typ kann damit nur an von dieser Klasse abgeleitete Unterklassen vererbt werden.
      - Es gibt keine Möglichkeiten, andere, unabhängig definierte, aber typ-kompatible Klassen zu solch einem Typ konform zu erklären.

      ➔ Klassenvererbung führt automatisch zu Typvererbung
    - Es ist in Java nicht möglich, Subklassen zu bilden, die nicht typ-konform zur Oberklasse sind.

  - ◆ durch eine Interface-Deklaration

    ➔ separate Typbeschreibung
    - Die Typ-Beschreibung erfolgt getrennt als `interface`
    - Danach kann im Rahmen einer Klassendefinition angegeben werden, daß die Klasse den Typ implementiert.
    - Der Compiler überprüft, ob in der Klassenimplementierung auch tatsächlich die Schnittstelle des Typs korrekt angegeben ist.

- 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
  - Während bei Klassenvererbung nur Einfachvererbung möglich ist, ist auf Typen auch Mehrfachvererbung zugelassen
    - d. h. ein Typ kann konform zu verschiedenen Obertypen sein
  - Da nur Schnittstellenkonformität deklariert wird, aber keine Implementierung übernommen wird, treten die Konfliktprobleme der Mehrfachverbung bei Klassen natürlich nicht auf.
- eine Klasse kann mehrere Typen implementieren
  - Bei einer Klassendefinition können mehrere, ggf. auch unterschiedliche Schnittstellen angegeben werden, die von der Klasse implementiert werden.
- Typkonformität ist transitiv
  - Wenn eine Klasse einen Typ implementiert, dann implementieren automatisch auch alle Subklassen davon diesen Typ.
- Exceptions sind auch Bestandteil der Typ-Schnittstelle
  - Im Gegensatz zu C++ können in Java auch die Exceptions, die eine Methode erzeugt in der Typ-Schnittstelle angegeben werden.
  - Ein Subtyp darf, um konform zu sein, nur die Exceptions des Basistyps (ggf. natürlich weniger Exceptions) bzw. konforme Exceptions erzeugen
    - Exceptions werden in Java durch Klassen repräsentiert. Eine konforme Exception ist damit eine Unterklasse bzw. eine zum Exception-Typ ebenfalls konforme Klasse.
- Beispiele:

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

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

  - Der Typ `Streamable` ist von den beiden Typen `FileIO` und `Printable` abgeleitet.
  - Die Methode Test kann eine Exception `TestException` erzeugen — diese Tatsache ist Bestandteil des Typs `Streamable`.
  - Die Klasse Test implementiert die beiden Typen `Streamable` und `Testinterface`






# 11 Generizität (*Genericity*)

- Möglichkeit, den Typ von programmiersprachlichen Einheiten durch Parameter festzulegen

  Beispiele:
  - generische Funktionsargumente (ein Funktionsparameter bestimmt den Typ anderer Parameter)
  - generische (parametrierbare) Klassen
- 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 [Mey86]

  **Aber:** eigentlich werden dadurch die Begriffswelten von Typ und Klasse (wie so oft) vermischt: Generizität ist ein Konzept, um mehr Flexibilität im Bereich Typisierung zu erreichen, während Vererbung Klassenhierarchien erzeugt!
- 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
  - Vor allem wenn Klassen in Standardbibliotheken bereitgestellt werden sollen, gibt es viele Fälle, in denen von vorneherein nicht festgelegt werden kann, für welche Typen solch eine Klasse gebaut werden soll. Typische Beispiele sind Container-Klassen wie Listen, Vektoren oder assoziative Felder.
- Grundsätzliche gibt es zwei Möglichkeiten, die Konstruktion solcher Klassen in einer Sprache zu unterstützen:

◆ dynamische Typprüfung zur Laufzeit

- Die betreffenden Typen werden zur Übersetzungszeit nicht festgelegt und durch dynamische Typprüfung zur Laufzeit wird eine konsistente Verwendung von Parametern und Instanzvariablen überprüft. Insbesondere in nicht-getypten Sprachen (wie Smalltalk) wird dieser Weg gewählt. Die Programmierung wird dadurch einfacher, vor allem sind alle Klassen sehr flexibel einsetzbar. Zur Laufzeit entsteht aber beträchtlicher Mehraufwand durch die dynamischen Typprüfungen.

◆ statische Typprüfung zur Übersetzungszeit + parametrierbare Klassen

- Die korrekte Verwendung von Typen bei Parameterübergabe und bei Zuweisungen wird zur Übersetzungszeit komplett überprüft. Um Klassen definieren zu können, ohne sich auf alle in der Klassenimplementierung verwendeten Typen von vorneherein festlegen zu müssen, werden parametrierbare Klassen eingeführt: Die Klassendefinition enthält Parameter, die in der Implementierung statt eines Typs eingesetzt werden können.

