Dependencies – High cohesion, low coupling

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Vad är ett ”bra” program?

- Korrekt?
- Effektivt?
- Användbart?
- Flexibelt?
- Robust?
- Skalbart?

Externa faktorer
(berör användare)

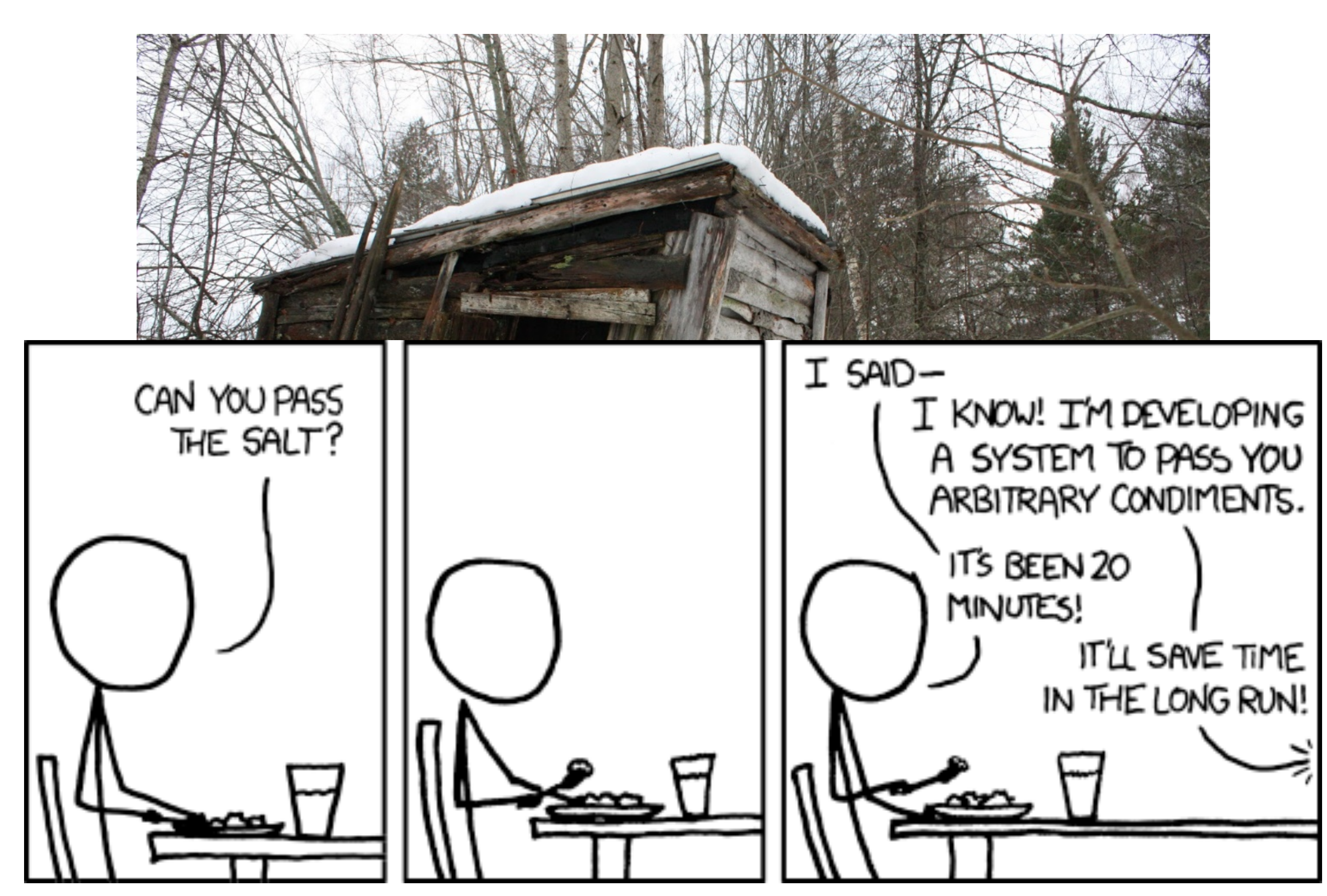
- Enkelt?
- Läsbart?
- Testbart?
- Maintainable?
- Reusable?
- Extensible?

Interna faktorer
(berör utvecklare)

Fokus för denna kurs!

