Polymorphie

Begriffsklärung und Anwendung

Klaus Kusche, Mai 2014

Inhalt

- Ziel & Voraussetzungen
- Was bedeutet “Polymorphie” ?
- Die einzelnen Arten der Polymorphie:
  - Konzept
  - Beispiel / Anwendung
  - Nutzen
  - Implementierung

Ziel

- Konzeptionelles Verständnis
- Implementierungstechnisches Verständnis
  (u.a.: Effizienz)
- Kenntnis typischer Beispiele und Einsatzgebiete

➔ *Bestehenden polymorphen Code (z.B. Libraries) verstehen und nutzen*

➔ *Selbst polymorphen Code entwickeln können, geeignete polymorphe Konstrukte auswählen*

➔ *Eignung von Programmiersprachen betreffend Polymorphie beurteilen können*

Voraussetzungen

Kenntnis zumindest einer

- *streng typisierten*,
- *objektorientierten*

Programmiersprache
(ideal: C++, auch: Java, C#, ...)
(eigener Background: C++)

Konzeptionelles Verständnis von

- **Klassen**
- **Vererbung / Ableitung**

Polymorphie ??? (1)

Griechisch:

"Vielgestaltigkeit"

=

Dasselbe "Ding" kann

je nach "Situation"

verschiedene "Gestalt"

annehmen!

Polymorphie ??? (2)

Bei *streng typisierten* Programmiersprachen:

"Ding": **Funktions- oder Methodenname**,
Rechenzeichen ("**+**", ...)

"Situation": **Typen der Argumentwerte**
im Aufruf

"Gestalt": Für diesen Aufruf
ausgeführter Code

Polymorphie ??? (3)

Das Konzept / die Fähigkeit einer Programmiersprache,

- dass ***derselbe Funktionsaufruf***
- je nach ***Typ der Argumente***
- ***verschiedenen Code*** ausführt.

Polymorphie ??? (4)

Weiter gefasst, nicht beschränkt
auf individuelle Funktionen / Methoden:

Dieselbe Datenstruktur / Klasse
kann typmäßig
verschiedene Gestalt annehmen.

Beispiel:

Eine allgemeine Klasse für verkettete Listen

==> Resultiert in Liste von `int`'s, von `string`'s,
von GUI-Elementen, ...

Arten der Polymorphie

Heute in der Praxis verbreitet:

3 Arten:

1. ***Ad-hoc**-Polymorphie*
2. ***Parametrische** Polymorphie*
3. ***Ableitungs**-Polymorphie*

Sind *getrennte Konzepte*, voneinander *unabhängig*!

(*alle* brauchen *strenge Typen*, *nur 3.* braucht *Objektorientierung*)

Ad-hoc-Polymorphie: Konzept (1)

"Ad-hoc" = Spontan, ungeplant, "von Fall zu Fall"

Besser bekannt unter

Function Overloading

Operator Overloading

- Mehrere, getrennte, *voneinander unabhängige* Definitionen (*Codestücke*)
- mit *verschiedenen*, voneinander unabhängigen *Parametertypen*
- für *denselben Funktionsnamen* / Methodennamen (oder dasselbe Rechenzeichen).

Ad-hoc-Polymorphie: Konzept (2)

Einzige Anforderung an die Definitionen:

Eindeutige Unterscheidbarkeit anhand der *Parameter*:

Für jede mögliche Kombination

von Parametern (betr. Anzahl & Typ)

darf maximal eine Definition passen!

Tücken:

- Der *Returntyp* zählt bei der Unterscheidung **nicht** mit!
- Achtung auf Funktionen mit *Default-Parametern!*
- Achtung auf *automatische Typ-Konvertierung* von Argumenten!

Ad-hoc-Polymorphie: **Beispiele**

- *Rechenzeichen*, z.B. "**+**":
  - Bewirkt unterschiedliche Berechnung
    für `int`, `double`, ev. `string`
  - Ev. auch *selbstdefiniert*: Für Brüche, Vektoren, ...
- *Mehrere Konstruktoren* für dieselbe Klasse
  mit verschiedenen Parametern
- Funktionen mit gewohnter *Standard-Bedeutung*,
  z.B. `length` oder `size`
  für Arrays, Listen, Bäume, Strings, ev. Files, ...

