Kapitel 3

Tücken der dynamischen Bindung

3.1 `this`-Pointer

Im Methodenaufruf `o.m(x)` ist `o` *Bezugsobjekt*

wird als zusätzlicher Parameter übergeben: `m(o,x)`

kann im Rumpf von `m` als **`this`** (bzw `self`) angesprochen werden

Jeder unqualifizierte Zugriff auf Member im Rumpf ist implizit mit **`this`** qualifiziert:

```
class C {
   T x;
   f(...) {...}
   m() {
     ... x ... f(...) ... m() ...
   }
}
```

entspricht

```
class C {
   T x;
   f(...) {...}
   m(...) {
     ... this.x ... this.f(...) ... this.m() ...
   }
}
```

Achtung: alle Methodenaufrufe mit **this** unterliegen ebenso wie Methodenaufrufe mit gewöhnlichen Pointern der dynamischen Bindung!

Wenn **this** nicht auf ein C-Objekt, sondern auf ein Unterklassenobjekt zeigt, wird dessen (eventuell redefiniertes) f() bzw m() aufgerufen!

Wann kann das passieren? Wenn man ererbte Methode aufruft, die selbst wiederum eine redefinierte Methode aufruft

Bem. **this** zeigt stets auf das sog. C-Subobjekt dieses Unterklassenobjektes, aber dessen Methodentabellen-Verweis zeigt auf die Unterklassen-Methodentabelle!

3.2 Dynamische Bindung und Rekursion

Spezialfall: subtile Interaktion zwischen Rekursion und dynamischer Bindung!

Beispiel:

```
class P {
  int m (int i) {
    System.out.println(i);
    if (i==0) {return i;};
    else { return m(i-1); }
  }
}
class C extends P {
  int m (int i) {
    System.out.println("Start");
    if (i<0) {return 0;};
    else { return super.m(i); }
  }
}
```

Idee: `C` bietet „besseres“ `m` (bessere Termination)

`super`: Zugriff auf verdeckte Oberklassen-Instanzvariablen bzw Methoden

Jedoch:

```
P p = new C();
p.m(42);
```

druckt “Start 42”; rekursiver Aufruf in `P` ruft `C.m`
⇒ “Start” wird nochmals ausgegeben!

Auch in Rekursionen gilt dynamische Bindung!

Lösungversuche:

- `this.m(i-1)` in P funktioniert nicht, da `this` auf ein `C`-Objekt zeigt (s.o.)
- in C++ kann man `P::m(i-1)` in `P` (!) explizit hinschreiben
- Type Cast in Java funktioniert *nicht*:
  `return ((P)this).m(i-1)` in `P` schaltet nicht die dynamische Bindung für `p.m()` ab!

  denn Type Cast ist nur die explizite Version von
  `P q = c; q.m(42);`

Dynamische Bindung kann gefährlich sein!

3.3 Dynamische Bindung und Evolution

gefährlichen Interferenzen zwischen dynamischer Bindung und Versionierung!

Beispiel 1:

```
/* --- Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        ausgabeBetrag += geldeinheit * anzahl;
    }
}
```

```
/* --- Modifiziertes Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    protected long anschaltzeit;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        initialisiere();
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        ausgabeBetrag += geldeinheit * anzahl;
    }
    public void initialisiere() {
        ausgabeBetrag = 0;
        anschaltzeit =
            new java.util.Date().getTime();
    }
}
```

```
/* --- Erbmodul: --- */
import java.util.*;
public class BankomatX extends Bankomat {
        protected int abhebungen;
        private Vector abhebungsZeiten;
        public BankomatX(double einheit) {
                super(einheit);
                initialisiere();
        }
        public void gibGeldeinheit() {
                super.gibGeldeinheit();
                abhebungsZeiten.addElement(new Long (new java.util.Date().getTime()));
        }
        public void gibGeldeinheit(int anzahl) {
                super.gibGeldeinheit(anzahl);
                abhebungsZeiten.addElement(new Long (new java.util.Date().getTime()));
        }
        public void initialisiere() {
                abhebungsZeiten = new Vector();
        }
}
```

Unterklassenkonstruktor ruft Oberklassenkonstruktor; Unterklasse hat Methode `initialisiere`; revidierte Oberklasse auch: im Konstruktor-Rumpf!

⇒ `b = new BankomatX(d)` führt wg. dynamischer Bindung Unterklassen-`initialisiere` aus !!?

Beispiel 2:

```
/* --- Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        ausgabeBetrag += geldeinheit * anzahl;
    }
}
```

```
/* --- Modifiziertes Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        for (int i=0; i<anzahl; i++)
            gibGeldeinheit();
    }
}
```

```
/* --- Erbmodul: --- */
public class BankomatX extends Bankomat {
    protected int abhebungen;
    private static int maxAbhebungen = 3;
    public BankomatX(double einheit) {
        super(einheit);
    }
    public void gibGeldeinheit() {
        if (abhebungen < maxAbhebungen) {
            super.gibGeldeinheit();
            abhebungen += 1;
        }
    }
    public void gibGeldeinheit(int anzahl) {
        if (abhebungen < maxAbhebungen) {
            super.gibGeldeinheit(anzahl);
            abhebungen += 1;
        }
    }
}
```

Eine existierende, redefinierte Methode wird in der revidierten Oberklasse an weiteren Stellen aufgerufen

⇒ revidiertes `gibGeldEinheit(int)` ruft wg. dynamischer Bindung evtl. Unterklassen-`gibGeldEinheit()` auf!!?