Småskalig programmering

- Triviala program
- Få klasser
- Några 100-tal rader kod
- En eller ett fåtal programmerare
- Ingen eller kort livstid
- ”Just do it”



Storskalig programmering

- Mycket komplexa programsystem
- Flera miljoner rader kod
- 100-tals programmerare, geografiskt utspridda
- Lång livstid
- ”Software engineering”
  - Behov av verktyg
  - Behov av processer

Objekt-orientering

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

- Rätt använd ger objekt-orientering stora möjligheter till:
  - Code reuse
  - Extensibility
  - Easier maintenance

- Fel använd riskerar objekt-orientering att skapa extra komplexitet
  - ”Big ball of mud”
  - Det finns mycket dålig kod därute…
  - Det finns väldigt många missförstånd kring objekt-orientering.

Design

- Designen (modellen) 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.

Maintenance (underhåll)

All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version. (Ivar Jacobson)

- Fel måste fixas.
- Mjukvara måste uppdateras till nya användarkrav.
- Den som ändrar koden är sällan den som ursprungligen skrev den.
- Största delen (~80%) av pengarna och tiden under ett systems livstid går till underhåll.

Utveckla för förändring!

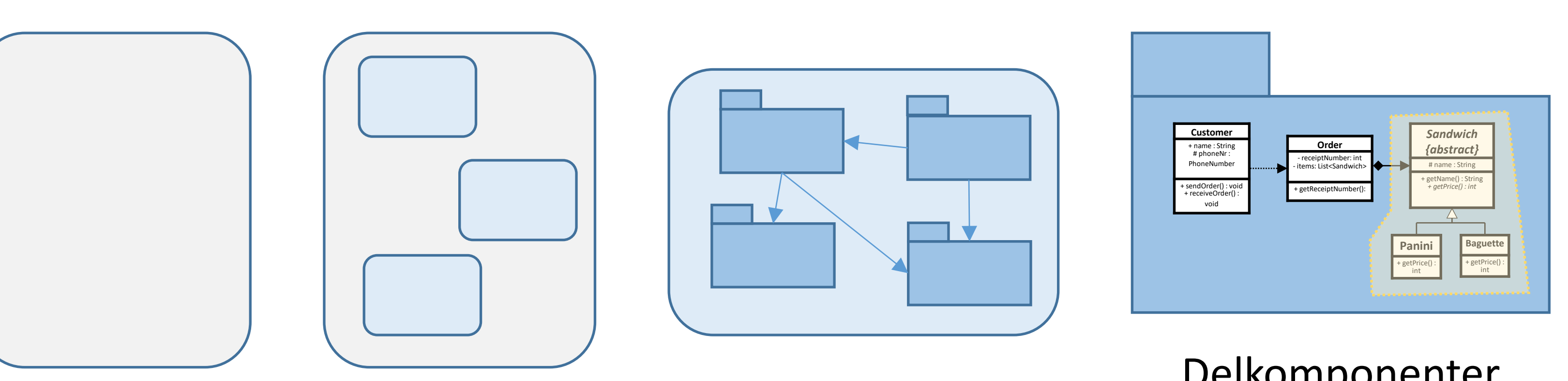
OCP: The Open-Closed Principle

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

- Vårt mål är reusability, extensibility, maintainability.
- OCP är kärnan i vår metodik för att uppnå detta.

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 classes, delkomponenter, paket, subsystem.




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

System

Subsystem

Packages

Delkomponenter
Classes

Metoder

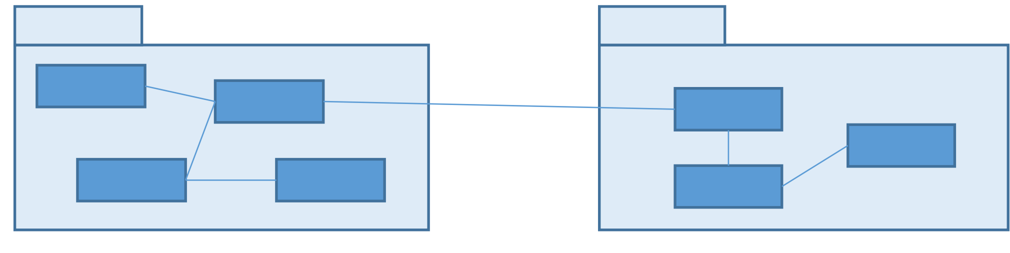
Modulär design

Fördelar med en *välgjord* modulär design:

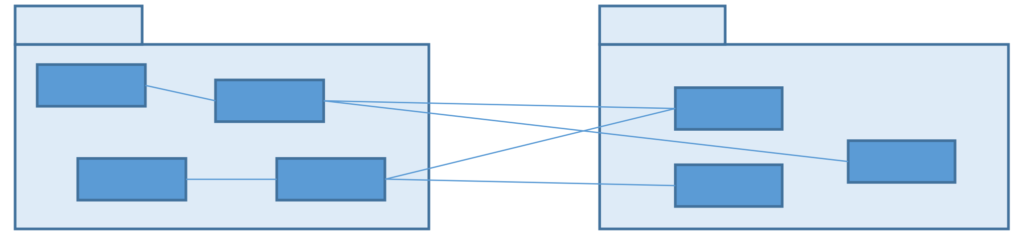
- Lätt att utvidga
- Moduler går att återanvända
- Uppdelning av ansvar
- Komplexiteten reduceras
- Moduler går att byta ut
- Tillåter parallell utveckling.

