Modulär design

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Vad är ett “bra” program?

- I kursen pratar vi om ”bra” kod utifrån ett utvecklar-perspektiv, dvs det som gör koden lätt att (vidare-)utveckla.

- Reusability
- Extensibility
- Maintainability
  - Enkelt, läsbart
  - Testbart
  - Robust (vid förändringar)

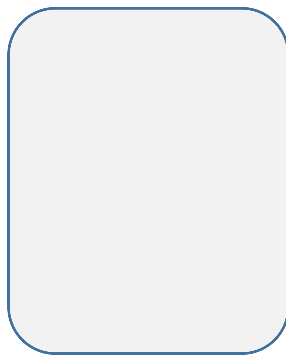
Separation of Concern principle

Do one thing – do it well.

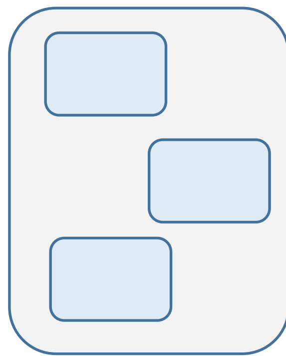
- Separation of Concern är inget specifikt för objekt-orientering, utan en generell maxim för all programmering.
- Termen ursprungligen från Edsger W. Dijkstra, som syftade på det systematiska användandet av *abstraktion*:
  - Betrakta en sak i taget, och håll alla andra så abstrakta som möjligt, för att bättre förstå och bestämma hur denna enda sak bör hanteras.

Modulär design

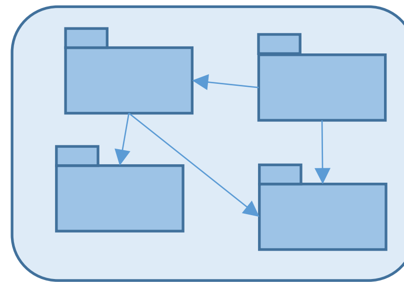
- Ett system som uppfyller Separation of Concern kallas *modulärt*:
  - Ett programsystem är för stort för att kunna förstås i sin helhet. Vi måste bryta ner det i delsystem för att förstå. Denna process kallas *dekomposition*.
  - Ett modulärt system är uppdelat i identifierbara abstraktioner på olika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



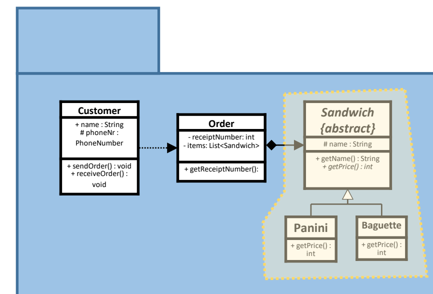
System



Subsystem



Packages



Delkomponenter
Classes

```
public class Customer {
  public void sendOrder(){
     …
  }
  public void receiveOrder(){
     …
  }
}
```

Metoder

Modulär design

Fördelar med en *välgjord* modulär design:

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

Modularitet

- Ju mer välspecificerad uppgift en komponent* har, och ju enklare dess gränssnitt och implementation är, desto större chans är det att komponenten kan:
  - Återanvändas i andra sammanhang.
  - Utökas med mer funktionalitet i andra komponenter
  - Läsas och förstås lätt
  - Testas isolerat från andra moduler
  - Motstå förändringar gjorda i andra moduler (robustness)

- * Komponent = metod, class, package, … - principerna gäller på alla nivåer

A word of moderation

- Ju större projekt – ju fler inblandade personer, ju fler rader kod, ju fler delsystem, ju längre livslängd etc – desto större behov av Separation of Concern, och allt det medför.
- Det omvända gäller förvisso också: ju mindre projekt, desto mindre blir kostnaden av att ”strunta i” designprinciper.
- Att följa designprinciper, inklusive Separation of Concern, *kan* lägga på overhead vars kostnad för små projekt överstiger vinsten.
- Rule of thumb: Använd alltid Separation of Concern som default – låt alla avsteg från principen vara medvetna!