Beispiel 3:

```
/* --- Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        ausgabeBetrag += geldeinheit * anzahl;
    }
}
```

```
/* --- Modifiziertes Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        gibGeldeinheit(1);
    }
    public void gibGeldeinheit(int anzahl) {
        ausgabeBetrag += geldeinheit * anzahl;
    }
}
```

```
/* --- Erbmodul: --- */
public class BankomatX extends Bankomat {
    protected int abhebungen;
    private static int maxAbhebungen = 3;
    public BankomatX(double einheit) {
        super(einheit);
    }
    public void gibGeldeinheit() {
        if (abhebungen < maxAbhebungen) {
            super.gibGeldeinheit();
            abhebungen += 1;
        }
    }
    public void gibGeldeinheit(int anzahl) {
        if (abhebungen < maxAbhebungen) {
            for (int i=0; i<anzahl; i++)
                super.gibGeldeinheit();
            abhebungen += 1;
        }
    }
}
```

wg. dyn. Bindung kann Endlosrekursion entstehen: `bankomatX.gibGeldEinheit()`

→`bankomat.gibGeldEinheit()`

→`bankomatX.gibGeldEinheit(1)` (!)

→`bankomat.gibGeldEinheit()`

→ ...

Beispiel 4:

```
/* --- Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        for (int i=0; i<anzahl; i++)
            gibGeldeinheit();
    }
}
```

```
/* --- Modifiziertes Basismodul: --- */
public class Bankomat {
    protected double geldeinheit;
    protected double ausgabeBetrag;
    public Bankomat(double einheit) {
        super();
        geldeinheit = einheit;
        ausgabeBetrag = 0;
    }
    public void gibGeldeinheit() {
        ausgabeBetrag += geldeinheit;
    }
    public void gibGeldeinheit(int anzahl) {
        ausgabeBetrag += geldeinheit * anzahl;
    }
}
```

```
/* --- Erbmodul: --- */
public class BankomatX extends Bankomat{
    private static int maxEinheiten = 20;
    public BankomatX(double einheit) {
        super(einheit);
    }
    public void gibGeldeinheit() {
        if(ausgabeBetrag / geldeinheit < maxEinheiten) {
            super.gibGeldeinheit();
        }
    }
}
```

`BankomatX.gibGeldEinheit(42)` ruft nicht mehr `BankomatX.gibGeldEinheit()` auf!!?

Bem. In C++ gilt innerhalb von Konstruktoren keine dynamische Bindung

[0]Obige Beispiele aus: Pipka/Mezini, Weiterentwicklung objektorientierter Systeme, 2000

3.4 Type Casts

Verwandlung in Oberklassenobjekt immer möglich: Zuweisung ist impliziter Type Cast (↑- Type Cast)

↑- Type Cast selektiert entsprechendes Subobjekt

Objektreferenz bleibt i.a. erhalten (außer bei Mehrfachvererbung)

↑- Type Cast macht verdeckte Oberklassen-Datamembers wieder sichtbar

↑- Type Cast schaltet nicht die dyn. Bindung ab!

```
class O { int x; void f();}
class U extends O { int x; void f();}
O a; U b; a = new O(); b = new U();
i = b.x; // liefert U::x
a = b; i = a.x; // liefert O::x
i = ((O) b).x; // dito
b.f(); // ruft U::f()
a.f(); // ruft U::f() !
((O) b).f() // ruft U::f() !
```

Java: `super.f()` in `U`-Meth.rumpf liefert `O::f()`
↓- Type Cast: nur sinnvoll, wenn Objekt wirklich vom Unterklassentyp ist; sonst Laufzeitfehler: in Java Cast-Exception; in C++ Absturz
⇒ ↓- Type Casts sind unsicher

3.5 Super

```
class O { int  x; void f(){}; }
class U extends O { int x ; void f() {}; }
```

1. Instanzvariablen: `super.x` in `U` liefert `O::x`

   Zugriff: $this + Offset(O :: x)$

2. Methoden: `super.f()` in `U` ruft `O::f()`

   Bezugsobjekt: O-Subobjekt von $this$; Einsprungadresse: statisch bekannt, kein $vptr$-mechanismus nötig (statische Bindung des Aufrufs)

3. Konstruktoren: `super(...)` ruft Oberklassen-Konstruktor der passenden Signatur

   Defaultkonstruktor `new C()` ruft immer Defaultkonstruktor der Oberklasse ($\rightsquigarrow$ Initialisierungen!)

3.6 Statische Variablenbindung

Wieso die Ungleichbehandlung von Variablen und Methoden?
- Methoden werden dynamisch gebunden
- Variablen werden statisch gebunden

Statische Variablenbindung hat sich in der Informatik zu Recht durchgesetzt!

dazu betrachte Pascal-Fragment:

```
procedure p();
  var x: integer;
  procedure q();
  begin writeln(x); end;

  procedure r();
    var x: integer;
  begin x := 17; q(); end:
begin x :=42; r() end;
```

erzeugt Ausgabe *42* und nicht *17*!

in *q* gelten die Deklarationen aus dem statischen Kontext und nicht die der Aufrufstelle!

Vorteil: effizient implementierbar (Frameptr+Offset), statisch typsicher, verständliche Programme

Nun betrachte analoges Java-Fragment:

```
class O {
  int x = 42;
  void q() { System.out.println(x); }
  void r() { x = 103; }
}
class U extends O {
  int x = 17;
  void r() { q(); }
}

O u = new U();
u.r()
```

Es wird *42* ausgegeben, denn *x* wird mit this-pointer von *q* adressiert, der auf O-Subobjekt zeigt! (impliziter Upcast beim Aufruf von *q*)

Vorteil: effizient implementierbar (this-ptr+Offset), statisch typsicher, verständliche Programme

Aber: dynamische Bindung für Variablen im Prinzip denkbar ($\rightsquigarrow$ LISP)

Hingegen wäre Vererbung ohne dynamische Methodenbindung völlig sinnlos, da nicht nutzbar