- 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]); }
};
```

- In der Klassendefinition können neben einem oder mehreren Typparametern auch andere Parameter auftreten. In dem Beispiel könnte man neben dem Typ **T** beispielsweise auch die Stackgröße als Template-Parameter angeben. Man könnte dann ein Objekt "Integer-Stack mit 10 Elementen" instantiieren. Dieses Objekt hätte aber einen anderen Typ als ein "Integer-Stack mit 20 Elementen". Wird die Größe dagegen im Konstruktor angegeben, so instantiiert man Objekte vom Typ "Integer-Stack" und übergibt dem Konstruktor die gewünschte Größe. Der Typ solcher Objekte (auch unterschiedlicher Größe) ist dann identisch.






# 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

- es wird keine parametrisierte Klasse erzeugt, von der Instanzen gebildet werden, sondern der Bytecode der Klasse liegt zur Laufzeit nur einmal vor
- Type-Erasure: Typ-Variablen werden durch Object (bzw. bei Bounds durch den angegebenen Obertyp) ersetzt. Compiler fügt Casts in den Code ein, um zur Laufzeit Typ-Sicherheit zu gewährleisten.

➤ Bounds erlauben die Einschränkung von Typ-Variablen auf bestimmte Subtypen

- `T extends C & T1 & T2 ...`
  Typ T muss Untertyp der Klasse C bzw. der Interfaces T1 oder T2 ... sein

- 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

  zusätzliche Probleme:
  - Koordinierung
    - gegenseitiger Ausschluß
  - Synchronisation
    - warten auf Ergebnisse anderer Aktivitätsträger
  - Verteilung
    - Verteilen der auszuführenden Objekte auf die verfügbaren Prozessoren

- Granularität: Nebenläufigkeit / Objekte (Kapseln)
  - ➤ Nebenläufigkeit feiner granular
    - ➔ NL auch objektintern
      - mehrere Aktivitätsträger gleichzeitig in einem Objekt
        → objektinterne Koordinierung erforderlich
  - ➤ Nebenläufigkeit grober granular
    - ➔ NL nur objektextern
      - immer nur maximal ein Aktivitätsträger zu jedem Zeitpunkt in einem Objekt
        → keine Koordinierungsprobleme innerhalb eines Objekts, Koordinierung kann auf die Objektinteraktionen beschränkt werden

- Integration der Kontrolle von Nebenläufigkeit in objektorientierte Sprachen
  - ➤ orthogonale Sprachen
    - Orthogonale Sprachen enthalten keine Sprachkonstrukte zur Kontrolle von Nebenläufigkeit. Zur Koordinierung muß der Programmierer sprach-externe Mechanismen wie z. B. Semaphore einsetzen.
  - ➤ nicht-orthogonale Sprachen
    - In nicht-orthogonalen Sprachen können Objekte gegen nebenläufige Ausführung geschützt werden. Dies wird entweder automatisch für alle Objekte vorgenommen
      ➔ **uniforme Sprachen**
      oder es kann durch Sprachkonstrukte vom Programmierer festgelegt werden, ob ein Objekt geschützt werden soll
      ➔ **nicht-uniforme 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
  - Die auszuführenden Methode wird in einer speziellen Klassen, die ein Interface **`Runnable`** implementiert als Methode **`run`** implementiert.
    - Thread erzeugen = "run-Methoden"-Objekt instantiieren, Thread-Objekt instantiieren und dem Konstruktor das "run-Methoden"-Objekt übergeben
    - Thread starten = Methode **`start`** an Thread-Objekt aufrufen, → Methode **`run`** wird nebenläufig ausgeführt
