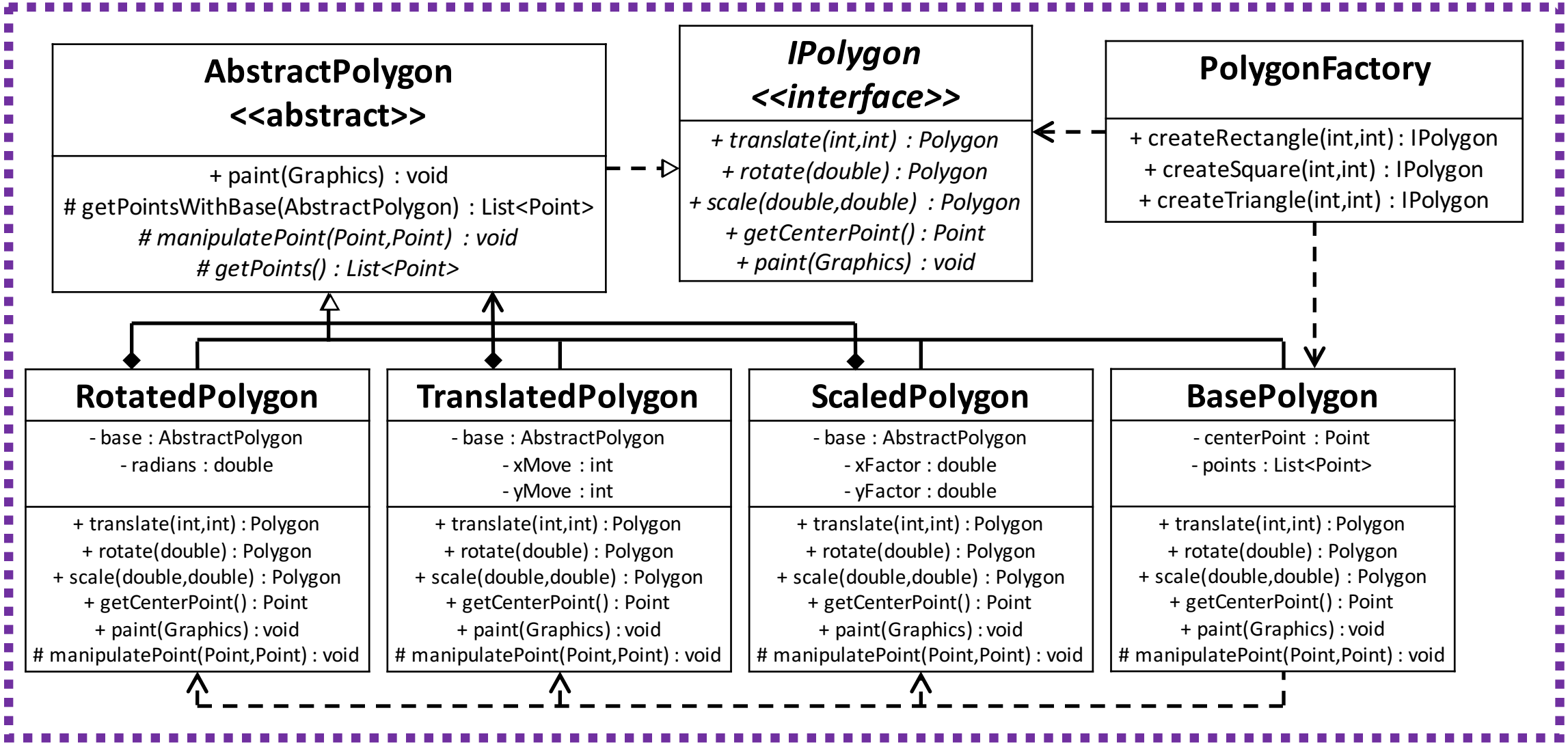
Principer, Patterns och Tekniker

Objekt-orienterad programmering och design

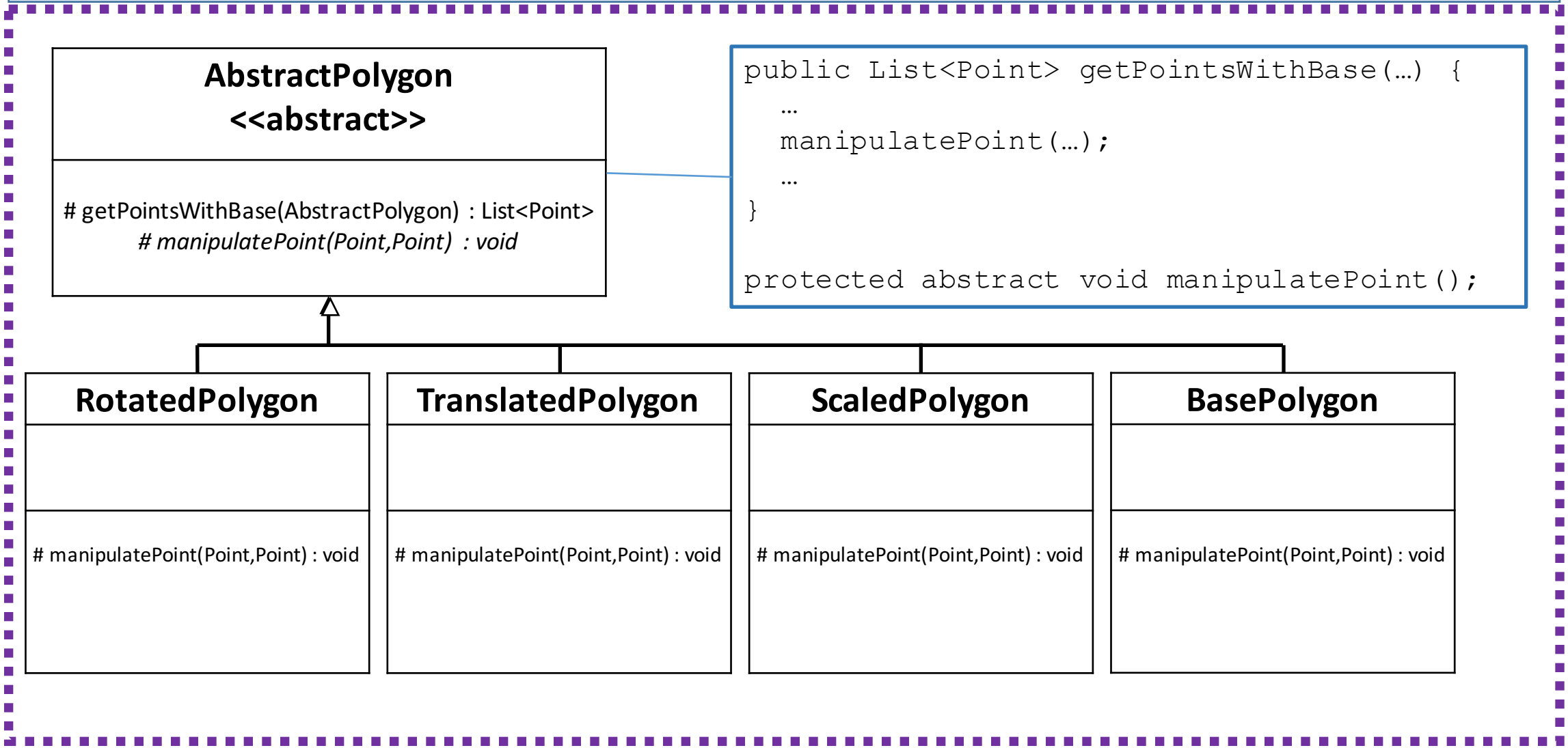
Johannes Åman Pohjola, 2017

Live code

- `tda551.polygon`


AbstractPolygon
<<abstract>>
+ paint(Graphics) : void
# getPointsWithBase(AbstractPolygon) : List<Point>
# manipulatePoint(Point,Point) : void
# getPoints() : List<Point>
IPolygon
<<interface>>
+ translate(int,int) : Polygon
+ rotate(double) : Polygon
+ scale(double,double) : Polygon
+ getCenterPoint() : Point
+ paint(Graphics) : void
PolygonFactory
+ createRectangle(int,int) : IPolygon
+ createSquare(int,int) : IPolygon
+ createTriangle(int,int) : IPolygon
RotatedPolygon
- base : AbstractPolygon
- radians : double
+ translate(int,int) : Polygon
+ rotate(double) : Polygon
+ scale(double,double) : Polygon
+ getCenterPoint() : Point
+ paint(Graphics) : void