Cohesion

- *Cohesion* (”sammanhållning”) är ett mått på den inre sammanhållningen i en modul (e.g. metod, class, package, …).
  - Hur väl samverkar komponenterna inom modulen?
- Vi eftersträvar *high cohesion* – dvs när samtliga komponenter samverkar för att lösa modulens ansvarsområde, utan att behöva samverka med komponenter i andra moduler.
  - Hur autonom är modulen? Klarar den sitt uppdrag på egen hand?
  - Hur väldefinierat är modulens ansvarsområde?



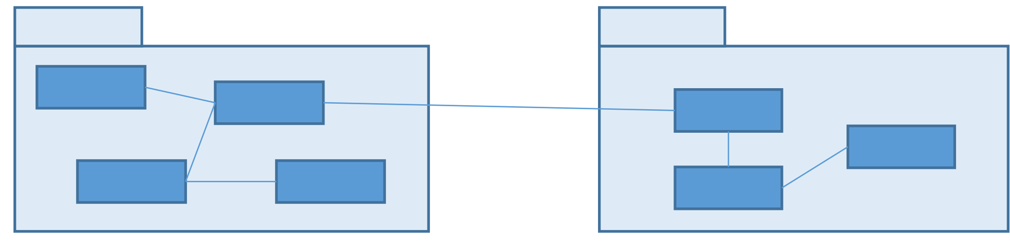
High cohesion



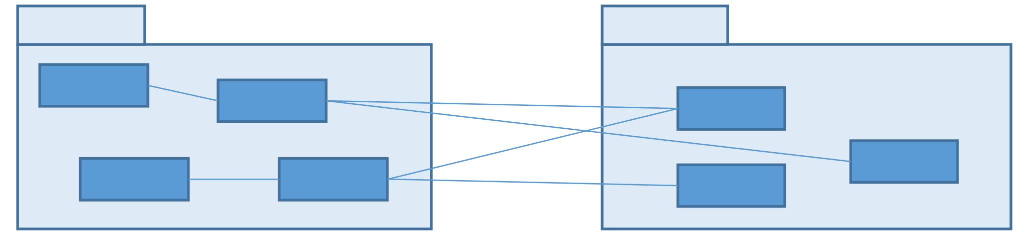
Low cohesion

Coupling

- *Coupling* (”sammanbindning”, ”koppling”) är ett mått på hur starkt beroendet är mellan två olika moduler.
  - Hur autonom är modulen? Klarar den sitt uppdrag på egen hand?
  - Hur väl avgränsad är modulen? Tillåter den andra moduler att bero på dess inre implementation?
- För att få en flexibel och modulär design måste ingående moduler vara så oberoende av varandra som möjligt. Vi eftersträvar *low coupling* mellan moduler.
  - Har vi starka kopplingar kommer förändringar i en modul framtvinga förändringar i andra moduler som är beroende av den – bryter mot OCP.



Low coupling



High coupling

Återanvändning

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

- Ju mer välspecificerad uppgift ett subsystem har, och ju enklare dess gränssnitt är, desto större chans är det att subsystemet kan att återanvändas i andra sammanhang.

Dependency Inversion Principle

Depend on abstractions, not on concrete implementations.

- Beroenden behövs! Men för en bra design gäller:
  - Minimera antalet beroenden.
  - Ha så lösa beroenden som möjligt (DIP).
  - Ha kontroll över beroendena.
    - Varje beroende ska vara avsiktligt.
    - Varje kopplingspunkt (dvs möjlighet att skapa beroenden) ska vara avsiktlig.
    - Varje kopplingspunkt ska ha ett väldefinierat gränssnitt.

Abstraktion

> **[A]bstractions** may be formed by reducing the information content of a concept […], selecting only the aspects which are relevant for a particular purpose.
>
> Wikipedia

- Abstraktion för våra syften handlar om att exponera så lite information som möjligt, för att göra beroenden så lösa som möjligt.
- Ju mindre en class (package, …) exponerar (metoder, attribut,…), desto mindre finns det för andra att bero på.