  - Es wird eine Subklasse von Thread definiert, die die **`run`**-Methode redefiniert. Eine Instanz der Subklasse wird erzeugt und an ihr die Methode **`start`** aufgerufen.
    - Thread erzeugen = Objekt der Subklasse instantiieren
    - Thread starten = Methode **`start`** an dem Objekt aufrufen → Methode **`run`** wird nebenläufig ausgeführt

- 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
  - Methoden oder Code-Blöcke können `synchronized` deklariert werden
  - Alle synchronized-Methoden und Code-Blöcke eines Objekts (einer Instanz, nicht aller Instanzen einer Klassen gemeinsam!) bilden gemeinsam einen Monitor: d.h. es kann immer nur einThread zu einem Zeitpunkt in einer der Methoden bzw. Code-Blöcke aktiv sein.
  - Ein Objekt kann weitere Methoden besitzten, die nicht als synchronized deklariert wurden. Solche Methoden (z.B. Methoden, die nur auf dem Objektzustand lesen, aber keine kritischen, koordinierungsbedürftigen Operationen ausführen) können auch mehrfach parallel ausgeführt werden.

◆ 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
  - Innerhalb eines Monitors kann "wait" aufgerufen werden
    ⇒ Monitor wird freigegeben und Thread legt sich schlafen.
  - Jemand anderes kann innerhalb des Monitors "notify" oder "notifyAll" aufrufen
    ⇒ Einer der bzw. alle blockierten Threads werden aufgeweckt






# 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
    - Vorteil: flexiblere Handhabung von Monitoren im Gegensatz zu den in die Sprache integrierten Mechanismen
    - Nachteil: expliziter Umgang mit Koordinierungsmechansmen ggf. fehlerträchtiger
  - ➤ 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 **[ABC83]**

■ "aktive" Daten → Programmiersystem
- "Aktive" Daten, die *kurzfristig* im Speicher angelegt sind, werden mit den Hilfsmitteln des Programmiersystems (Sprache) bearbeitet.

■ "passive" Daten → DBMS oder Dateisystem
- "Passive" Daten, die die Programmausführung überleben sollen, werden einer Datenbank (DBMS) oder einem Dateisystem übergeben und mit dessen Funktionen verwaltet.

→ 2 Sichten auf Daten

→ Nachteile für den Anwender

➤ Konvertierung notwendig
- Programmieraufwand für die Konvertierung und den Transport zwischen passivem und aktivem Zustand (*scanf*/*printf*-Anweisungen)
(Atkinson schätzt i. a. ca. 30% des Codes!)

➤ Datenschutz des Programmiermodells geht verloren
- Schutzmechanismen des Programmiermodells (z. B. Typisierung) greifen nur, solange Daten aktiv sind - bei der ersten Wandlung kann alles verloren gehen.

★ 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
- Eine traditionelle Datei ist ein persistentes Objekt mit Methoden wie *read*, *write* und *seek*.

➤ Datenbanksysteme
➤ persistente Kommunikationsobjekte
- Beispielsweise ein Puffer-Objekt

■ Eigenschaften des objektorientierten Programmiermodells gehen automatisch auf Mechanismen über, die damit erstellt wurden
Beispiele:
- Vererbung zur Realisierung unterschiedlicher Dateiobjekte
- Kontrolle von Nebenläufigkeit

# C.7 Objektorientierte Software-Entwicklung

- Objektorientiertes Software -Engineering
- Phasen der Software-Entwicklung
  - Anforderungsanalye
  - Objektorientierte Analyse
  - Objektorientierter Entwurf
  - Implementierung
  - Test
- UML: The Unified Modeling Language — ein kurzer Abriss
  - Use-Case Diagramme
  - Klassendiagramme
  - Interaktionsdiagramme