# manipulatePoint(Point,Point) : void
TranslatedPolygon
- base : AbstractPolygon
- xMove : int
- yMove : int
+ translate(int,int) : Polygon
+ rotate(double) : Polygon
+ scale(double,double) : Polygon
+ getCenterPoint() : Point
+ paint(Graphics) : void
# manipulatePoint(Point,Point) : void
ScaledPolygon
- base : AbstractPolygon
- xFactor : double
- yFactor : double
+ translate(int,int) : Polygon
+ rotate(double) : Polygon
+ scale(double,double) : Polygon
+ getCenterPoint() : Point
+ paint(Graphics) : void
# manipulatePoint(Point,Point) : void
BasePolygon
- centerPoint : Point
- points : List<Point>
+ translate(int,int) : Polygon
+ rotate(double) : Polygon
+ scale(double,double) : Polygon
+ getCenterPoint() : Point
+ paint(Graphics) : void
# manipulatePoint(Point,Point) : void

Quiz: Vilket Design Pattern?

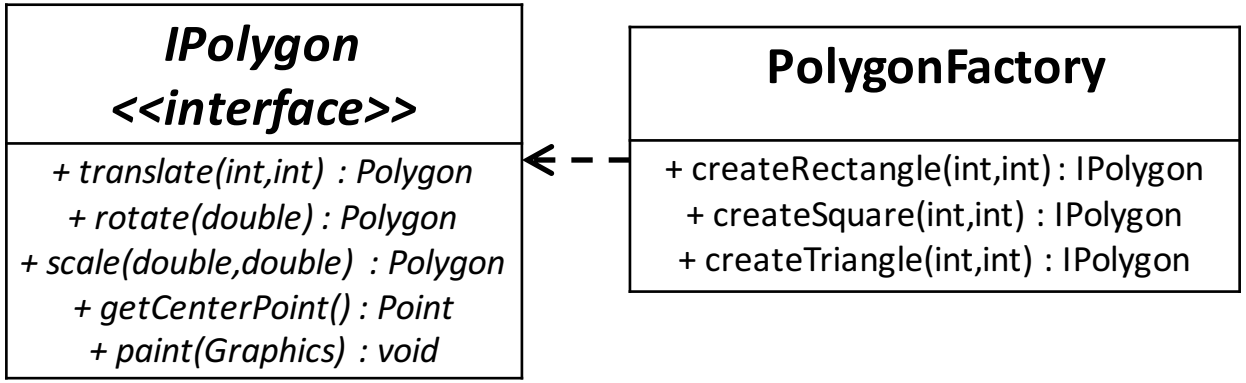


Template Method Pattern

When a mostly generic algorithm has context-dependent behavior:
Create an abstract superclass that implements the algorithm;
Include an abstract method representing the context-dependent behavior, and call this method from the algorithm;
Implement the context-dependent behavior in different sub-classes.

- När kod som är till stora delar gemensam, men beror på en liten del som inte är gemensam: bryt ut det som är gemensamt i en abstrakt klass, och låt det som inte är gemensamt representeras av en abstrakt metod som kan implementeras olika i olika sub-klasser.
- Syftet är att åstadkomma återanvändning av kod, trots att denna kod till vissa delar skiljer sig åt mellan de olika kontexten där den används.

Quiz: Vilket Design Pattern?



Hide concrete choice of data representation and constructors by creating concrete objects through static methods, whose return types specify an interface type.

