

Die Bedeutung abstrakter Datentypen in der objektorientierten Programmierung

Klaus Kusche, September 2014

Inhalt

- Ziel & Voraussetzungen
- Was sind “abstrakte Datentypen” ?
- Was kann man damit grundsätzlich?
- Drei Arten der Anwendung
incl. Beispielen
- Nutzen

Ziel

- Konzeptionelles Verständnis
 - Kenntnis typischer Beispiele und Einsatzgebiete
 - Kenntnis des Nutzens des Konzeptes
-
- *Bestehenden Code mit abstrakten Klassen verstehen und nutzen können*
 - *Selbst Aufgabenstellungen durch abstrakte Klassen sinnvoll strukturieren und codieren können*

Voraussetzungen

Kenntnis zumindest einer

- streng typisierten,
- objektorientierten

Programmiersprache

ideal: C++, auch: Java, C#, ...
eigener Background: C++

Konzeptionelles Verständnis von

- **Klassen, Methoden, ...**
- **Vererbung / Ableitung**

Definition (1)

Hier: Programmiertechnische Bedeutung:

“Abstrakter Datentyp”

=

“Abstrakte Klasse”

Nicht: Bedeutung in der Logik / Mathematik:

Axiomatische Definition von Datentypen:

*“Wie definiere ich einen Datentyp formal
als Gleichungen auf seinen Funktionen?”*

Auch nicht: Programmiertechnische Bedeutung “Templates” / “Generics”

Definition (2)

In Java usw.: Klasse mit “**abstract**” davor

???

Definition (2)

In Java usw.: Klasse mit “**abstract**” davor

In C++: Klasse mit “rein virtueller” Methode =

*Klasse, bei der der **Code**
mindestens einer deklarierten Methode
(im Extremfall aller Methoden) **fehlt**.*

Entweder explizit: = \emptyset statt { **code...** }

Oder “fehlend geerbt” und nicht überschrieben

Es dürfen auch die Member-Deklarationen fehlen

(d.h. eine abstrakte Klasse kann Daten enthalten, muss aber nicht).

Definition (3)

- Methoden-Deklaration (= Prototyp) ist vorhanden

→ Die abstrakte Klasse beschreibt die Schnittstelle eines Datentyps nach außen:

“Was können die Objekte alles (mindestens)?”

- Methoden-Implementierung (= Code) fehlt

→ Die abstrakte Klasse lässt die tatsächliche Realisierung der Funktionalität offen:

Nicht: “Wie machen die Objekte das?”

Ev. nicht einmal: “Wie werden die Daten dargestellt?”

Abstrakte Klassen: Was geht?

Am wichtigsten:

- Man kann von abstrakten Klassen ganz normal

ableiten

- Man kann damit Variablen, Parameter, Arrays, ...

deklarieren

Genauer: *Nur Pointer auf Objekte abstrakter Klassen*

In C++:

Man muss explizit Pointer deklarieren

In Java, ...:

Objekt-Variablen sind intern automatisch immer Pointer

... und was geht nicht?

Man kann

kein Objekt einer abstrakten Klasse
direkt erzeugen

... weil dieses Objekt die von der Klasse deklarierten
“*codelosen*” Methoden ***nicht ausführen könnte!***

Aber:

Objekte davon ***abgeleiteter Klassen***
sind ***normal erzeugbar***

... wenn die abgeleitete Klasse nicht mehr abstrakt ist,
d.h. ***alle fehlenden Methoden implementiert*** hat!

Wie funktioniert das?

Code, der abstrakte Klassen verwendet, weiß

“Ich kann für eine Objekt-Variable (Parameter, ...) einer abstrakten Klasse alle in der Klasse deklarierten Methoden aufrufen, ...

... weil jedes konkrete Objekt (abgeleiteter Klassen), auf das die Variable zur Laufzeit verweisen kann, alle Methoden können wird!”

Der aufgerufene Code muss zur Compile-Zeit nicht bekannt sein, nur der Prototyp!

Erfordert dynamische Bindung zur Laufzeit (tatsächliche Klasse des Objekts → aufzurufender Code)

Anwendung 1: “Klassische” ADT

Abstrakte Vaterklasse erlaubt

mehrere verschiedene, jederzeit austauschbare
Implementierungen derselben Funktionalität.

Beispiel: “Stack” (**push**, **pop**, **isEmpty**, ...)

Zwei Implementierungen mit Array oder mit Liste,
die sich nach außen absolut ident verhalten.

- → “Aufrufender” Code läuft unverändert mit beidem!
Einzige anzupassende Stelle:
Erzeugung des konkreten Stack-Objektes

Beispiel: “CryptAlg” (**setKey**, **encode**, **decode**, ...)