Modularitet på olika nivåer

- Roadmap för dagens föreläsning:
  - Modularitet för metoder
    - Dekomposition (uppdelning) med avseende på specifik uppgift
    - Functional decomposition
  - Modularitet för classes
    - Uppdelning med avseende på ansvar
    - Single Responsibility Principle
  - Modularitet för paket och sub-system
    - Uppdelning med avseende på aspekter
    - Modell vs kontroll

Modularitet för metoder

- På metod-nivå tillämpar vi *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, väl utfört:
    - Reducerar den kognitiva komplexiteten i varje (del-)metod.
    - Möjliggör code reuse för del-beräkningar och -beteenden som återkommer.
    - Möjliggör separat testning av del-beräkningar och -beteenden.
    - 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.

Live code

```
Rectangle.getCorners()
```

Abstraktion kräver bra namn

- För variabler, använd namn som specificerar vad de representerar.
  - Typiskt substantiv
  - `centerPoint` isf `xy`, `rotation` isf `degrees`

- För metoder, använd namn som indikerar deras syfte och beteende.
  - Typiskt verb eller verb-fraser
  - `sort`, `print`, `add`, `getRotation`, `isEmpty`

Live code

```
Mystery.x()
```

Sidoeffekter

- En sidoeffekt av en metod är varje modifikation av data när metoden anropas, som är *observerbar* utanför metoden.

- Om en metod inte har några sidoeffekter kan man anropa den hur många gånger som helst och alltid få samma svar (såvida ingen annan metod med sidoeffekter har anropats under tiden).

- En metod som inte har några sidoeffekter kallas *pure*.
  - (Haskell har purity som default, sidoeffekter enbart i `IO` (typ).)

Quiz

```
A a = new A();
int r = a.method(1) - a.method(1);
```

- Vad är värdet på `r`?
- Svar: Utan purity, ingen aning!

Principer för sidoeffekter

- Grundkonventionen i objekt-orienterade språk (konvention, inte regel) som Java är att metoder får ha sidoeffekter.
- För att undvika förvirring krävs att vi upprätthåller vissa principer:
  - Sidoeffekter ska vara väl deklarerade och intuitivt lätta att förstå, utifrån en metods namn, signatur, dokumentation, och kontext.
  - Generellt är det acceptabelt att metoder uppdaterar tillståndet (attribut) hos sitt implicita argument, dvs objektet som metoden anropas på.
  - En metod bör *inte* förändra andra objekt – de explicita argumenten, eller tillgängliga statiska variabler (”global state” – mer senare).

```
myList.addAll(otherList);
```

  - Vi förväntar oss att `otherList` ska vara oförändrad efter anropet, men att `myList` har ändrats.

The Command-Query Separation Principle

> En metod ska antingen ha sido-effecter (en *mutator*) eller returnera ett värde (en *accessor*), inte båda.

- En metod som returnerar information om ett objekt ska inte ändra tillståndet hos objektet.
- En metod som ändrar tillståndet hos ett objekt ska – som regel – inte samtidigt returnera information om objektet.

Quiz

- Nämn några metoder från Javas standardbibliotek som bryter mot Command-Query Separation Principle.

- Svar: `Iterator.next()`, `Stack.pop()`, …

- Om principen bryts – se till att det görs av god anledning, och är mycket väldokumenterat och intuitivt lätt att förstå.
- Om en mutator också returnerar information, se till att det finns en ren accessor som kan returnera samma information utan sidoeffekt.
  - E.g. `Stack.peek()`

God abstraktion för metoder

- Väl utförd functional decomposition hänger på bra abstraktion:
  - Bra namngivning
  - Inga överraskningar
  - Inga oönskade sidoeffekter

- Om (och endast om) vi har bra utförd abstraktion kan vi uppnå fördelarna med modularitet:
  - Reducerad komplexitet, lätt att förstå, lätt att testa, …

Modularitet för classes

- Vi säger att en class ska ha ett väl avgränsat ansvarsområde. Vad innebär det?