Factory Method Pattern

- En *Factory Method* är en (oftast) statisk metod som abstraherar (döljer) beteendet hos en eller flera konkreta konstruktorer.
  - Minskar beroendet på paket-interna konkreta representationer.
  - Kan ha ett mer sofistikerat beteende än en konstruktor, t ex genom att välja mellan flera tillgängliga konstruktorer, ev från olika konkreta classes.
    - Alternativt namn: Smart Constructor (används oftare för funktionella språk).
  - Kan vara så enkel som att bara delegera till en specifik explicit konstruktor.
    - ”Framtidssäkring”: Om vi i framtiden behöver mer eller ändrad funktionalitet kan vi ändra i vår Factory Method, istället för att behöva ändra alla anrop till konstruktorn.

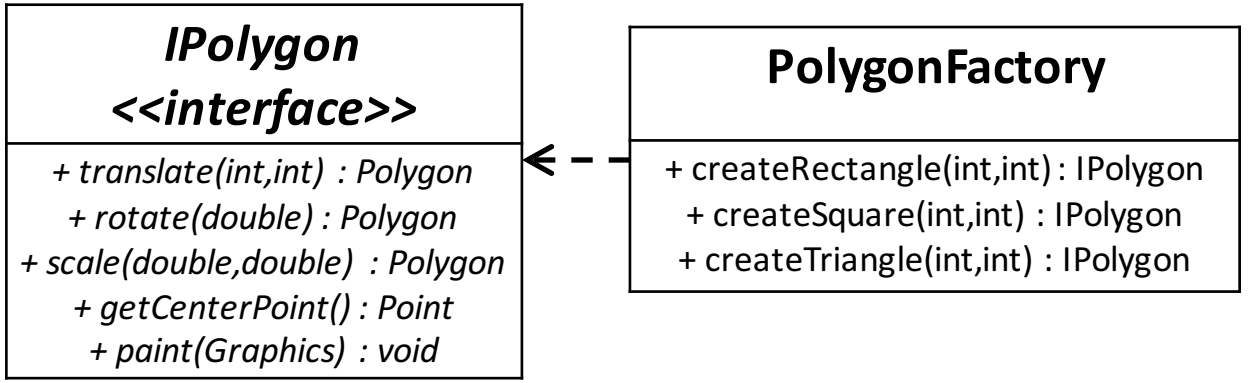
- En *Factory* är, per analogi, en class vars gränssnitt består av Factory Methods.
  - Obs: Abstract Factory Pattern är ett pattern som involverar användning av en väldigt specifik sorts Factory. De flesta Factories vi använder har inget med Abstract Factory Pattern att göra.

Abstract Factory Pattern

Create an abstract type for a Factory that specifies one or more abstract Factory Methods. Create different concrete Factories as subtypes to the abstract Factory type, each of which can return a objects of different concrete types for each specified Factory Method.

- Poängen med Abstract Factory Pattern är att abstrahera över många olika *Factories*, där varje konkret Factory kan returnera olika konkreta representationer från sina Factory Methods.
  - E.g. Vi skulle kunna ha två olika Factories för polygoner, med samma gränssnitt, där den ena returnerar stora polygoner och den andra returnerar små. DrawPolygons kan sen (låta användaren) välja av dessa den vill använda, utan att förändra resten av koden.

Quiz: Vilket Design Pattern?