# 1 Objektorientiertes Software-Engineering

- 1980 - 1990: Entwicklung von OO Programmiersprachen
  - Smalltalk
  - C++
  - Eiffel
  - LOOPS, Flavors
- ab 1990: Verbreitung von objektorientierten Software-Engineering Methoden
  - Sally Shlaer & Steve Mellor
  - Peter Coad & Ed Yourdon
  - Grady Booch
  - Jim Rumbaugh et al. (OMT: Object Modeling Technique)
  - Ivar Jacobson (OOSE, Objectory)
- 1995: Booch, Rumbaugh und Jacobsen beginnen die Entwicklung der UML

## 2 Warum objektorientiertes Software-Engineering?

- Optimaler Einsatz des objektorientierten Paradigmas
- Übergang auf OO Programmiersprachen ist einfach
- Das Problem ist der ***Paradigmen-Wechsel***
  - ➤ Vorteile der Objektorientierung hängen davon ab, wie ein Problem angegangen wird
  - ➤ sonst Gefahr, dass OO Konzepte falsch eingesetzt werden
    - ➥ C++
- Objektorientierte Software-Engineering-Methoden helfen, objektorientierte Konzepte richtig anzuwenden
  - ➤ richtige Bestimmung der Klassen
  - ➤ richtiger Einsatz von Vererbung
  - ➤ …






## 2 Warum objektorientiertes Software-Engineering? (2)

▲ Software häufig zu komplex um als Ganzes überblickbar zu sein

➥ Modelle um von Details des Systems zu abstrahieren

- ➤ verschiedene Blickrichtungen — z. B.
  - – Spezifikationssicht
  - – Entwurfssicht
- ➤ verschiedene Abstraktionsebenen
- ➤ Abstraktionen für Systemteile
- ➤ verschiedene Aspekte des Systems
  - – Anforderungen
  - – statische Aspekte — Struktur
  - – dynamische Aspekte — Verhalten

## 2 Warum objektorientiertes Software-Engineering? (3)

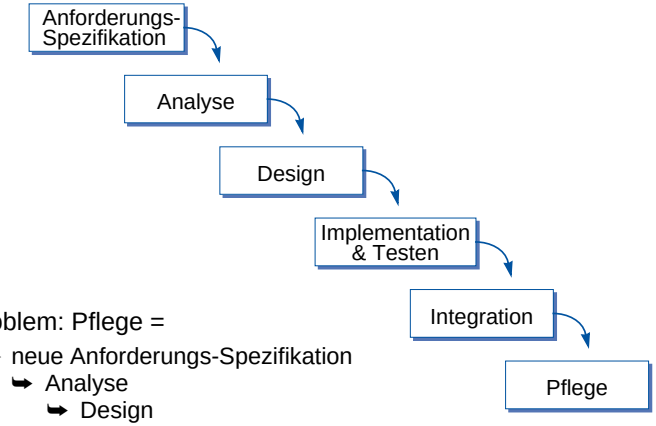
- ▲ Kontrolle des Entwicklungs-Prozesses
- ■ Unkontrollierte Entwicklung führt zu
  - ➤ nicht ausreichender Analyse der Anforderungen
    - ➥ Entwicklung stimmt nicht mit den Anforderungen des Auftraggebers überein
  - ➤ zu früher Start der Implementierung
    - ➥ Analyse und Entwurf sind noch nicht ausgereift
    - ➥ Fehler in Analyse und Entwurf werden erst später entdeckt
    - ➥ hohe Kosten für die daraus folgenden Änderungen
- ▲ Software-Engineering-Methoden geben Anleitungen
  - ➤ Komplexität heutiger Software zu beherrschen
  - ➤ Software wirtschaftlich zu entwickeln
  - ➤ die Software-Entwicklung zu einem kontrollierten, kontrollierbaren und kalkulierbaren Prozess zu machen






## 3 Phasen der Software-Entwicklung

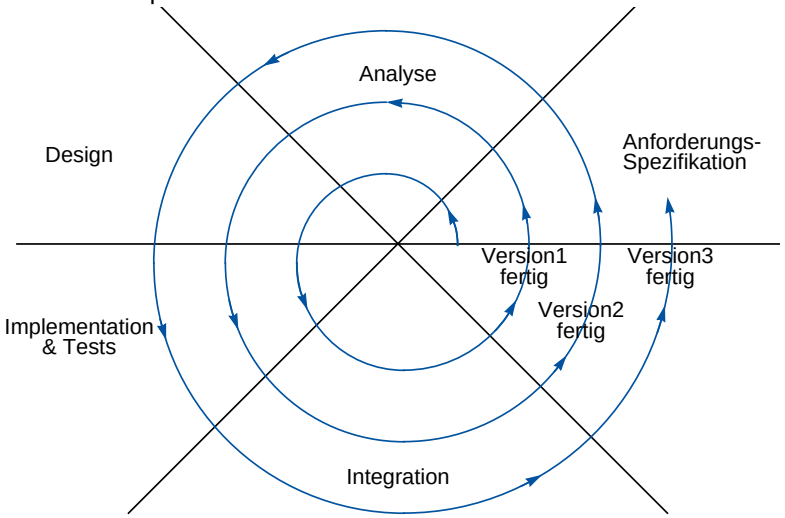
- ■ Traditionell: Wasserfall-Ansatz




- ■ Problem: Pflege =
  - ➤ neue Anforderungs-Spezifikation
    - ➥ Analyse
      - ➥ Design
        - ➥ neue Implementation
          - ➥ …

# 3 Phasen der Software-Entwicklung (2)

- Besser: Spiral-Modell



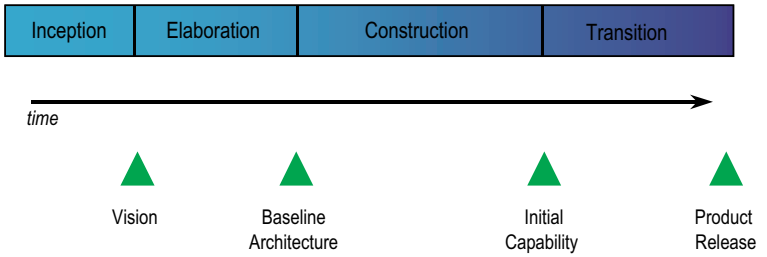






# 3 Phasen der Software-Entwicklung (3)

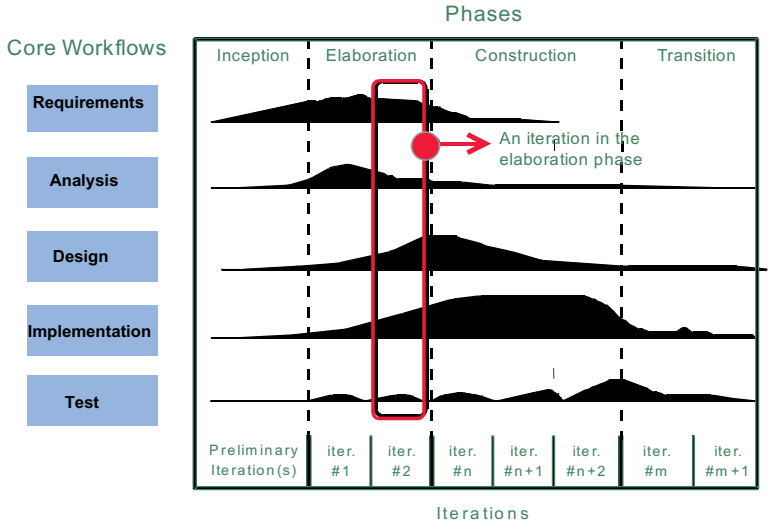
- Weiter verbessert: Iterativer Ansatz
- Vier Lebenszyklus-Phasen




  - Start (Inception): definiere Umfang des Systems, entwerfe Geschäftsplan
  - Ausarbeitung (Elaboration): plane Projekt, spezifiziere Eigenschaften, lege Grundlinien der Architektur fest
  - Konstruktion: baue das Produkt
  - Transition: übergebe das Produkt an seine Anwender

# 3 Phasen der Software-Entwicklung (4)

- Iterationen und Workflows









# C.8 Objektorientierte Analyse

- ▲ *Was* soll mein System tun?

- Basis: Analyse der "realen Welt"