- Hur definierar vi konkret ett ansvarsområde?

Live code

- `Tda551.shapes: DrawPolygons, Triangle, Rectangle`

Single Responsibility Principle

A class should have only one reason to change.
(Robert C. Martin)

- Robert C. Martin (”Uncle Bob”) är en av författarna till The Agile Manifesto.
  - SRP är en central aspekt av Agile software development
  - "Gather together the things that change for the same reasons. Separate those things that change for different reasons.” ← alternativ formulering, Robert C. Martin igen.

Varför SRP?

- Att ha classes som följer SRP ger många fördelar (det börjar bli lite upprepning...):
  - Robusthet: Om funktionalitet behöver förändras minskar risken för sammanblandning, och förändringar kan hållas väl avgränsade.
  - Testning: En class med ett väl avgränsat ansvarsområde är lättare att testa separat från annan funktionalitet.
  - Easy access: All kod som rör ett visst ansvarsområde kan hittas på ett enda ställe
    - Bra namngivning av classes gör detta lättare.

Live code

- Fundera `DrawPolygons`

Modularitet för paket/subsystem

- Samma principer som gäller för komponenter på lägre nivåer gäller även för större delar.
- Vi delar upp med avseende på *aspekt* eller *funktion*. E.g.:
  - Datamodellen bör inte blandas samman med kontrollen.
  - Modeller för olika data-aspekter bör modelleras separat från varandra.
  - Olika topp-nivå-beteende (”applikationer”) bör inte bo tillsammans.

Java packages

- Språk-feature för att definiera väl avgränsade komponenter på högre nivå än classes.
  - Lägg filerna som hör ihop i en separat mapp, med samma namn som paketet.
  - Lägg till en *package declaration* högst upp i filerna:

```
package tda551.shapes;
```

  - I andra filer som ska använda paketet, lägg till *import statements:*

```
import tda551.shapes.*;
```

  - Obs! `shapes` är inte ett package, `tda551.shapes` är. Även i andra subpackages till `tda551` måste hela sökvägen anges.

Point

JComponent

JFrame

ArrayList

Polygon
<<abstract>>

+ drawer : DrawPolygons
+ centerPoint : Point

+ paint(Graphics) : void
+ updateCenter(int,int) : void

DrawPolygons

+ polygons : ArrayList<Polygon>
+ frame : JFrame
+ ticker : int
+ direction : boolean