Encapsulation

> **Encapsulation** is the packing of data and functions into a single component. […] It allows selective hiding of properties and methods.
>
> Wikipedia

- Encapsulation (”inkapsling”) är en specifik språkmekanism i e.g. Java som hjälper oss att åstadkomma abstraktion.
  - Notera: Encapsulation är vårt verktyg - abstraktion vårt (del-)mål.

Encapsulation i Java

- Fyra *access modifiers*:
  - `public` – tillgänglig för alla som vill
  - `private` – tillgänglig enbart i den egna klassen (inte ens subklasser)
  - *package private* – tillgänglig för alla inom samma package
  - `protected` – tillgänglig för alla inom samma package, samt subklasser till den egna klassen (även i andra packages)

- För classes:
  - `public` eller *package private* (default)
- För metoder och attribut:
  - `public`, `protected`, *package private* (default), `private`

UML

- Unified Modelling Language
  - Grafiskt modelleringsspråk för att beskriva olika aspekter av objekt-orienterade system.

- Ett hjälpmedel för att:
  - Kommunicera design.
  - Förstå vilka beroenden ett system innehåller eller möjliggör.
  - Upptäcka problem på en högre nivå än att försöka förstå ogenomtränglig spaghetti-kod.

Klassdiagram: Relationer

- Två grundläggande typer av relationer mellan classes och interface, med vardera två ”styrkor”:
  - Förhållande mellan subclass (subinterface) och superclass (superinterface):
    - Generalisering (Is-A)
      - En class (interface) är en direkt subclass (subinterface) till en annan class (interface).
    - Realisering (implements)
      - En class implementerar ett interface.
  - Förhållande som anger grad av kännedom om annan class eller interface:
    - Association (Has-A)
      - En class har attribut (fields) som håller objekt av en annan class (interface).
    - Beroende (usage dependency)
      - En class (interface) använder eller skapar objekt av en annan class (interface).