Implementierungen mit verschiedenen Algorithmen

Anwendung 2: Heute übliche ADT (1)

Die abstrakte Vaterklasse

*fasst die gemeinsamen Grund-Eigenschaften
verschiedener, aber verwandter Klassen
zusammen,*

*d.h. beschreibt, was jedes Objekt der “Familie”
mindestens hat oder kann.*

→ *Jede abgeleitete Klasse kann typischerweise
mehr als die abstrakte Vaterklasse!*

→ *Die abgeleiteten konkreten Klassen
verhalten sich verschieden.*

Anwendung 2: **Heute übliche ADT (2)**

Beispiel aus der GUI-Programmierung:

- Viele Klassen für alle Dialog-Elemente
(d.h. alles, was in einem Dialog-Fenster angezeigt werden kann):
Button, TextEntry, ProgressBar, Choice, ...
- Gemeinsame, abstrakte Vaterklasse **DialogElement**:

Enthält gemeinsame Eigenschaften:

x/y-Pos, x/y-Größe, Vorder- und Hintergrundfarbe, ...

... und gemeinsame Methoden:

Draw(), Resize(...), Enable() / Disable(), ...

Anwendung 2: Heute übliche ADT (3)

... aber kann diese offensichtlich nicht implementieren, weil sie für jede konkrete Klasse etwas Anderes tun!

Jede abgeleitete Klasse

- ... enthält meist zusätzliche Member
(z.B. Button...Label, TextEntry...Text, ProgressBar...%)
- ... implementiert die fehlenden “Pflicht-Methoden”
(in jeder Klasse anders!)
- ... bietet meist zusätzliche Methoden
(z.B. Button...SetLabel, TextEntry...GetText, ...)

Anwendung 2: Heute übliche ADT (4)

Davon unabhängig: Fenster-Klasse **DialogFrame**,
enthält als Member alle darin enthaltenen GUI-Elemente:

Ein Array von **DialogElement**

Code beispielsweise beim Öffnen des Dialogfensters:

- Schleife über dieses Array
- Aufruf von **Draw()** für jedes Element
(ohne zu wissen, was konkret gezeichnet wird!)

Weil die abstrakte Klasse **DialogElement** garantiert:

Jedes **DialogElement** “kann” **Draw()**
(mit derselben Schnittstelle)!

Anwendung 2: Heute übliche ADT (5)

Zur Laufzeit:

Das Array enthält eine Mischung
von Objekten verschiedener Klassen

→ Für jedes wird intern
eine andere Draw()-Implementierung aufgerufen!

Analog z.B.:

Code für Anordnung und Sizing der Dialogelemente:
Ordnet abstrakte “Rechtecke” (x/y-Pos und x/y-Größe) an,
ohne konkrete Kenntnis von deren Inhalt.

Anwendung 3: Interfaces: Konzept

- Interfaces sind “völlig abstrakte” Klassen:

Nur Methoden-Deklarationen

Kein Code, auch keine Konstruktoren/Destruktoren

Keine Daten (keine Member-Variablen)

- Interfaces beschreiben

allgemeine Eigenschaften oder Fähigkeiten,

die mehrere beliebige,

logisch voneinander unabhängige Klassen

gemeinsam haben können.

Anwendung 3: **Interfaces: Beispiele**

- **“comparable”**:
“Objekte der Klasse sind in der Größe vergleichbar”
→ Klasse hat == und < (bzw. in Java: **compareTo(...)**)
- **“serializable”**:
“Objekte der Klasse sind als Byte-Sequenz darstellbar”
→ Objekte können u.a. auf Platte gespeichert
und via Netz übertragen werden
- **“iterable”**:
(bei Containern = Datenstrukturen, die mehrere Elemente enthalten)
“Ich kann eine Schleife über alle Elemente machen”
→ Klasse hat z.B. **getFirst(...)**, **getNext(...)**

Anwendung 3: **Interfaces: Verwendung**

Interfaces werden meist per Mehrfach-Vererbung als sekundäre Vaterklassen “dazugeheiratet” (unabhängig von der eigentlichen Klassen-Hierarchie!):

Z.B.: Alles was vergleichbar ist, bekommt **comparable** (*Strings, Datumstyp, Bruchzahlen, Personendaten, ...*)

Eine universelle Sortierfunktion wird dann z.B. mit Parameter “Array von **comparable**” definiert
→ *Sie weiß dadurch, dass die Elemente == und < haben!*

Warum müssen Interfaces “völlig abstrakt” sein?
Verständnis und technische Implementierung sind viel einfacher und effizienter als bei “universeller” Mehrfach-Vererbung!

Nutzen (1)

- Klarere Modellierung,
bessere Abbildung gedanklicher Strukturen
- Bessere Entkopplung der Klassen,
Entwicklungstätigkeiten früher parallelisierbar
- Zusammenfassung gleicher Funktionalitäten,
Vermeidung von doppeltem Code:
 - Umfangreiche Code-Ersparnis
 - Bessere Wartbarkeit
 - Weniger und kürzere Methodennamen
 - Bessere, intuitivere Lesbarkeit

Nutzen (2)

- Leichte nachträgliche Erweiterbarkeit um neue konkrete Klassen (GUI-Beispiel: z.B. **ButtonWithIcon**):

Auf abstrakter Klasse basierender allgemeiner Code (hier: **DialogFrame** usw.) braucht nicht geändert werden, nicht einmal neu kompiliert.

Im Extremfall:

*Laden beliebiger neuer konkreter Klassen
als Plugin zur Laufzeit!*

“The end”

Fragen?