Programmiersprachen

Vorlesung 9 vom 04.12.23

Datenabstraktion

Christoph Lüth

Universität Bremen

Wintersemester 2023/24

Organisatorisches

- Vorträge — kleine Verschiebungen
- Aktuelle Version immer auf der Webseite:
  `https://user.informatik.uni-bremen.de/clueth/lehre/ps.ws23/`

Wo sind wir?

Abstraktion

- Definition Wikipedia:

> Das Wort Abstraktion[a] bezeichnet meist den [...] Denkprozess des erforderlichen Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres. Daneben gibt es spezifische sowie unspezifische Verwendungen des Begriffes in bestimmten Einzelwissenschaften und einzelnen Theorien, Thesen sowie Behauptungen.
>
> [a] lat. *abstractus* "abgezogen", Partizip Perfekt Passiv von abs-trahere "abziehen", "trennen"

- Weiter: "In der Mathematik [...] werden Abstrakta meist mit Äquivalenzklassen identifiziert."
- Im Lambda-Kalkül ist Abstraktion $\lambda x.\, t$ — die Einführung des Funktionsparameters

Arten der Abstraktion

- Kontrollabstraktion (done):
  - Strukturierte Programmierung (statt GOTO)
  - Ausnahmen (strutkurierte Fehlerbehandlung)
  - Prozeduren und Parameter
- Datenabstraktion (heute):
  - Abstrakte Datentypen
  - Verkapselung und Objekte
  - Module
  - Packages

Wozu Datenabstraktion?

- Kontrollabstraktion hilft uns, Programme **verständlich** zu machen.
- Datenabstraktion hilft uns, **große** Programme verständlich zu machen.
- Indem wir existierende Daten und Funktionen (Methoden) zu neuen Datentypen zusammenfassen erlauben wir Abstraktion in der Sprache.
  - Abstraktion = "geordnetes Weglassen", hier: von Implementationsdetails.
  - Triviales Beispiel: `Int` als `Bool`, ist `0` jetzt `True` oder `False`?

Abstrakte Datentypen

Abstrakter Datentyp (ADT)

Ein ADT besteht aus besteht aus einem (oder mehreren) **Typen** und **Operationen** darauf, mit folgenden Eigenschaften:

1. Werte des Typen können nur über die Operationen **erzeugt** werden
2. Eigenschaften von Werten des Typen werden nur über die Operationen **beobachtet**.
3. Die Einhaltung von **Invarianten** über dem Typ kann garantiert werden

- Damit eine Programmiersprache ADTs unterstützt, müssen wir die **Sichtbarkeit** einschränken können (*information hiding*).
- **Repräsentationsunabhängigkeit**: Eigenschaften sollten von konkreter Implementation unabhängig sein.

Beispiel: Stack

Ein Stack (von ganzen Zahlen) hat mehrere Operationen:

- den leeren Stack (**empty**),
- eine Zahl auf den Stack schieben (**push**),
- die oberste Zahl vom Stack nehmen (**top** und **pop**),
- und einen Test, ob der Stack leer ist (**isEmpty**).

mit folgenden Eigenschaften:

1. der leere Stack ist leer, und nur der;
2. das oberste Element das Stacks ist dasjenige, was zuletzt darauf geschoben worden ist;
3. wenn ich von einem Stack, auf den ich ein Element geschoben habe, das oberste Element wieder herunternehme, ändert sich nichts.

Spezifikation

- Wollen Stacks **mathematisch** beschreiben
  - Möglichst unzweideutig
  - Können daraus Tests generieren
- Hier:

$$\text{top}(\text{push}(s, x)) = x \qquad \text{isEmpty}(\text{empty})$$

$$\text{pop}(\text{push}(s, x)) = s \qquad \neg(\text{isEmpty}(\text{push}(s, x)))$$

- Gleichungen gelten nur für **unveränderliche** (zustandsfreie) Stacks
- Was bedeutet Gleichheit?
  - Gleichheit für `int` — bekannt
  - Gleichheit für `stack` — nicht gleich, nur **beobachtbar** gleich

Stack: Implementation

- Beispiel: Stack in Python
  - Stack als Liste
  - Stack als Feld
- Unveränderliche Stacks (immutable): `push` und `pop` liefern **neuen Stack**
- Veränderliche Stacks (mutable): `push` und `pop` haben Seiteneffekte
- Problem: Python schränkt Sichtbarkeit bedingt ein
  - Keine Trennung zwischen Interface und Implementation
  - Private Felder **syntaktisch** gekennzeichnet (`__name`), Zugriff trotzdem möglich (als `_Class__name`)
  - Python Design-Prinzip: "Wir sind alle erwachsen."