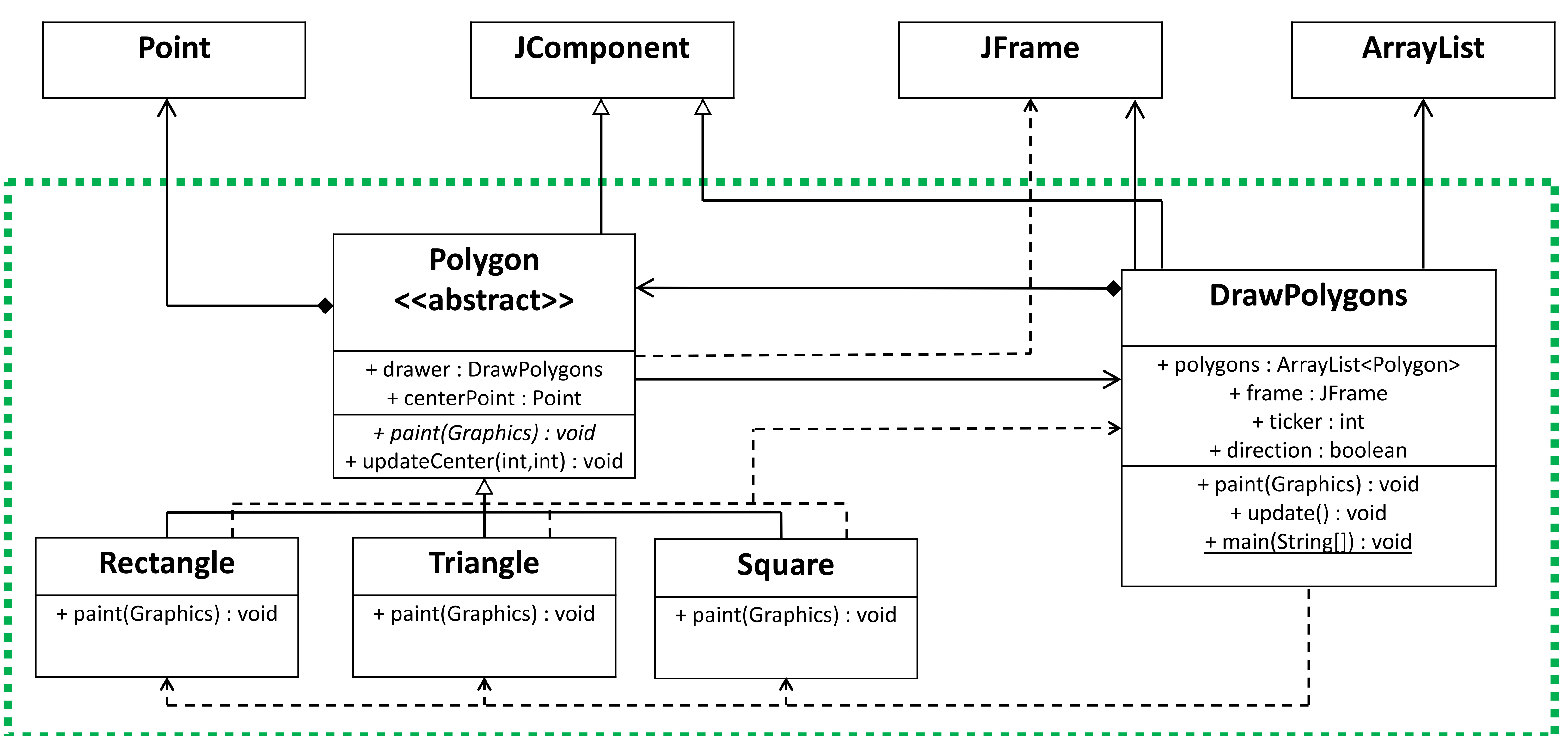
Live code

- …

Beroenden?

- Vilka beroenden finns i vår ”design”?

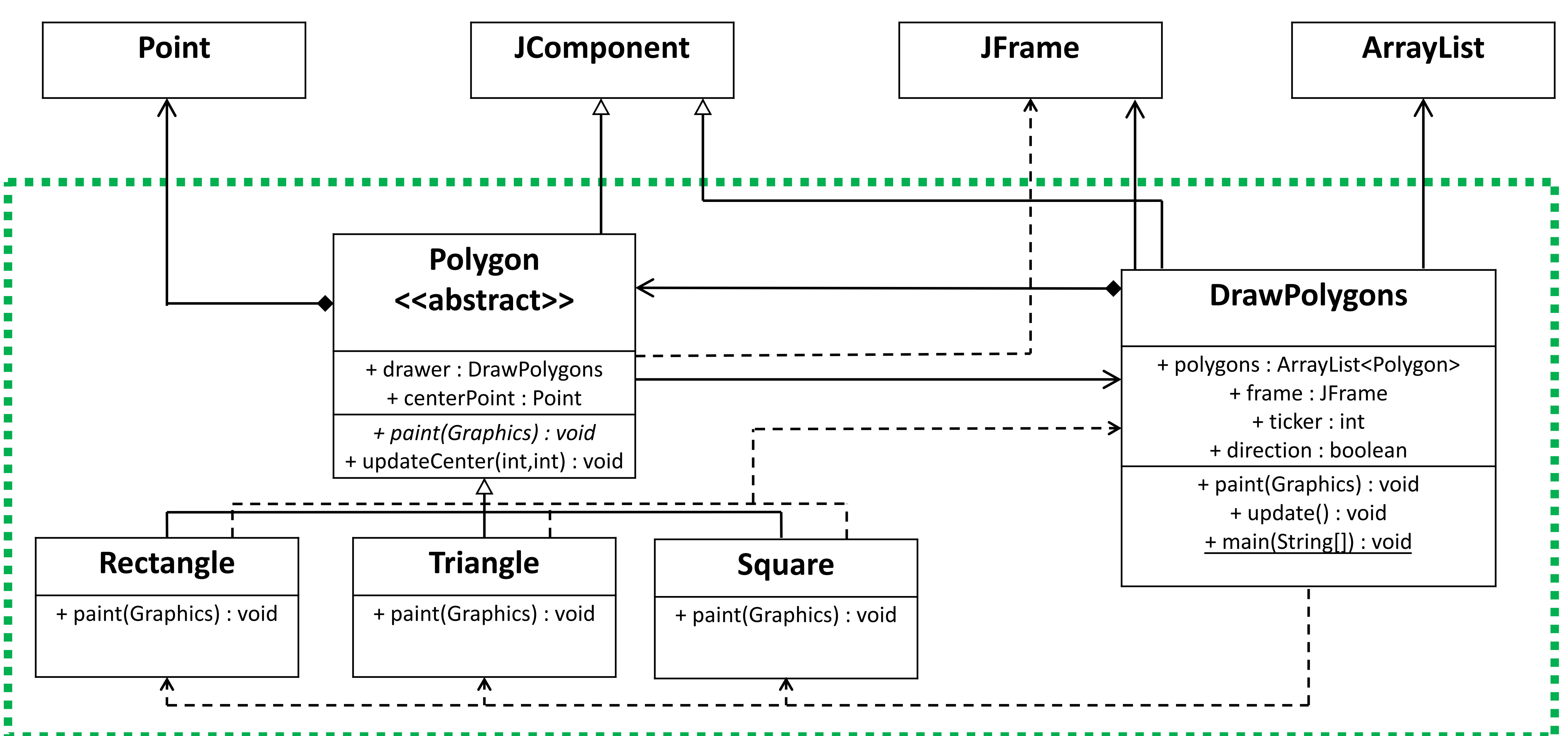
- Steg ett: Rita UML


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

Low coupling?

