Object

- alle Klassen direkt oder indirekt von `Object` abgeleitet
- alle Klassen erben Methoden von `Object`:
  - `String toString()`: Umwandlung in Zeichenkette
  - `boolean equals(Object)`: Vergleich auf Gleichheit
  - `int hashCode()`: gleiche Objekte $\Rightarrow$ gleiche Hashwerte
  - `Class getClass()`: Klasse das Objekts (= dyn. Typ)
  - `Object clone()`: Erzeugen einer Kopie
  - `void finalize()`: Aufräumen vor Speicherfreigabe
  - `wait`, `notify`, `notifyAll`: nebenläufige Programmierung

Implizite Umwandlung in Zeichenkette

- ist ein Operand von + eine Zeichenkette, wird der andere Operand in eine Zeichenkette umgewandelt
- bei Referenztypen Umwandlung mittels `toString`:

  ```
  String s = "Prüfung " + b + "! ";
  ```

  entspricht

  ```
  String s = "Prüfung " + b.toString() + "! ";
  ```

- pragmatisch: Überschreiben von `toString` nötig

equals mit instanceof und Cast

- Überschreiben von `equals` notwendig, aber schwierig:

```
public class Scheibe implements AufFlaeche {
    private double x, y, r;
    public boolean equals(Object obj) {
        return obj instanceof Scheibe
               && r == ((Scheibe)obj).r
               && x == ((Scheibe)obj).x
               && y == ((Scheibe)obj).y;
    }
    ...
}
```

- Casts auf Referenztypen sind möglichst zu vermeiden

equals mit super

- Zugriff auf überschriebene Methode durch `super`

```
public class FarbigeScheibe extends Scheibe {
    private long r;
    public boolean equals(Object obj) {
        return obj instanceof FarbigeScheibe
               && super.equals(obj)
               && r == ((FarbigeScheibe)obj).r;
    }
    ...
}
```

equals mit getClass

```
public class Scheibe implements AufFlaeche {
    public boolean equals(Object obj) {
        return this.getClass() == obj.getClass()
               && r == ((Scheibe)obj).r && ... ;
    }
    ...
}
public class FarbigeScheibe extends Scheibe {
    public boolean equals(Object obj) {
        return super.equals(obj)
               && r == ((FarbigeScheibe)obj).r;
    }
    ...
}
```

equals und hashCode

- `a.equals(b)` $\Leftrightarrow$ `b.equals(a)`
- `a.equals(b)` $\Rightarrow$ `a.hashCode() == b.hashCode()`
- `a.equals(b)` $\not\Leftarrow$ `a.hashCode() == b.hashCode()`
- müssen die Bedingungen beim Programmieren beachten

Quellcode als Kommunikationsmedium

- Programmierer kommunizieren über Kommentare
- Qualität der Kommentare entscheidend (nicht Umfang):
  - was man für Verwendung wissen muss
  - keine Beschreibung/Wiederholung der Syntax
- Namen unterstützen die Intuition
- Auffindbarkeit von Information von großer Bedeutung:
  - Faktorisierung spielt wesentliche Rolle
  - wichtigste Informationen an Schnittstellen

```
// Scheibe ist kreisrundes zweidimensionales Objekt auf
// einer Fläche. Es ist auf der Fläche verschiebbar.
public class Scheibe implements AufFlaeche {
    private double x, y;  // die Koordinaten (karthesisch)
    private double r;     // nicht-negativer Scheibenradius

    // Mittelpunkt der Scheibe anfangs auf "ix" und "iy".
    // Absolutwert von "rad" bestimmt Scheibenradius.
    public Scheibe(double ix, double iy, double rad) {...}

    // Verschiebe Scheibe auf Fläche um dx Einheiten
    // nach rechts und dy Einheiten nach oben.
    // Negative Parameterwerte bewirken eine Verschiebung
    // nach links bzw. unten.
    public void verschiebe(double dx, double dy) {...}
}
```

Wo Information zu finden ist

Klasse, Interface: Zweck, Grobstruktur, Besonderheiten

Methoden, Konstruktoren: Parameter, Verhalten, Ergebniswerte, Bedingungen für Aufruf

Objektvariablen: Zweck, Eigenschaften für Konsistenz

Kommentare als Zusicherungen

Vorbedingungen auf Methoden und Konstruktoren:
was vor Aufruf gelten muss, hauptsächlich Parameterwerte

Nachbedingungen auf Methoden und Konstruktoren:
was nachher gilt; was gemacht und zurückgegeben wird

Invarianten auf Variablen, Klassen, Interfaces:
müssen in konsistenten Zuständen stets erfüllt sein

History Constraints auf Variablen, Klassen, Interfaces:
Entwicklung von Objekten im Laufe der Zeit

Gute Faktorisierung

- alle zusammengehörigen Eigenschaften und Aspekte zu übersichtlichen Einheiten zusammengefasst
- Eigenschaften und Aspekte, die nichts miteinander zu tun haben, unabhängig voneinander änderbar
- Problem: Programmänderungen kaum voraussehbar

Klassen-Zusammenhalt

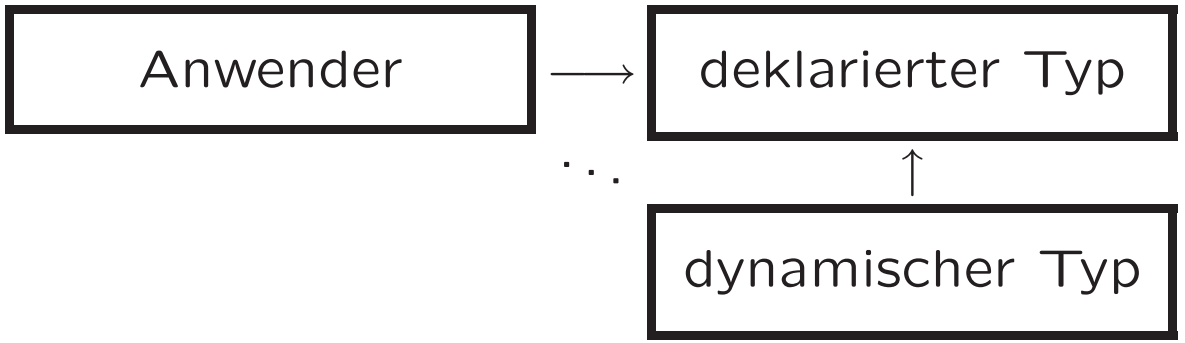
- Grad des Zusammenhangs zwischen Klasseninhalten
- hoher Zusammenhalt = Hinweis auf gute Faktorisierung
- hoher Zusammenhalt erkennbar durch:
  - Variablen und Methoden passen gut zusammen
  - Namen und Kommentare treffend
  - Änderungen verringern Klassenzusammenhalt merklich
- gedankliche Abschätzung schon im Vorhinein möglich

Objekt-Kopplung

- Stärke der Abhängigkeiten der Objekte voneinander
- schwache Kopplung = Hinweis auf lokale Änderbarkeit
- schwache Kopplung erkennbar durch:
  - wenige nach außen sichtbare Methoden und Variablen
  - wenige Nachrichten an andere Objekte
  - geringe Anzahl an Parametern
- gedankliche Abschätzung schon im Vorhinein möglich

Ersetzbarkeit und Verhalten

- Methoden in Unterypen müssen sich so verhalten, wie von Methoden in Obertypen erwartet
- Kommentare in Untertypen spezifizieren dasselbe Verhalten genauer als in Obertypen
- Ersetzbarkeit entkoppelt Programmteile:



- Ableitung von stabilen Typen (weit oben in Typhierarchie)