  - ➤ Komponenten, Begriffe, Aufgaben
  - ➤ Anforderungen und Einschränkungen

- Abstraktion von unwichtigen Aspekten und Implementierungsdetails

- Abstraktion von Implementierungsdetails
  - – momentan unwichtig: wie implementiere ich Dinge?
  - + zu berücksichtigen: Aspekte der Implementierungsumgebung und Ausführungsplattform
    welche Mittel habe ich zur Verfügung?

# C.8 Objektorientierte Analyse (2)

## 1 Der Prozess

- Analysiere Anforderungen
- Beschreibe Use Cases
- Lege Begriffe des "problem domain" fest
- Finde Objekte

Anforderungs-Modell

- Organisiere Objekte
- Lege erste interne Strukturen der Objekte fest
- Beschreibe Intaktionen der Objekte
- Lege Operationen der Objekte fest

Analyse-Modell



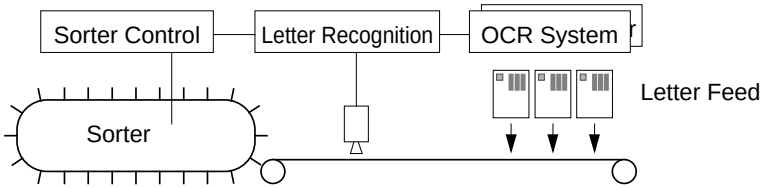



## 2 OOA — Anforderungsanalyse

- Fokus liegt auf dem "problem domain"
- Lege Ziele und Aufgaben fest
  - ➤ Basis für alternative Lösungen
  - ➤ Basis für Überprüfungen und Bewertungen
- Sicht des Kunden
- Performance-, Benutzer- and Architektur-bezogene Anforderungen
- Informelle Beschreibung

# 3 OOA — Beispiele einer Anforderungsanlyse

- Briefsortieranlage




- Problem domain: Postleitzahlenerkennung
- Aufgabe: erkenne PLZ auf einem Brief und melde an Sorter Control die Nummer des Ausgabefachs
- Anforderungen:
  - Erkennung darf max. 1 Minute dauern
  - eigene OCR-Rechner für Schrifterkennung
  - Statistiken über PLZ-Häufigkeit und Fehler



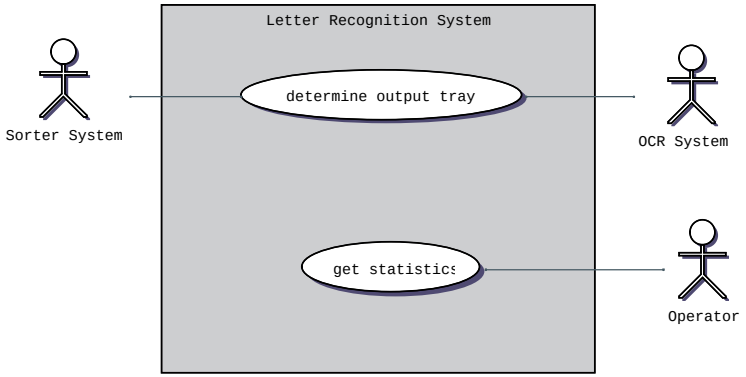



# 4 OOA — Use Cases

- Beschreibe Interaktionen zwischen "Benutzern" und System
- Beteiligte:
  - ➤ Aktor: Person oder eine andere Komponente des Systems **! in einer bestimmten Rolle**
  - ➤ Use case: "Dialog" zwischen Aktor und System um einen bestimmten Zweck zu erreichen
- weitere Festlegung der Anforderungsanalyse
- erster Schritt zur Modularisierung des Problems

# 4 OOA — Use Cases (2)

- Briefsortierer
  - ➤ Sorter Control fordert Nummer des Ausgabefachs an
  - ➤ Benutzer fragt Statistiken ab
- UML Diagram









# 5 OOA — Objekte finden

- Begriffe des "problem domain" identifizieren
  - ➤ nach Substantiven in der Terminology des "problem domain "suchen

BEISPIEL
- ➤ Sorter Control
- ➤ OCR
- ➤ Letter
- ➤ Camera

  - ➤ Strategie:
    - – Auftraggeber skizziert seine Sicht
    - – Systemanalyst schreibt Begriffe mit
- Ziel: Basis für die weitere Arbeit erstellen
  - ➤ NICHT das ganze System beschreiben

# 6 OOA — Objekte organisieren

- Modell strukturieren → Analysemodell

- verschiedene Kategorien von Objekten
  - Schnittstellenobjekte
  - Objekte, die Dinge repräsentieren (Entity objects)
  - Kontrollobjekte






# 6 OOA — Objekte organisieren (2)

**Schnittstellenobjekte**

- Repräsentieren Aktoren der Use Cases
- Startpukte für Aktivitäten im System
- Schnittstellen des Systems zur "Aussenwelt"

BEISPIEL
- ➤ Sorter Control
- ➤ Operator Panel
- ➤ OCR System

**Entity Objects**

- Repräsentieren den Systemzustand
  - ➤ sind verhältnismäßig langlebig
  - ➤ überdauern oft Ausführung eines Use Case

BEISPIEL
- ➤ Statistics
- ➤ Letter
- ➤ Picture

# 6 OOA — Objekte organisieren (3)

**Kontrollobjekte**

- Problem: eine Aufgabe oder Aktivität kann keinem der Objekte zugeordnet werden
  - ◆ Haupt-Fokus liegt auf dem Ablauf
  - ◆ solche Abläufe resultieren häufig direkt aus einem Use Case