- Vår ”design” har löjligt många, och fullständigt ogenomtänkta, beroenden.

- Steg två: fundera över vilka beroenden som är rimliga och inte.


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

Mutual dependencies

- Beroenden mellan två klasser åt båda håll är (nästan) aldrig rimligt!
  - Starka beroenden (association) åt båda håll är extra dåligt.
  - Om A beror av B och B beror av A så är de i praktiken en enhet.
  - Det finns specialfall då detta är rimligt – men det ska i så fall designas mycket medvetet.

- Slutsats: DrawPolygons och Polygon bör inte båda bero av varandra.

Data model vs controller

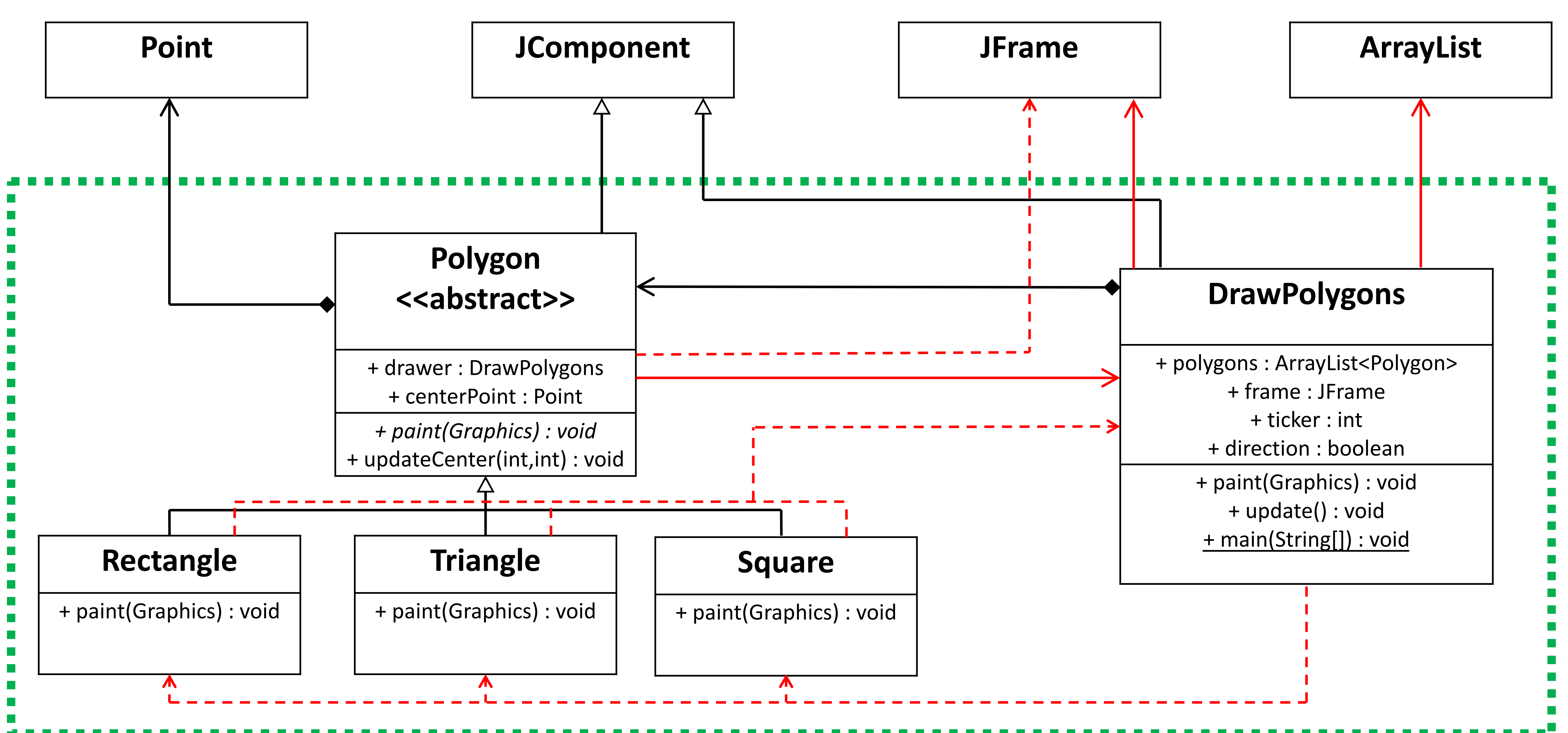
- Vi har i princip alltid klasser som representerar vår ”datamodell”, och klasser som hanterar denna modell.
- Klasserna som representerar data bör vara självständiga.
  - Ger möjlighet till återanvändning i andra sammanhang.
  - (Mycket mer om Model-View-Controller pattern nästa vecka.)