  (nur falls *unabhängig*, *nicht* von gemeinsamer Vaterklasse)

Ad-hoc-Polymorphie: **Anwendung**

Besonders in **C++** üblicherweise *intensiv* genutzt:

- Alle ***Standard-Rechenzeichen***
  (incl. Vergleiche, bool'sche Operatoren, **++** und **−−**, ...),
- die ***Zuweisung*** (incl. Kurzformen **+=**, ...),
- der ***Index-Operator*** **[ ]** und einige mehr

können für beliebige Argument-Typen

selbst definiert

und *beliebig oft überladen* werden!

Ad-hoc-Polymorphie: Nutzen

Für sich allein *kein konzeptioneller Nutzen*
(die Sprache gewinnt nicht an Mächtigkeit),

aber *großer Komfort-Nutzen*:

Weniger, kürzere und intuitivere Namen
(ein + statt 10 verschiedene Funktionsnamen)

- ➔ Leichter *lesbar*
- ➔ Weniger Probleme mit *Namens-Kollisionen*

Wesentlicher *konzeptioneller Nutzen*
in *Kombination mit parametrischer Polymorphie!*

Ad-hoc-Polymorphie: Implementierung

Rein "*statisches*" Konzept (zur Compile-Zeit)

==> *Kein Laufzeit-Overhead* (weder Platz noch Speicher)!

Compiler macht, was man sonst selbst machen müsste:

- Compiler generiert normale, unabhängige Funktionen mit *verschiedenen Funktionsnamen*:

  "Name Mangling"

  ==> Debugger usw. muss Namen "demanglen" !

- Compiler analysiert bei Aufrufen die *Argument-Typen*, sucht die "richtige" Funktion, ruft sie "ganz normal" auf.

Parametrische Polymorphie: Konzept (1)

In C++: ***"Templates"*** In Java: ***"Generics"***

"*Parametrisch*" hat *nichts*
mit den Aufruf-Parameter-Werten der Funktion zu tun!

Sondern:

Ein Stück Code
(= Funktions- oder Klassen-Definition)
hat ein oder mehrere

"Typ-Parameter"

Parametrische Polymorphie: Konzept (2)

Ein **Typ-Parameter** ist …

- ein *Platzhalter* (z.B. "**T**")
- für einen *beliebigen / unbekannten Typ*
- der in diesem Stück Code
  statt eines konkreten Typs verwendet werden kann
  (überall, wo man "`int`" schreiben könnte,
  kann man auch "T" verwenden)
- und beim Aufruf *durch einen konkreten Typ ersetzt* wird
  (entweder *explizit* = durch *Angabe in* **< >**
  oder *implizit* = vom Compiler hergeleitet
  aus den Typen der Argument-Werte)

Parametrische Polymorphie: Beispiel (1)

```
template <typename T>
T min(T a, T b)
{
   T tmp;
   if (a < b) tmp = a;
   else tmp = b;
   return tmp;
}

int data[ARR_SIZE], smallest;
myBruch a, b;
...
smallest = min(data[i], smallest); // implizit int
b = min<myBruch>(a, smallest);
```

Parametrische Polymorphie: Beispiel (2)

==> Ein und dieselbe Definition des Codes

- ist für viele *verschiedene Typen verwendbar*

  (sowohl für *Objekte* als auch für *Basis-Typen*!)

- und macht auch je nach Typ Verschiedenes:
  - Ruft *unterschiedliche Varianten von Funktionen / Operatoren* auf

    (Kombination mit Ad-hoc-Polymorphie!)
  - Legt *Variablen verschiedener Größe* an
  - ...

Parametrische Polymorphie: "Constraints" (1)

Problem:

- *Welche Operationen* gibt es im Template auf T-Werten?
- *Welche Typen* dürfen für T eingesetzt werden?

In C++:

- Template-Code darf "*alles*" mit T-Werten machen!
- Typ-Parameter sind immer "*ganz beliebig*"!

==> Nicht optimal für Lesbarkeit, Dokumentation, ...

