2. Objektorientierung

Geschichte

Dahl, Nygaard: Simula 67
(Algol 60 + Objektorientierung)

Kay et al.: Smalltalk
(erste rein-objektorientierte Sprache)

Object Pascal, Objective C, C++
(wiederum Erweiterungen einer gegebenen Basissprache)

Java
(C++ ohne Hacks?)

Software-Entwicklung (I)

Große Softwaresysteme:

- Entwurf: (wiederholte) Zerlegung in Komponenten.
- Implementierung und Test von einzelnen Komponenten.
- Integration: Zusammenbau der Komponenten.

Problem: Komponenten nicht isoliert.

Software-Entwicklung (II)

Komponenten müssen ohne Kenntnis ihrer Implementierung verwendet und ohne Kenntnis ihrer Verwendung implementiert werden können:

- Überschaubarkeit.
- Zu Beginn: keine Implementierung vorhanden.
- Auswechselbarkeit.

Komponenten dürfen nicht auf Implementierungsdetails anderer Komponenten zugreifen:

- Auswechselbarkeit
- Zugriff auf interne Daten erhöht Gefahr von Fehlern.

Abstraktion

Benutzer und Implementierer einer Komponente bekommen die wesentlichen Informationen:

- Schnittstelle.
- Verhalten.
- Kontrakt.

Kapselung

Benutzer und Implementierer einer Komponente bekommen nicht mehr als die wesentlichen Informationen.

Implementierungsdetails sind von außen nicht zugänglich.

Aufrufende Komponente hat keinen Zugriff auf private Daten der aufgerufenen Komponente (und umgekehrt).

Mechanismen (I)

Prozeduren, Funktionen, Module, abstrakte Datentypen, *Objekte*.

Prozeduren, Funktionen:

- Trennung zwischen globalen und lokalen Daten.
- Trennung zwischen Aufruf (Schnittstelle) und Implementierung der Funktion.
- Keine syntaktische Unterscheidung zwischen von außen zugreifbaren Funktionen und Hilfsfunktionen.
- Keine für mehrere Funktionen lokalen Daten.
- In der Regel: Lokale Daten bei jedem Funktionsaufruf neu initialisiert.

Mechanismen (II)

Module:

- Separater Namensraum.
- Namensraum unterteilt: öffentlich/privat.
- Feinere Abstufungen der Lokalität.
- Keine mehrfachen Kopien (Instantiierung).

Mechanismen (III)

Abstrakte Datentypen:

Benutzerdefinierter Datentyp
+ öffentliche Operationen auf diesem Datentyp
(+ Spezifikation des Verhaltens)

Zugriff ausschließlich über öffentliche Operationen.

Implementierung privat (wie bei Modulen).

Trennung zwischen öffentlichen und privaten Daten auch für dynamisch erzeugte Datenstrukturen.

Objekte und Klassen (I)

Im wesentlichen:

Abstrakter Datentyp $\approx$ Klasse.

Element eines ADT $\approx$ Objekt.

$$\text{Objekt} = \underbrace{\underline{\text{Zustand}} + \underline{\text{Verhalten}}}_{\text{gekapselt}}.$$

Jedes Objekt ist Exemplar (Instanz) einer Klasse.

Klassen definieren das Verhalten ihrer Exemplare.

Objekte und Klassen (II)

Betrachte Operationen (Methoden) als Bestandteile eines Objektes (analog zu Recordkomponenten):

Ein Objekt `st` der Klasse `Stack` besitzt Methoden `push` und `top`.

`push(st,elem)` $\rightsquigarrow$ `st.push(elem)`
`x = top(st)` $\rightsquigarrow$ `x = st.top()`

Objekte und Klassen (III)

Nomenklatur:

Ein anderes Objekt schickt dem Objekt `st` eine Nachricht `push(elem)`.

`st` führt daraufhin eine Methode aus.

Berechnung = Kommunikation.

Konzept gut geeignet für graphische Benutzungsoberflächen.

Berechnung = ereignisgesteuerte Simulation.

Konzept gut verträglich mit Parallelismus.

Objekte und Klassen (IV)

Objekte verschiedener Klassen können die gleiche Nachricht verstehen ...

square1.draw(screen)
helpText.draw(screen)
processorLayout.draw(screen)
photo.draw(screen)

stack.push(x)
robot.push(x)

und unterschiedlich interpretieren:

Jedes Objekt führt seine Methode `draw(screen)` bzw. `push(x)` aus.

Die Verantwortung für das Interpretieren einer Nachricht liegt beim Empfänger.

Objektorientiert ↔ prozedural

Prozedural:

Eine Prozedur `draw`.

⇝ große Switch-Anweisung.