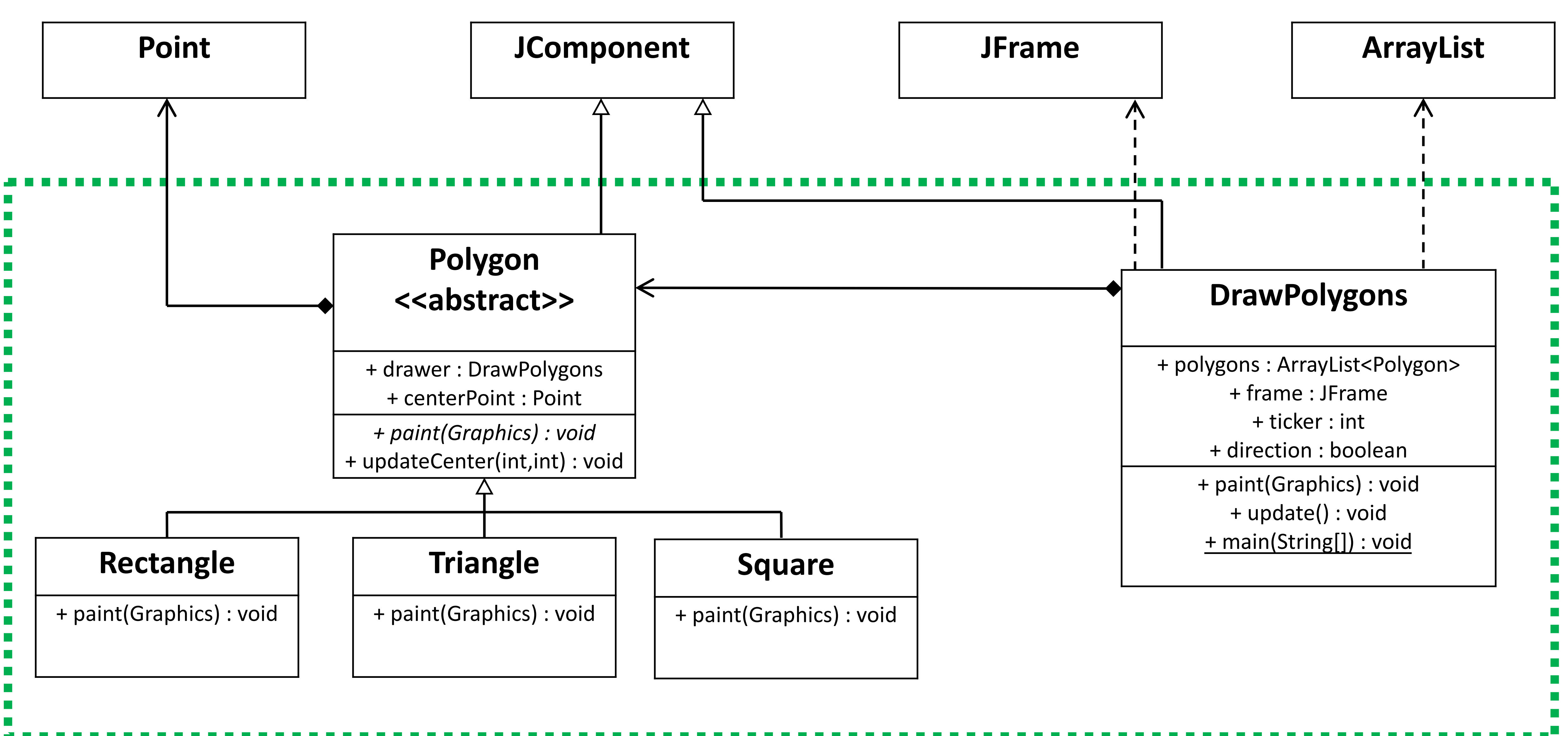
- Slutsats: Polygon bör inte bero av, eller ens känna till, DrawPolygons. Inte heller bör subclasses av Polygon göra det.

DIP: Dependency Inversion Principle

- Bero på abstraktioner, inte på konkreta implementationer.

- Slutsatser:
  - DrawPolygons bör inte direkt bero på ArrayList.
  - DrawPolygons bör inte vara starkt associerat med JFrame.
  - DrawPolygons bör inte bero på konkreta subclasses till Polygon.