==> Compiler-Fehler bei `min` für einen Typ ohne "<":
"Finde kein passendes '<' für Typ **xxx**"

Parametrische Polymorphie: "Constraints" (2)

In Java: Für jeden Typ-Parameter *festlegbar*:

Constraints

(= *Anforderungen, Einschränkungen*)

z.B. bei `min`: `T` muss "`Comparable`" sein ("`<`" haben)!

Beschränkt,

- welche *Operationen* auf `T`-Werten zulässig sind
- und welche *Typen* für `T` zulässig sind.

==> Compiler-Fehler bei `min` für einen Typ ohne "`<`":
"Typ **xxx** *erfüllt Constraint '*`Comparable`*' nicht"*

Parametrische Polymorphie: Anwendung

- Kleine *universelle Hilfsfunktionen*
  (`swap`, `min`, ...)
- *Container-Klassen*
  (Liste, Vektor, Baum, Hashtable, ...)
- Array- bzw. *Container-Algorithmen*
  (`sort`, `find`, ...)

Parametrische Polymorphie: Nutzen

- *Weniger, kürzere und intuitivere Namen*
  ==> Besser lesbarer Code
- ***Spart doppelten Code*** (statt *n* Mal Listen-Code)
  ==> weniger Entwicklungsaufwand
  ==> bessere Wartbarkeit
- Erlaubt ***universelle Algorithmen & Datenstrukturen***, vor allem ***Container*** (sonst nicht universell möglich!)

Vordefinierte Container-Klassen sind
meist qualitativ *hochwertig & effizient*

==> *verwenden!!!*

Parametrische Polymorphie: Implementierung

Statisch zur Compile-Zeit ==> *Kein Overhead* zur Laufzeit

Gedanklich (zumindest in C++, Java ist noch komplexer):

Der Source-Code des Templates / Generics wird

- pro verschiedenem T *ein Mal dupliziert*
- und *mit konkretem Typ* für T eingesetzt *compiliert*
  (für jedes T mit einem neuem, *künstlichen Namen*)

==> Jeder Aufruf ruft "seine" Kopie des Templates
ganz "normal" auf.

==> Ev. *unerwartet großer Binär-Code*
(viele Code-Duplikate!).

Parametrische Polymorphie: Notlösung

Wenn eine Sprache *keine* parametrische Polymorphie hat (z.B Java vor 2004), aber …

alle Klassen eine *gemeinsame Basisklasse* "`Object`" haben:

Ersetzen durch *Vererbungs-Polymorphie*:

Container-Klassen usw.
für ***Elemente der Klasse*** "**`Object`**" schreiben!

Probleme: *Viele!*

- "*Boxing-Problem*": Verwendbar nur für Objekte, nicht für Basis-Typen wie `int`, …
- Teilweise deutlich *langsamer* (Auflösung zur Laufzeit!)

Ableitungs-Polymorphie: Konzept (1)

In einer Hierarchie *voneinander abgeleiteter Klassen*:
Jede Klasse kann (aber muss nicht)

eigenen Code für
geerbte (virtuelle) Methoden definieren.

Es gibt daher

- *verschiedenen Code* (je nach Klasse)
- für *dieselbe Methode*:

  Gleicher Namen und *idente Parameter-Typen*
  (oder zumindest "ableitungskompatible" Parameter-Typen)

Ableitungs-Polymorphie: Konzept (2)

Eine *Variable* (ein Parameter, ein Container-Element, ...),
die *mit der Basisklasse deklariert* ist,

- kann auch ein *Objekt*
  einer ***beliebigen abgeleiteten Klasse*** enthalten,
- und "kennt" alle *Methoden*,
  die ***in der Basisklasse deklariert*** sind.

Beim *Aufruf einer Methode* für diese Variable:

Welcher Code
wird für diesen Aufruf ausgeführt ???

Ableitungs-Polymorphie: Konzept (3)

Bei "*statischer Bindung*" (= Methode *nicht virtuell*):

*Der Code der **"deklarierten" Klasse***
*der Variable (= **Basisklasse**),*

unabhängig von der
tatsächlichen Klasse des Objektes.

- Default in **C++**.
- Meist "*falsch*" (nicht der erwartete Effekt).
- Hier nicht relevant.