    - ➤ definiere etwas, das für den Ablauf verantwortlich ist
      - ➥ erzeuge ein Objekt dafür

BEISPIEL

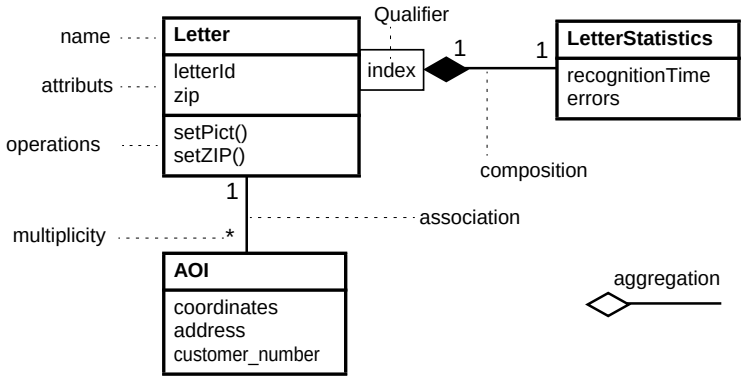
- ➤ Aufgabe: führe PLZ-Erkennung durch
  - – Bild aufnehmen
  - – Adressbereich suchen
  - – Auftrag an OCR-Rechner übergeben
  - – Ausgabefach an sorter control melden
    - ➥ Objekt *LetterController*





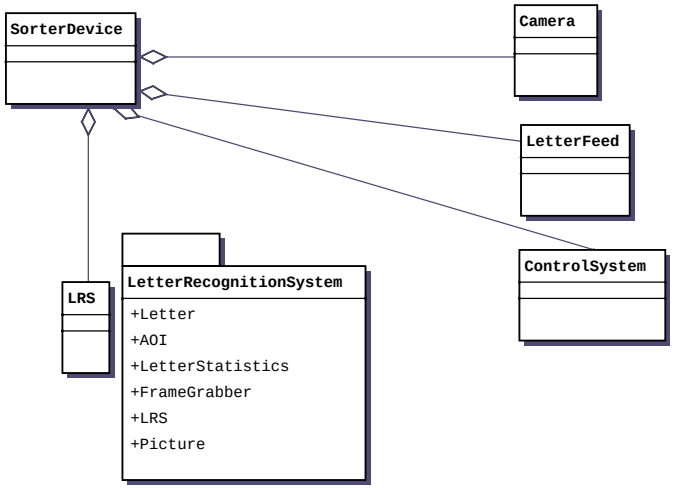

# 6 OOA — Objekte organisieren (4) UML-Notation: Klassendiagramme

- Klassen mit ihren Attributen (Variablen) und Operationen (Methoden)
- Beziehungen zwischen Klassen
- Beispiel:



# 6 OOA — Objekte organisieren (5)

- Briefsortierer — Systemüberblick: UML-Diagramm



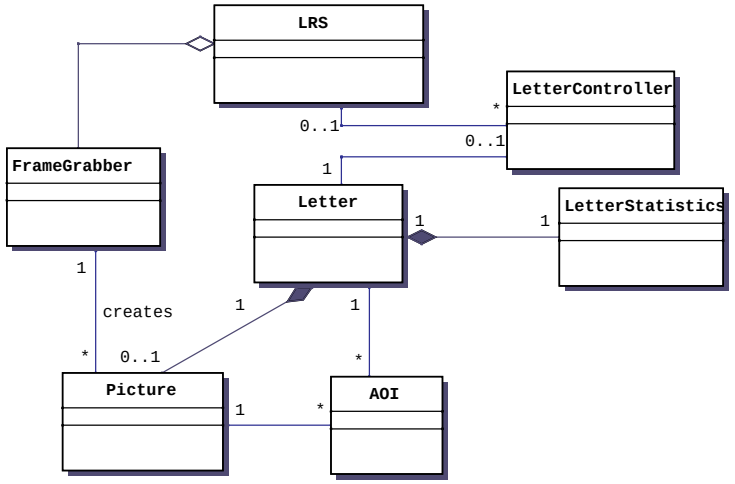






# 6 OOA — Objekte organisieren (6)

- Briefsortierer — Brieferkennung (LRS): UML-Diagramm



# 6 OOA — Objekte organisieren (7)

- Identifiziere Zustand von Objekten
  - ➤ Attribute (werden zu Instanzvariablen)
  - ➤ Typen
- Identifiziere Verhalten
  - ➤ Operationen / Methoden
  - ➤ Interaktion zwischen Objekten
- Suche nach Ähnlichkeiten, Gemeinsamkeiten
  - ➤ Basis für Vererbungshierarchie
- Suche nach Abhängigkeiten zwischen Objekten
  - ➤ Aggregation
  - ➤ Komposition

BEISPIEL

- ➤ Letter:
  - – Letter ID
  - – ZIP Code
  - – Output Tray

BEISPIEL

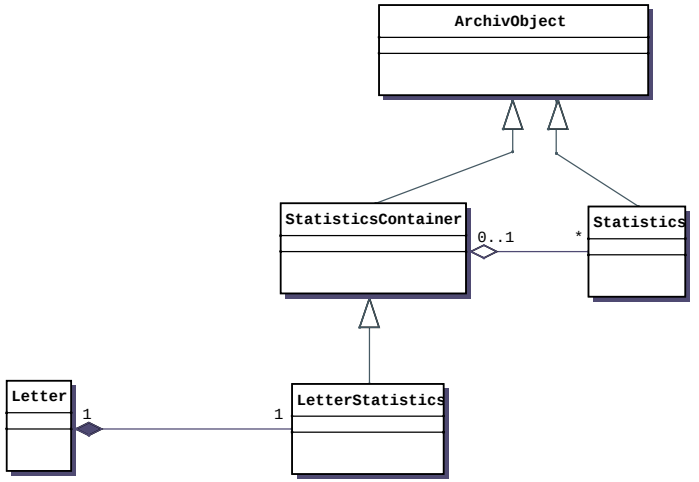
- ➤ Letter "has" a:
  - – Picture
  - – Statistics object






# 6 OOA — Objekte organisieren (8)

- Briefsortierer — Statistik: UML-Diagramm mit Vererbungsbeziehungen



# 7 OOA — Beschreibe Interaktionen

- Ausführung von Use Cases

BEISPIEL
- ➤ Brief bearbeiten
- ➤ Statistiken abfragen

**Lege Operationen fest**

- die wesentlichen Methoden der Objekte

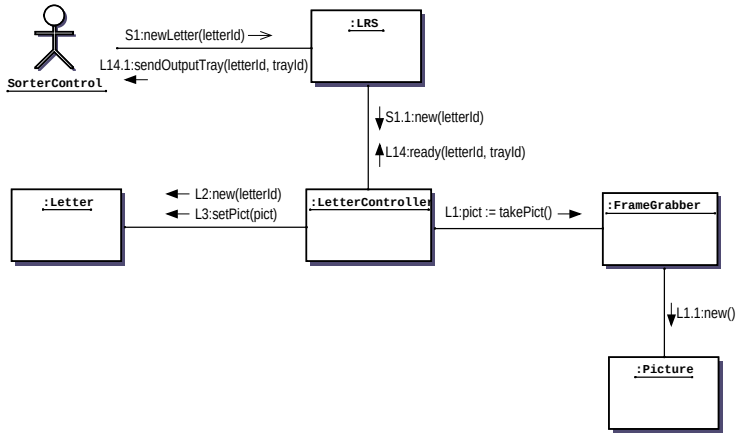
BEISPIEL
- ➤ Sorter Control:
  - – Brief in Fach werfen





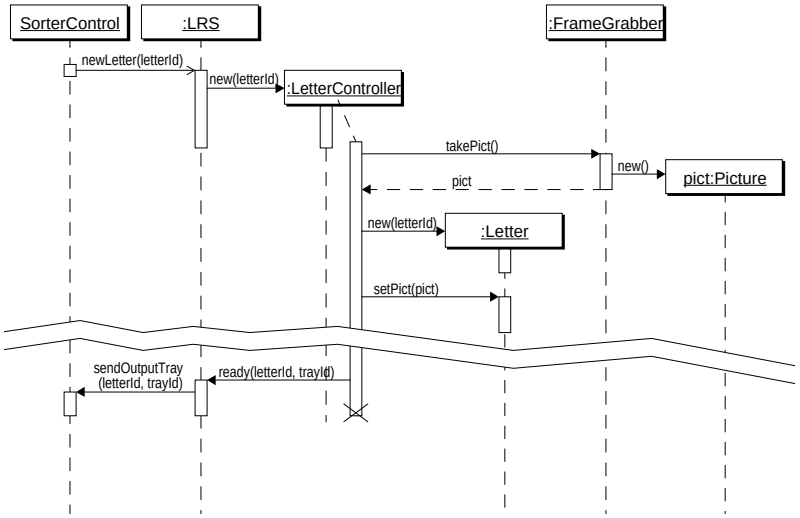

# 7 OOA — Beschreibe Interaktionen (2)

- Briefsortierer — UML Kommunikations-Diagramm



# 7 OOA — Beschreibe Interaktionen (3)

- Briefsortierer — UML Sequenz-Diagramm









# 8 OOA — Struktur verfeinern

- Use Cases strukturieren
  - ➤ Gemeinsame Abläufe identifizieren
- Objekte detaillierter dokumentieren
  - ➤ Attribute
  - ➤ Operationen
  - ➤ Rollen und Verantwortlichkeiten beschreiben

## 9 OOA - OOD?

**Wo endet die Analyse und wo fängt das Design an?**

- Analyse nimmt oft über 50% eines Entwicklungszyklus ein!
- Design beginnt nach 20 - 30%
- Übergänge sind sehr fließend
  - ➤ irgendwann können Implementierungsaspekte nicht mehr zurückgestellt werden






# C.9 Objektorientiertes Design

- Transformation des Analysemodells zu einem implementierbaren Modell
- Aspekte der Implementierungsumgebung aufnehmen
- Strukturelle und strategische Entscheidungen treffen
  - ➤ wo brauche ich eigene Threads
  - ➤ Verteilung der Anwendung auf mehrere Rechner
  - ➤ welche Interprozesskommunikation wird eingesetzt
  - ➤ Datenbankschnittstellen
  - ➤ Fehlerbehandlung
  - ➤ Garbage collection
- Weitere Verfeinerung der Objektinteraktionen und Schnittstellen

## 1 Phasen

- Klassen-Entwurf
  - ➤ Finde Software-Klassen für die Klassen des Analysemodells
  - ➤ Zerteile Analysemodell-Klassen
  - ➤ Entferne unnötige Klassen des Analysemodells
  - ➤ Füge neue Klassen hinzu (z. B. Listen oder Hashtab. zur Verwaltung)

  ! Objektgrenzen müssen erhalten bleiben!
