# Objekt-orientering

- Objekt-orientering är en metodik för att - *rätt använd* - reducera komplexitet i mjukvarusystem.

- Rätt använd ger objekt-orientering stora möjligheter till:
  - Återanvändning av kod
  - Enklare att vidareutveckla system
  - Lättare att underhålla
  - Enkelt att förstå och återanvändas av andra

# Objekt-orienterad modellering

- Ett program är en modell av en *(verklig eller artificiell)* värld.
- I en objekt-orienterad modell består denna värld av en samling objekt som tillsammans löser den givna uppgiften.
- De enskilda objekten har specifika ansvarsområden.
- Varje objekt definierar sitt eget beteende.
- För att fullgöra sin uppgift kan ett objekt behöva support från andra objekt.
- Objekten samarbetar genom att kommunicera med varandra via meddelanden.
- Ett meddelande till ett objekt är en begäran att få en uppgift utförd.

# Objekt-orienterad design

- Designen utgör underlaget för implementationen.
- Bra design minskar kraftigt tidsåtgången för implementationen.
- Brister i designen överförs till implementationen och blir mycket kostsamma att åtgärda.
- Vanligaste misstaget i utvecklingsprojekt är att inte lägga tillräckligt med tid på att ta fram en bra design.
- Bra design är svårt!
- I allmänhet bör mer tid avsättas för design än för implementation.

Hours of planning can save you days of coding

# Symptom på dålig design

- **Stelhet** (rigidity)
  - Svårt att genomföra en ändring pga beroenden med många andra delar av systemet.
- **Bräcklighet** (fragility)
  - En ändring skapar fel i delar av systemet som inte konceptuellt är kopplade till den ändrade modulen.
- **Orörlighet** (immobility)
  - Koden är svår att återanvända i andra applikationer.
- **Seghet** (viscosity)
  - En ”bra” ändring kräver stora insatser, lättare att göra ett ”hack”.
- **Oklarhet** (opacity)
  - En modul är svår att läsa och förstå.

# Motverka dålig design

- Inse från början att kraven kommer att förändras.
  - ”Utveckla för förändring”
- Använd beprövade tekniker och design patterns för att minska beroenden.
  - On the shoulders of giants.
  - Tala samma språk - *lättare för framtida utvecklare att förstå koden*
- Refaktorera kontinuerligt
  - Refactoring är omstrukturering av koden som bevarar funktionalitet men förbättrar struktur

# Symptom på dålig implementation

Synliga tecken i koden att designen är på väg att ruttna:

- Duplicerad kod, ”klipp-och-klistra”-programmering
- Långa metoder
- Långa parameterlistor
- Stora klasser
- Klasser med enbart data
- Publika instansvariabler
- Långa if- eller switch-satser
- Avsaknad av kommentarer
- Onödiga kommentarer
- Oläslig kod

# Abstract class

- En abstract class är en class som inte kan skapa några objekt.
  - Syfte: Att samla gemensam kod för subclasses. *(Undvika kod-duplicering)*
  - Kan *(men måste inte)* ha metoder märkta abstract.
    - Dessa har ingen body och subclasses måste göra override *(om de inte själva är abstract).*
  - Kan ha konstruktorer; dessa kan enbart anropas via `this()` i den egna klassen eller `super()` i subclasses.

# Interface

- Ett interface är en specifikation av ett antal metod-signaturer som tillsammans bildar en typ.
  - Enbart signaturerna ges – alla metoder är helt abstrakta.

- Interfaces i Java får specificera konstanter - attribut som implicit är `public static final`, och som initialiseras till ett konstant värde.
  - Används sällan.
    Grundprincipen för ett interface är en samling metodsignaturer.

# Abstract class vs interface

Använd det som passar avsikten bäst. *Tumregel:*

- En abstract class representerar en generalisering av flera olika subclasses.
  - `Mammal` generaliserar `Cow, Sheep, Cat`
  - `Shape` generaliserar `Rectangle, Triangle, Circle`

- Ett interface representerar en egenskap som kan delas av många (helt) olika classes.
  - `Clonable` representerar alla objekt som kan skapas exakta kopior av
  - `Runnable` representerar alla objekt som kan köras som en separat thread
  - `Moveable` representerar alla objekt som kan förflytta sig

# OCP: The Open-Closed Principle

> Software modules should be open for extension, but closed for modification.

- Vad fyller principen för syfte?
  - Att skriva kod där ny funktionalitet enkelt kan läggas till (open for extension) samtidigt som så mycket som möjligt av existerande kod kan lämnas orörd (closed for modification).

- Varför är det bra att följa den?
  - Koden blir maintainable och extendable.