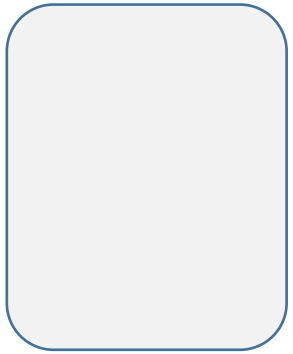
Separation of Concern

Objekt-orienterad programmering och design

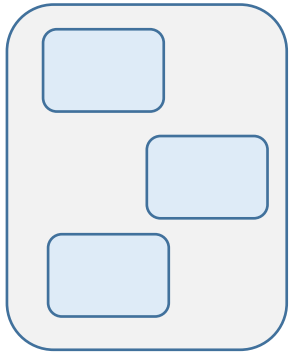
Alex Gerdes, 2018

Modulär design

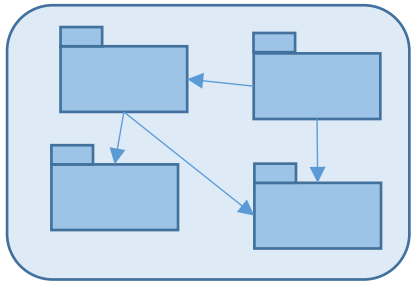
- Ett programsystem är för stort för att kunna förstås i sin helhet. Vi måste bryta ner det i delsystem för att förstå. Denna process kallas *dekomposition*.
- Ett modulärt system är uppdelat i identifierbara abstraktioner på olika nivåer – t ex metoder, classes, delkomponenter, paket, subsystem.



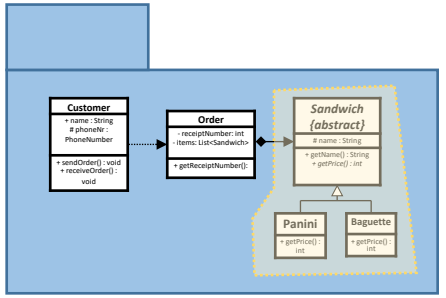
System



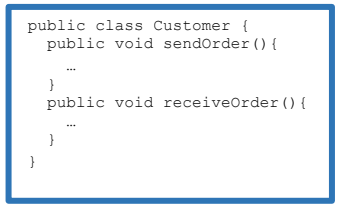
Subsystem



Packages



Delkomponenter
Classes



Metoder

Modulär design

Fördelar med en *välgjord* modulär design:

- Lätt att utvidga
- Moduler går att återanvända
- Uppdelning av ansvar
- Komplexiteten reduceras
- Moduler går att byta ut
- Tillåter parallell utveckling

Återanvändning

- High cohesion och Low coupling lägger grunden för återanvändning av komponenter.

- Ju mer välspecificerad uppgift en modul har, och ju enklare dess gränssnitt och implementation är, desto större chans är det att subsystemet kan att återanvändas i andra sammanhang.
  - Gäller på alla nivåer: subsystem, paket, classes, metoder.

Functional decomposition

- På metod-nivå kan vi tillämpa *functional decomposition* (”funktionell nedbrytning”) för att reducera komplexiteten:
  - Enklare del-beräkningar eller -beteenden bryts ut till egna metoder som anropas från den ursprungliga metoden med rätt argument.
  - Ger flera fördelar:
    - Reducerar den kognitiva komplexiteten i varje (del-)metod.
    - Möjliggör code reuse för del-beräkningar och -beteenden som återkommer.
    - Sätter namn på del-beräkningar och -beteenden, vilket ger högre grad av abstraktion och förståelse.
    - Följer OCP: Större chans att framtida förändringar kan återanvända och utöka, istället för att behöva ändra.

Example: Functional decomposition

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money amount = e.calcuatePay();
      e.deliverPay(amount);
    }
  }
}
```

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void
  calculateAndDeliverPay(Employee e) {
    Money amount = e.calcuatePay();
    e.deliverPay(amount);
}
```

Mer kod är bättre?

- Exemplet till höger innehåller fler rader kod – är inte det högre komplexitet?
- Svar Nej!
  - Exemplet till höger innehåller lika många rader kod, med undantag för anrop av abstraherade metoder.
  - Exemplet till höger innehåller fler metod-signaturer, vilka (rätt gjorda) hjälper förståelsen och reducerar komplexiteten.
  - Varje enskild metod är mindre komplex.
  - Sannolikheten att kod kan återanvändas är mycket högre. Högre grad av maintainability.

Övning

- Börja från koden som finns att ladda ner från hemsidan.
- Vår gamla kod för `DrawPolygons` (oförändrad från förra gången) ligger nu i ett paket `tda551.polygons`. Det finns också ett till paket `tda551.shapes` som ger en alternativ implementation av polygoner med mer funktionalitet. I förlängningen (inte idag) är det tänkt att vi ska byta ut vår naiva hantering mot den mer kraftfulla – men båda två har problem som behöver lösas först.
- På metod-nivå: Titta på metoderna i de olika klasserna, i båda paketen. Vilka metoder har ett väldefinierat och väl avgränsat ansvarsområde? Vilka metoder gör mer än en sak? Vilka metoder gör överlappande saker (kod-duplicering)?
  - Tillämpa funktionell nedbrytning och refactoring för att stegvis lösa problemen.
- På class-nivå: Titta på de olika klasserna, i båda paketen. Vilka klasser har ett väldefinierat och avgränsat ansvarsområde? Vilka klasser gör mer än en sak?
  - De klasser som ni tycker gör en sak – ge en definition av vad denna enda sak är. Hur ”abstrakt” är er definition – dvs hur mycket av funktionaliteten täcker definitionen (inte)?
- På paket-nivå (för mer utmaning): Paketet `tda551.shapes` är tänkt att vara en väl avgränsad (dock inte väl implementerad ännu) implementation av (Swing-)polygoner. Fundera över vilka delar av av `tda551.polygons` som skulle behöva bytas ut mot detta paket. Vilka problem ser ni som ligger i vägen för ett sådant byte? Vad hade vi förutseende kunnat göra med `tda551.polygons` för att göra ett sådant framtida byte enklare?
  - I enlighet med OCP: förutsäg förändring, och möjliggör extension istället för modification.

- Fun quiz (orelaterat): I paketet `tda551.quiz` finns en class `Mystery`. Vad är dess syfte?