+ paint(Graphics) : void
+ update() : void
+ main(String[]) : void

Rectangle

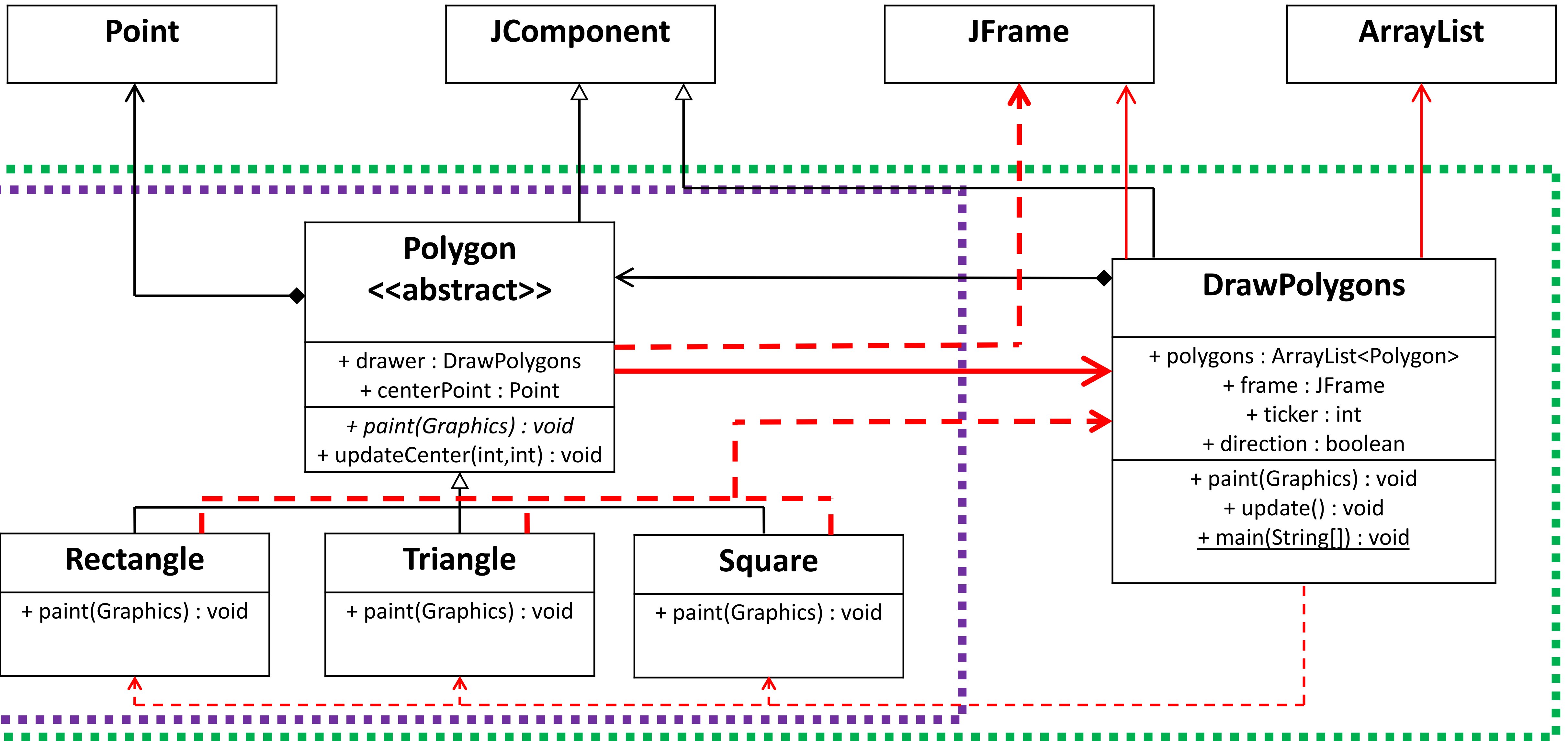
+ paint(Graphics) : void

Triangle

+ paint(Graphics) : void

Square

+ paint(Graphics) : void


Point
JComponent
JFrame
ArrayList
Polygon
<<abstract>>
+ drawer : DrawPolygons
+ centerPoint : Point
+ paint(Graphics) : void
+ updateCenter(int,int) : void
DrawPolygons
+ polygons : ArrayList<Polygon>
+ frame : JFrame
+ ticker : int
+ direction : boolean
+ paint(Graphics) : void
+ update() : void
+ main(String[]) : void
Rectangle
+ paint(Graphics) : void
Triangle
+ paint(Graphics) : void
Square
+ paint(Graphics) : void

Live code

- `tda551.polygons`

Sammanfattning: Modulär design

- Väl utförd modularisering – dvs uppdelning i separata och väl avgränsade komponenter – ger många fördelar:
  - Reusability
  - Extensibility
  - Maintainability
    - Enkelt, läsbart, överblickbart
    - Testbart
    - Robust (vid förändringar)

  - Precis de aspekter vi är ute efter i den här kursen!

Cliffhanger

- Nu har vi `tda551.polygons` som ett eget subpackage – vi har löst de ”utåtgående” beroendena. Vi har dock fortfarande ”inåtgående” beroenden på interna implementationsdetaljer, som gör det svårt att byta ut `tda551.polygons` mot – till exempel – `tda551.shapes`.

- Hur kan vi göra `tda551.polygons` bättre avgränsad?
  - Väl-definierat gränssnitt till hela sub-paketet.
  - Väl genomtänkta kopplingspunkter.
  - Väl inkapslad (encapsulated) intern implementation och representation.

What’s next

Block 5-1:
Introduktion till Design patterns