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

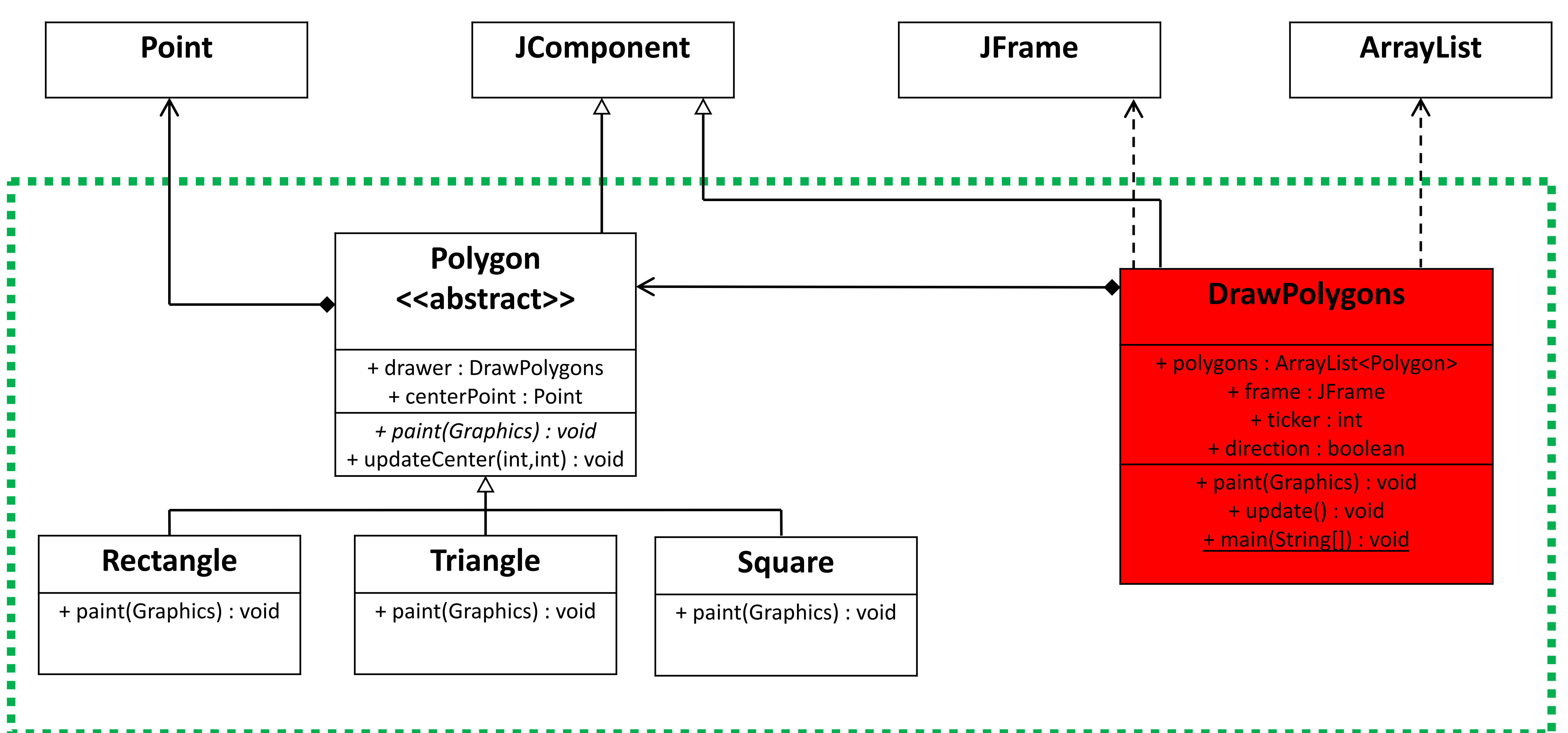
- …

Minska beroenden

- Steg 3: Fundera över *hur* beroenden kan minskas.
- Detta är vad vi kommer spendera kommande veckor på!

Ansvar

- I en bra design vill vi att varje del (paket, class, metod) har ett väl avgränsat ansvarsområde.
  - High cohesion.
- Mer om Single Responsibility Principle och Separation of Concern Principle nästa vecka.
- Fråga: Har DrawPolygons ett väl avgränsat ansvarsområde?


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

Quotes

Any intelligent fool can make things bigger and more complex. It takes a touch of genius and a lot of courage to move in the opposite direction. (*Albert Einstein*)

Simple can be harder than complex. You have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains. (*Steve Jobs*)

What’s next

Block 4-2:

Separation of Concern,
introduktion till Design Patterns