Stacks in anderen Sprachen

- Beispiel: Stack in C
  - Interface `stack.h` syntaktisch getrennt
- In Haskell:
  - Wir können Sichtbarkeit einschränken, aber syntaktisch nicht trennen
  - **Nur** funktionale Lösung
- In Java:
  - **Interface** als separates Konstrukt

Sprachmittel zur Unterstützung von Datenabstraktion

- Sichtbarkeitseinschränkungen aka. Module
- Syntax zur Beschreibung von Schnittstellen
- Trennung der Schnittstelle von der Implementation
- Modularisierung der Schnittstellen (`interface` in Java, `trait` in Rust, Mix-Ins in Python)

Module

- Ein **Modul** ist die Zusammenfassung mehrerer Definition zu einer Einheit
  - Oft mit Verkapselung — Modul hat definierte **Schnittstelle**
  - Module dienen oft auch zur **getrennten Übersetzung** (aber nicht notwendigerweise)
- Module werden deklariert, definiert und benutzt (importiert).
  - Trennt die Sprache das?
  - Wie erfolgt die Benutzung?
  - Wie verhalten sich Module zu Quelldateien?
- Flacher Namensraum für Module vs. Pfade für Module

Import

- Wird **qualifiziert** oder **unqualifiziert** importiert?
  - Qualifizierter Import: Bezeichner `f` aus Modul `M` wird zu `M.f`.
- Wird immer **alles** importiert, oder kann der Importeur selektieren (wie)?
  - Positive Selektion: importiere `f`, negative Selektion: alles bis auf `f`
- Kann beim Import **umbenannt** werden?
  - Importiere `f` aus `M` und nenne es `g`; importiere `f` aus `M` und nenne das Module `N`

Module in Python

- Module sind Quelldateien
- Keine Interface-Definition
- Kaum Sichtbarkeitseinschränkungen
- Keine Datenabstraktion
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Haskell

- Jede Quelldatei ist ein Modul
- Interface: Sichtbarkeit von Bezeichnern kann eingeschränkt werden
- Aber:
  - algebraische Datentypen und Konstruktoren bleiben erkennbar
  - Typsynonyme sind nicht abstrakt
  - Klasseninstanzen werden immer exportiert
- Datenabstraktion möglich (manchmal umständlich)
- Interface keine separate Datei
- Import: mit Namen, qualifiziert/unqualifiziert, alles oder selektiv, Umbenennung möglich

Module in Java

- Module sind **Klassen** (nicht an Quelldatei gebunden)
- Klassen verkapseln interne Repräsentation, Einschränkung der Sichtbarkeit (`public`, `private`, `protected`)
  - Aber Konstruktoren bleiben erkennbar
- Datenabstraktion möglich (durch Reflektion zu durchbrechen?)
- Interfaces sind separates Konstrukt
- Import: nur ganze Klassen, keine Umbenennung, impliziter Import möglich

Module in C

- Module sind Quelldateien (*translation units*)
- Sichtbarkeitseinschränkungen für Bezeichner (`static`, `extern`)
  - Local and global linkage
- Interfaces sind **per Konvention** separate Dateien (`.h`)
  - Konvention wird durch den Präprozessor ermöglicht
- Datenabstraktion möglich (durch Zeigeroperationen zu durchbrechen)
- Import: immer alles, nur unqualifziert, keine Umbenennung

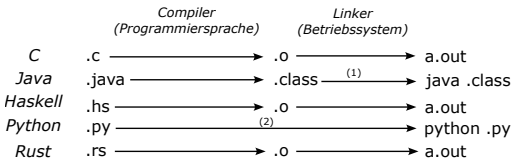
Packages

- Packages schränken die Sichtbarkeit von Modulen ein.
- Existiert in Java, Python.
- Haskell kennt nur hierarchische Module.
- Wenn Module Quelldateien entsprechen, sind Packages Verzeichnisse.

Sprachneutral Interfaces: IDL

- IDL ist die **Interface Definition Language** der OMG
- Definition der Schnittstelle von Komponenten in CORBA
- Syntax an C angelehnt
- Compiler erzeugt aus IDL "Rumpf" in entsprechender Programmiersprache
- Alle Funktionsaufrufe gehen über einen **Broker**
- Nicht mehr ganz aktuell.

Übersetzung: Von der Quelle zum executable



- Java hat plattformübergreifendes Objektdateiformat (1)
- Python interpretiert Quellcode direkt (2)

Der Build-Prozess: Am Anfang war `make`

- Programmiersprachenunabhängig
- Programmierer spezifiziert **Abhängigkeiten manuell**:

```
a.o:  a.c b.h c.h
    gcc -c a.c

b.o:  b.c c.h
    gcc -c b.c

c.o:  c.c
    gcc -c c.c

abc: a.o b.o c.o
    ld -o abc a.o b.o c.o -lfoo
```

Vereinfacht durch vordefinierte **Regeln**:

```
a.o: b.h c.h

b.o:  c.h

abc: a.o b.o c.o
    ld -o abc a.o b.o c.o -lfoo
```


Unterstützung des Build Prozesses

- Analyse der Abhängigkeiten in viele Übersetzer **integriert** (Haskell, Rust)
- Verwaltung **externer Büchereien**:
  - Erweitertes Abhängigkeitsprinzip
  - Globale Büchereien (zentrales Repository)
- Werkzeuge:

Sprache	Werkzeugk	Zentr. Repository
C	(CMake)	—
Java	Maven	Maven
Haskell	stack/cabal	Hackage
Python	pip	PyPI
Rust	cargo	`crates.io`

Zusammenfassung

- Datenabstraktion durch **abstrakte Datentypen**:
  - Typ mit Operationen darüber
  - Zugriff nur über definierte Schnittstelle
  - Repräsentationsunabhängigkeit
- Module: **Verkapselung** durch Einschränkung der Sichtbarkeit
  - Was wird verkapselt: Sichtbarkeit der Bezeichner, Typrepräsentation, Konstruktoren?
  - Sind Interfaces explizit oder implizit?
  - Modul = Quelldatei?
  - Wie wird importiert?
  - Qualifizierte Bezeichner `M.f`
- Packages: Sammlungen von Modulen
- Fallbeispiel: Module in SML