Hinzufügen einer neuen Klasse graphischer Objekte ⇒ alle Funktionen, die wie `draw` graphische Objekte verarbeiten, müssen erweitert werden.

Objektorientiert:

Jedes Objekt hat seine Methode `draw`.

Neue Klasse graphischer Objekte definiert ihre eigenen Methoden.

Klassenhierarchie (I)

Klassen sind hierarchisch angeordnet: Oberklasse ↔ Unterklasse.

In Smalltalk und Java: alle Klassen sind Unterklassen von `Object`.

Unterklassen erben Daten und Verhalten (Methoden) ihrer Oberklasse. (transitiv!)

Unterklassen erweitern ihre Oberklasse indem sie zusätzliche Daten und Methoden definieren.

Klassenhierarchie (II)

Objekte von Unterklassen können überall verwendet werden, wo Objekte ihrer Oberklasse verwendet werden können:

Bsp.: Oberklasse `Window`, Unterklassen `TextEditWindow` und `NotifierWindow`.

Ein Objekt der Klasse `TextEditWindow` versteht jede Nachricht, die ein `Window` versteht (aber nicht umgekehrt).

Eine Methode, die als Argument ein Objekt der Klasse `Window` akzeptiert, akzeptiert auch ein `TextEditWindow`.

Einer Variable mit dem Typ `Window` kann man Objekte der Klassen `Window`, `TextEditWindow` oder `NotifierWindow` zuweisen.

Klassenhierarchie (III)

Ersetzbarkeitsprinzip: Objekte von Oberklassen *sollten* ohne sichtbaren Effekt durch Objekte von Unterklassen ersetzbar sein.

Klassenhierarchie (IV)

Unterklassen können Methoden ihrer Oberklasse überschreiben (*override*, eigentlich: umstoßen, nichtig machen).

Bsp.: Oberklasse `Window`, Unterklasse `IconWindow`.

Ein `IconWindow` verhält sich im wesentlichen wie ein anderes `Window`:

z. B.: Methoden `display`, `move`, `delete`.

Aber: `iconize` für `IconWindow` nicht sinnvoll.

⇒ In `IconWindow` wird die Methode `iconize` der Oberklasse überschrieben (Fehlermeldung oder Nichtstun).

⇒ Ersetzbarkeitsprinzip kann verletzt werden.

Klassenhierarchie (V)

Überschreiben von Methoden

⇒ Welche Methode bei einer bestimmten Nachricht ausgeführt wird, kann erst zur Laufzeit entschieden werden:

Variable `win` des Typs `Window` kann ein `Window` oder `TextEditWindow` oder `NotifierWindow` oder `IconWindow` enthalten.

Welche Methode wird bei `win.iconize` ausgeführt?

In den meisten OO-Programmiersprachen: Suche von der Klasse des Objekts an aufwärts. (Ausnahme: C++.)

Klassenhierarchie (VI)

Zwei Schnittstellen:

– gegenüber Benutzer der Klasse,

– gegenüber Implementierer einer Unterklasse.

Sichtbarkeit von Daten und Methoden?

Vorteile der Vererbung

Wiederverwendung

Code sharing

Konsistenz der Schnittstellen

Aber: Effizienz?

Abstrakte Oberklassen (I)

Bsp.: Klassen `List`, `Tree`, `Hashtable`.

Alle Klassen verstehen Nachrichten `insert(x)`, `delete(x)`, `size()`, ...

Aber: keine gemeinsame Implementierung.

Abstrakte Oberklassen (II)

Idee: führe Klasse `Collection` ein; definiere `List`, `Tree`, `Hashtable` als Unterklassen von `Collection`.

Abstrakte Oberklasse `Collection` deklariert Methoden `insert(x)`, `delete(x)`, `size()`, aber implementiert sie nicht.

Darum: `Collection` selbst besitzt keine Exemplare; Objekte gehören immer zu einer Unterklasse von `Collection`.

Aber: Variable können Typ `Collection` haben.

Mehrfache Vererbung

In manchen Sprachen: Klasse kann als Unterklasse mehrerer Oberklassen definiert werden.

Bsp.: `ColoredText` Unterklasse von `Text` und `ColoredObject`.

Probleme:

Konflikte zwischen Methoden in den Oberklassen.

Objekte können in verschiedenen Oberklassen unterschiedlich initialisiert werden.

OOP zusammengefaßt

Objekt = Daten + Methoden (Funktionen)

Objekte haben privaten Speicher (andere Objekte).

Berechnung = Kommunikation (Nachrichten $\rightsquigarrow$ Methoden).

Objekte sind Exemplare (Instanzen) von Klassen.

Klassen definieren Verhalten von Objekten.

Klassen sind hierarchisch angeordnet (Vererbung/Überschreiben von Methoden).