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

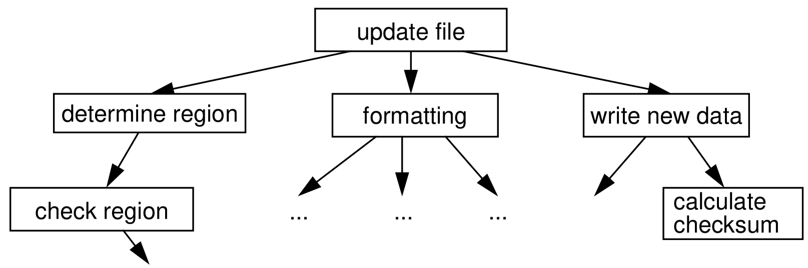
- Probleme bei sehr komplexen Aufgabenstellung und objektbasierten oder objektorientierten Programmiersprachen
  - *Structured design* resultiert in einer Problemstrukturierung, die nicht auf die Mechanismen in objektbasierten oder objektorientierten Sprachen abgestimmt ist

- 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.

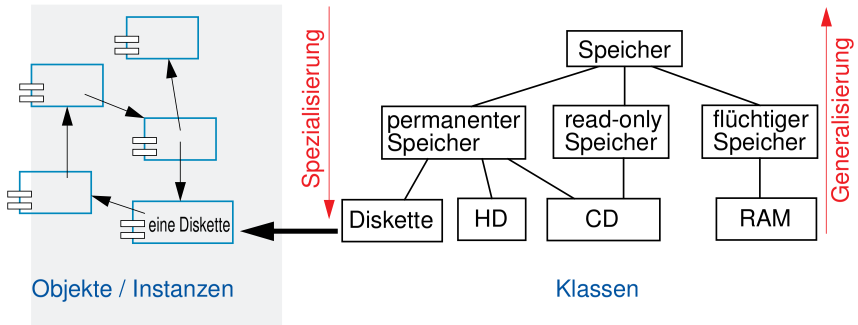
siehe auch [Mey88]

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

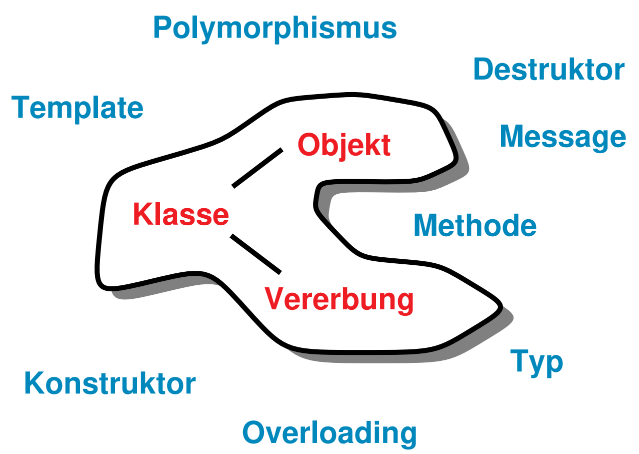
ist und die Klassen Bestandteil einer über

Vererbungsbeziehungen

definierten Hierarchie von Klassen sind.

- Diese Definition enthält die wesentlichen Grundbegriffe objektorientierter Programmierung:

2 Grundbegriffe



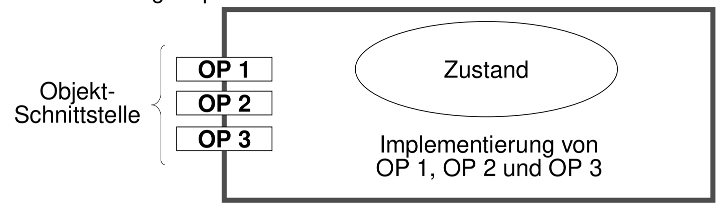
➜ Objekt
- Objektorientierte Programmierung verwendet Objekte und nicht Algorithmen als die grundlegenden Bausteine der Problemlösung

➜ 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

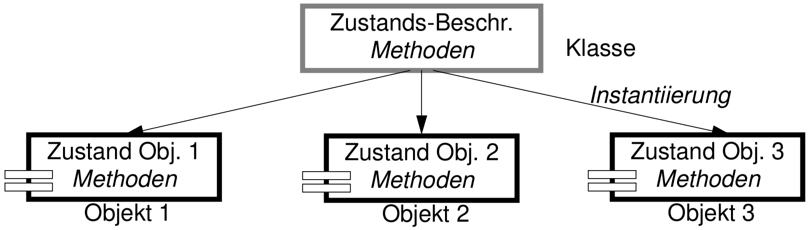
  → Abhilfe: Schablone zur Erzeugung von Objekten

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)

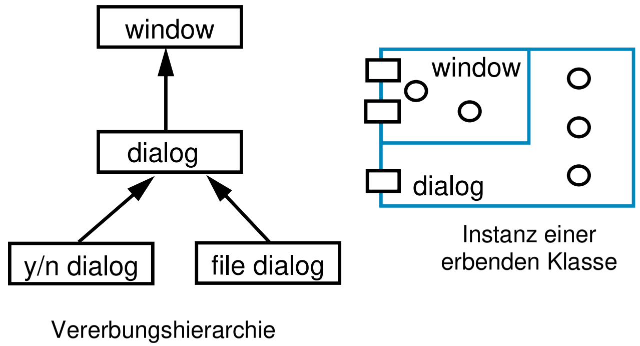
➜ **Klassenbasierte Programmiersprachen**
= Objekte & Klassen

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



- 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.

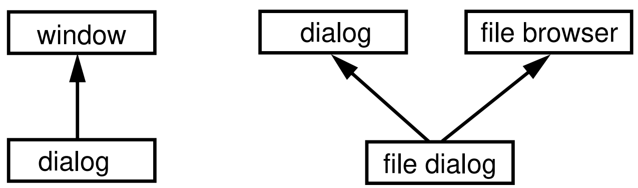
Oberklasse

- 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)

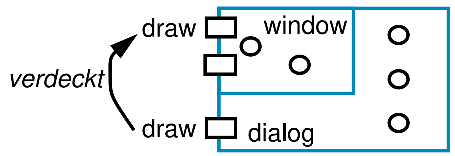
★ **Programmiertechnische Sicht**

- 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

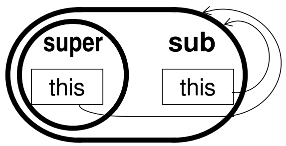
- Entscheidung welche Methode ausgeführt wird erfolgt zur Laufzeit (dynamisch)

```
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.

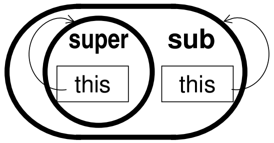
- Gilt auch, wenn ein Objekt eine Methode an sich selbst aufruft!
  - ◆ Beispiel:
    - – **`move()`** ruft am Ende **`display()`** auf, um das Fenster neu zu zeichnen
    - – **`BorderedWindow`** erbt **`move()`** von **`Window`**
    - – ein Aufruf von **`move()`** an einer Instanz von **`BorderedWindow`**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