Ableitungs-Polymorphie: Konzept (4)

Bei "*dynamischer Bindung*" (*virtuelle* Methode):

Der "am besten passende" Code:

"Suche von der tatsächlichen Klasse des Objektes entlang der Ableitungs-Hierarchie Richtung Basisklasse und nimm die erste Implementierung, die du dabei findest!"

- Default in fast allen anderen Sprachen (incl. **Java**).
- Intuitiv das "*richtige*" Verhalten.
- Ein Fall von *Polymorphie*:

  Je nach tatsächlicher Klasse führt derselbe Aufruf (mit denselben Parametern!) *verschiedenen Code* aus!

Ableitungs-Polymorphie: Beispiel (1)

GUI-Programmierung:

- *Basisklasse* "`DialogElement`" (deklariert u.a.
  virtuelle Methode **`draw`**: *"Zeichne dich selbst"*)
- *Abgeleitete Klassen* für
  Button, Textfeld, Auswahlbox, Fortschrittsbalken, ...
  (jede implementiert `draw` anders!)
- *Klasse für Dialog-Fenster* enthält
  Array aller Dialog-Elemente in diesem `Fenster`
  ==> *Typ laut Deklaration:* Array von `DialogElement`
  ==> *Zur Laufzeit:* Enthält Objekte
  verschiedener abgeleiteter Klassen gemischt!!!

Ableitungs-Polymorphie: Beispiel (2)

Dialog-Fenster will sich komplett neu zeichnen:

- *Schleife* über das Array der Dialog-Elemente
- Aufruf von `draw` *für jedes Array-Element*

==> Muss das "***richtige***" **`draw`**
für das jeweilige Dialogelement
(Button, Textfeld, Auswahlbox, Fortschrittsbalken, ...)
aufrufen!

(in der *Basisklasse* `DialogElement` ist vermutlich
überhaupt *kein Code* für `draw` implementiert!)

Ableitungs-Polymorphie: Nutzen (1)

Virtuelle Methoden sind die
zentrale Errungenschaft
objektorientierter Programmierung.

In vielen Bereichen *nicht mehr wegzudenken* (z.B. GUI):

- *Weniger Namen*
  (z.B. GUI: Wären Dutzende `draw`-Methoden!)
- *Weniger (duplizierter) Code* (==› Aufwand & Wartung!)
- Viel *intuitiverer, kürzerer, besser lesbarer Code!*
- Händischer *Nachbau* wäre *komplex und unleserlich*
  (*Negativ-Beispiel* GTK: Objektorientiertes GUI für C)
  (*früher*: `union`, `switch`-Befehle, Function Pointer, ...)

Ableitungs-Polymorphie: Implementierung (1)

Keine Unterscheidung der Fälle zur Compile-Zeit möglich

==> *Implementierung zur Laufzeit:*

- Jedes *Objekt* verweist auf seine tatsächliche *Klasse*
- Jede *Klasse* enthält eine *Tabelle mit den Code-Adressen* der "richtigen" virtuellen Methoden ("`vtable`")
- Die *Code-Adresse* der "richtigen" Methode wird *zur Laufzeit über das Objekt* ermittelt
- Der Methoden-Aufruf erfolgt mit einem *"indirekten" Call-Maschinenbefehl*

Besonders komplex bei *Mehrfach-Vererbung*!

Ableitungs-Polymorphie: Implementierung (2)

Nachteile:

- *Langsamer*
  (Aufrufe aufwändiger)
- *Braucht zusätzlichen Speicher*
  (`vtable`, Klassen-Pointer in jedem Objekt),

  "versaut" den Cache
  (mehr "nichtlokale" Zugriffe)
- *Verhindert interprozedurale Optimierungen*

Aber ...

Ableitungs-Polymorphie: Nutzen (2)

Essentiell für (nachträglich) *erweiterbaren Code*:

Neue abgeleitete Klassen
und *neue Methoden*-Implementierungen
können ***ohne Änderung am Bestandscode***
eingebunden werden!

- *Keine Neu-Kompilierung*
  bestehender Libraries nötig!
- Laden separat entwickelter *Plugins*
  zur Laufzeit möglich!

Beispiel wie vorher: Nachträglich “Button mit Icon”.

“The end”

Fragen?