  Analyse ➔ Design ➔ Implementierung (Traceablity)
- Systementwurf
  - ➤ Nicht-problembezogene Aspekte
  (Verteilung, Nebenläufigkeit, Betriebmittel, Systemschnittstellen, …)
- Programmentwurf
  - ➤ Programmiersprache
  - ➤ Fehlerbehandlung (Exceptions, Rückgabewerte)
  - ➤ Performance-Aspekte






# C.10 OOA / OOD - Zusammenfassung

- OOA
  WAS soll mein System tun — nicht WIE
  - ➤ Anforderungsanalyse
  - ➤ Objekte finden und strukturieren
  - ➤ Interaktionen anlysieren
- OOD
  WIE soll mein System arbeiten
  - ➤ Integriere Aspekte der Ausführungsumgebung
  - ➤ treffe strategische Entscheidungen
  - ➤ verfeinere das Objektmodell zu einem implementierbaren Modell

# C.11 Entwurfsmuster (Design Patterns)

[GHJ+97]

> *Each pattern describes a problem which occurs over and over again in our environment,*
> *and then describes the core of the solution to that problem,*
> *in such a way that you can use this solution a million times over,*
> *whithout ever doing it the same way twice*
> *[Christopher Alexander, 1977 —*
> *talking about patterns in buildings and towns]*

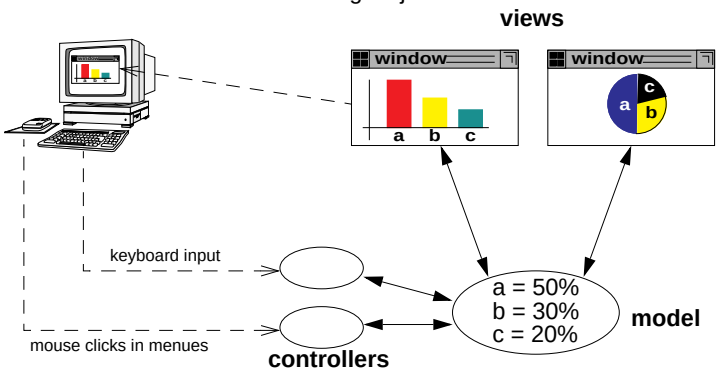
- Entwurfsmuster sind *Lösungen* für *Probleme*, die auftreten, wenn Software in einem bestimmten *Zusammenhang* entwickelt wird
- Muster erfassen die statische und dynamische *Struktur* und die *Zusammenarbeit* zwischen den wesentlichen *Objekten* in einem Software-Entwurf
- Klassen / Klassenbibliotheken = Wiederverwendung von Implementation
  Entwurfsmuster = Wiederverwendung von Entwürfen






# 1 Beispiel: Smalltalk's Model/View/Controller

- Grundlegendes Konzept zum Aufbau von Benutzerschnittstellen
  - *Model*: das Anwendungsobjekt
  - *View*: Bildschirmdarstellung des Anwendungsobjekts
  - *Controller*: nimmt Benutzereingaben entgegen und modfiziert Anwendungsobjekt




Das MVC-Konzept wurde eingeführt, um das eigentliche Anwendungsobjekt von seiner Darstellung am Bildschirm und von der Bearbeitung von Benutzereingaben zur Manipulation zu trennen. Ziel dieser Vorgehensweise ist flexiblere Erweiterbarkeit und Wiederverwendbarkeit.

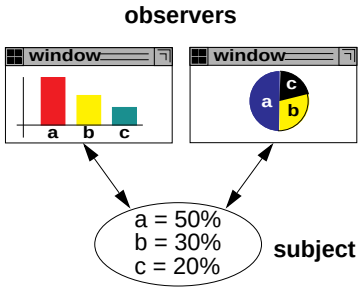
- Model und View werden durch ein subscribe/notify-Protokoll entkoppelt. Ein View meldet sich beim Model an (subscribe) und wird ab diesem Zeitpunkt über Zustandsänderungen durch einen Methodenaufruf (notify) informiert.
  - Auf diese Weise können beliebige Views zur Darstellung eines Objekts eingesetzt werden (prinzipiell auch mehrere gleichzeitig), solange sie die notify-Aufrufe des Models verstehen.
  - Selbst wenn das Model nur das subscribe eines Views unterstützt, können problemlos mehrere Views angekoppelt werden, indem man einfach einen "Verteiler-View" zwischenschaltet, an dem sich dann die eigentlichen Views anmelden.
- Benutzereingaben (Mausklicks, Eingaben über Kontrolltasten, …) werden von einem Controller-Objekt in Methodenaufrufe an dem Model umgesetzt. Verschiedene Benutzer (Anfänger, routiniertes Personal) können unterschiedliche Controller benutzen, ohne dass das eigentliche Anwendungsobjekt hierüber etwas wissen muss. Auch für unterschiedliche Views können jeweils dazu passende Controller eingesetzt werden.

In dem MVC-Konzept kann man verschiedene Design Patterns identifizieren:

# 1 Beispiel: Smalltalk's Model/View/Controller (2)

➥ **Observer pattern**

- ➤ *View* wird von *Model* entkoppelt
- ➤ *View* = *Observer*
- ➤ *Model* = Subjekt
- ➤ subscribe/notify-Protokoll
- ➤ *Observer* unabhängig von Subjekt







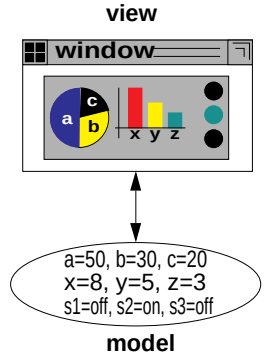


# 1 Beispiel: Smalltalk's Model/View/Controller (3)

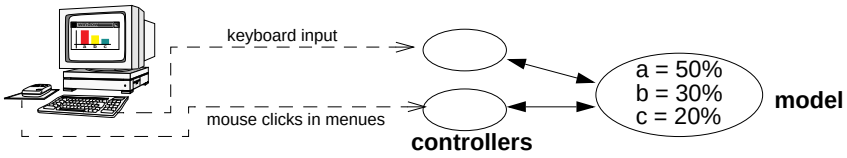
- MVC erlaubt geschachtelte *Views*
  - ➤ ein geschachtelter *View* ist selbst ein *View* (**`CompositeView`** = Unterklasse von **`View`**)

➥ **Composite design pattern**

- ➤ Klassen-Hierarchie
- ➤ primitive Objekte (z.B. Linie, Kreis, Polygon)
- ➤ kompatible Verbundobjekte, die einfache Objekte zu komplexeren (z. B. Bild) zusammenfassen
- ➤ Verbundobjekte können statt einfacher Objekte verwendet werden



# 1 Beispiel: Smalltalk's Model/View/Controller (4)




- Benutzereingaben werden von eigenem *Controller*-Objekt angenommen, das Umsetzung in Methodenaufrufe am *Model* vornimmt

➥ **Strategy pattern**

- ➤ *Strategy*-Objekt steht für einen Algorithmus
- ➤ kann statisch oder dynamisch ersetzt werden — unabhängig vom *Model*
- ➤ *Strategy*-Objekte können unterschiedliche Varianten eines Algorithmus kapseln
- ➤ *Strategy*-Objekte können komplexe Datenstrukturen kapseln






# 2 Elemente eines Entwurfsmusters

- Pattern-Name
- Problem
  - ➤ beschreibt das Problem und den Kontext
- Lösung
  - ➤ beschreibt
    - die Elemente,
    - ihre Beziehung unterienander,
    - Verantwortlichkeiten,
    - das Zusammenwirken
- Resultate
  - ➤ Ergebnisse
  - ➤ Auswirkungen, Nebenwirkungen
    (Probleme in Bezug auf Platz- und Zeitbedarf, Programmiersprache, Implementierung, …)

# 3 Design Pattern Space

| | | Purpose | | |
|---|---|---|---|---|
| | | Creational | Structural | Behavioral |
| Scope | Class | Factory Method | Adapter (class) | Interpreter<br>Template Method |
| | Object | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter (object)<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

In der Literatur ist heute eine große Zahl weiterer Entwurfsmuster zu finden - häufig abgestimmt auf bestimmte Anwendungs- oder Problembereiche. Gamma, Helm, Johnson und Vlissides haben sich in ihren Arbeiten bemüht, die Menge der beschriebenen Entwurfsmuster möglichst klein und universell zu halten. Gamma ist der Ansicht, dass ein Großteil der seit dem von anderen publizierten Entwurfsmuster eigentlich auf die ursprünglich identifizierten Muster zurückführbar ist.

Dies bedeutet natürlich nicht, dass die Entwicklung spezieller Entwurfsmuster grundsätzlich nicht sinnvoll ist. Eine gute Kenntnis der "Basis-Muster" und ihrer Anwendung kann einem bei die Bewertung, Einordnung und Auswahl spezieller Entwurfsmuster für spezielle Anwendungsbereiche aber stark erleichtern.

Beispiele:

- Abstract Factory: Objekt mit einer Schnittstelle zum Erzeugen von Objekten
- Builder: Objekt zur Erzeugung eines komplexen Objektaggregats
- Adapter (Wrapper): Passe Schnittstelle an andere, vom Client erwartete Schnittstelle an
- Bridge (Handle/Body): Entkopplung von Abstraktion und Implementierung
- Proxy: Stellvertreterobjekt
- Action: Befehle als Objekt kapseln
- Iterator: Sequentieller Zugriff auf Elemente eines zusammengesetzten Objekts
- Memento: Zustand eines Objekts erfassen (um ihn später rekonstruieren zu können)