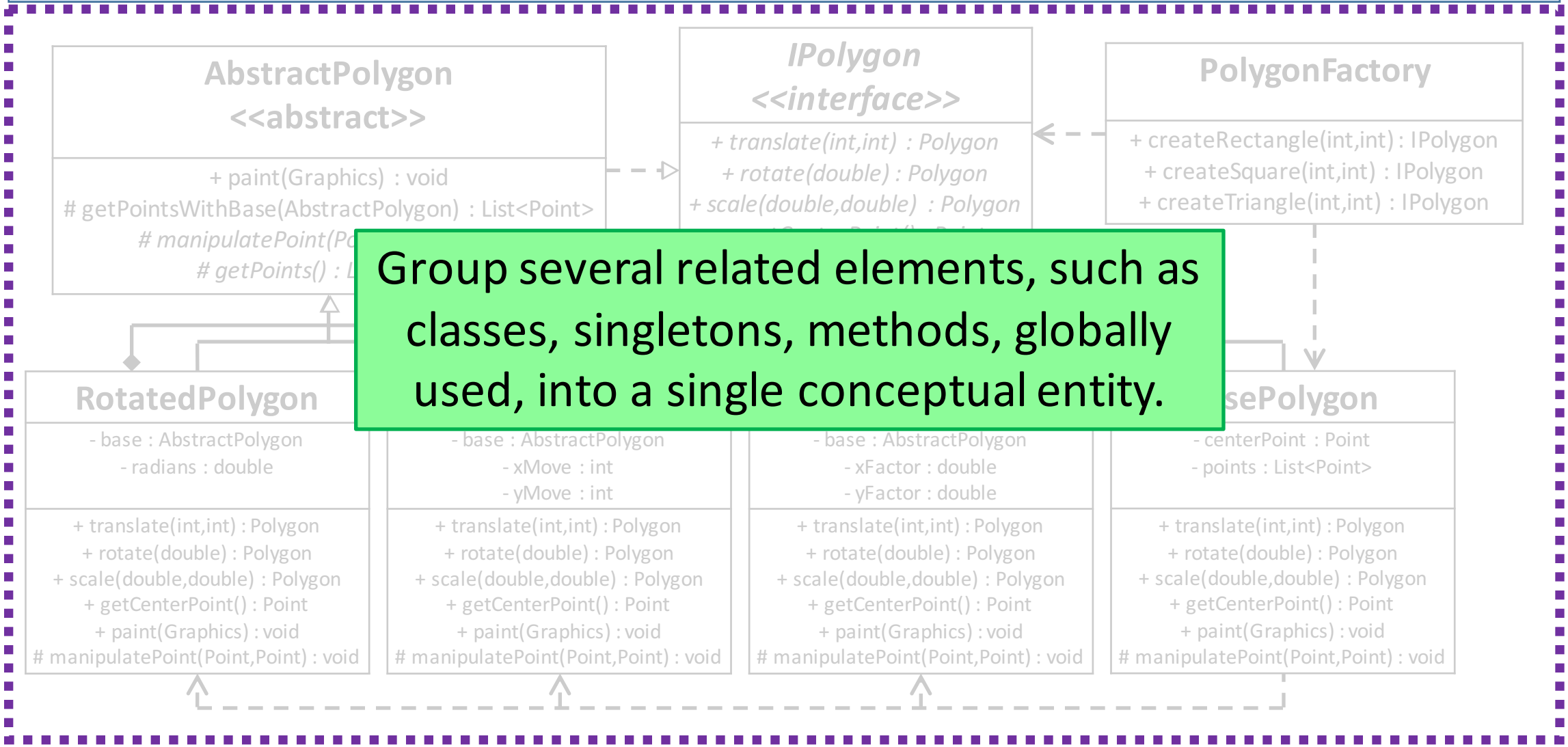


Hide internal complexity by introducing an object that provides a simpler interface for clients to use.

Facade Pattern

- Syftet med Facade Pattern är att öka abstraktionen för en sub-komponent (e.g. package), genom att gömma intern komplexitet bakom en ”fasad”, som ger ett förenklat (och abstrakt) gränssnitt.
  - Gör en komponent – ”bibliotek” – lättare att förstå och använda.
  - Reducera beroenden från extern kod på interna detaljer.
  - Ibland: dölj ett dåligt designat gränssnitt bakom ett bättre.
- Resten av sub-komponenten måste inte vara ”package private” för att ett Facade object ska vara användbart. Vi kan ha det för att ge ett ”skyltfönster” som tillhandahåller de vanligaste operationerna.
- Obs: En Facade måste inte vara en Factory, det råkar bara vara det i vårt exempel.
  - En Facade till ett package som tillhandahåller en datarepresentation, innehåller dock ofta Factory Methods.

Quiz: Vilket Design Pattern?

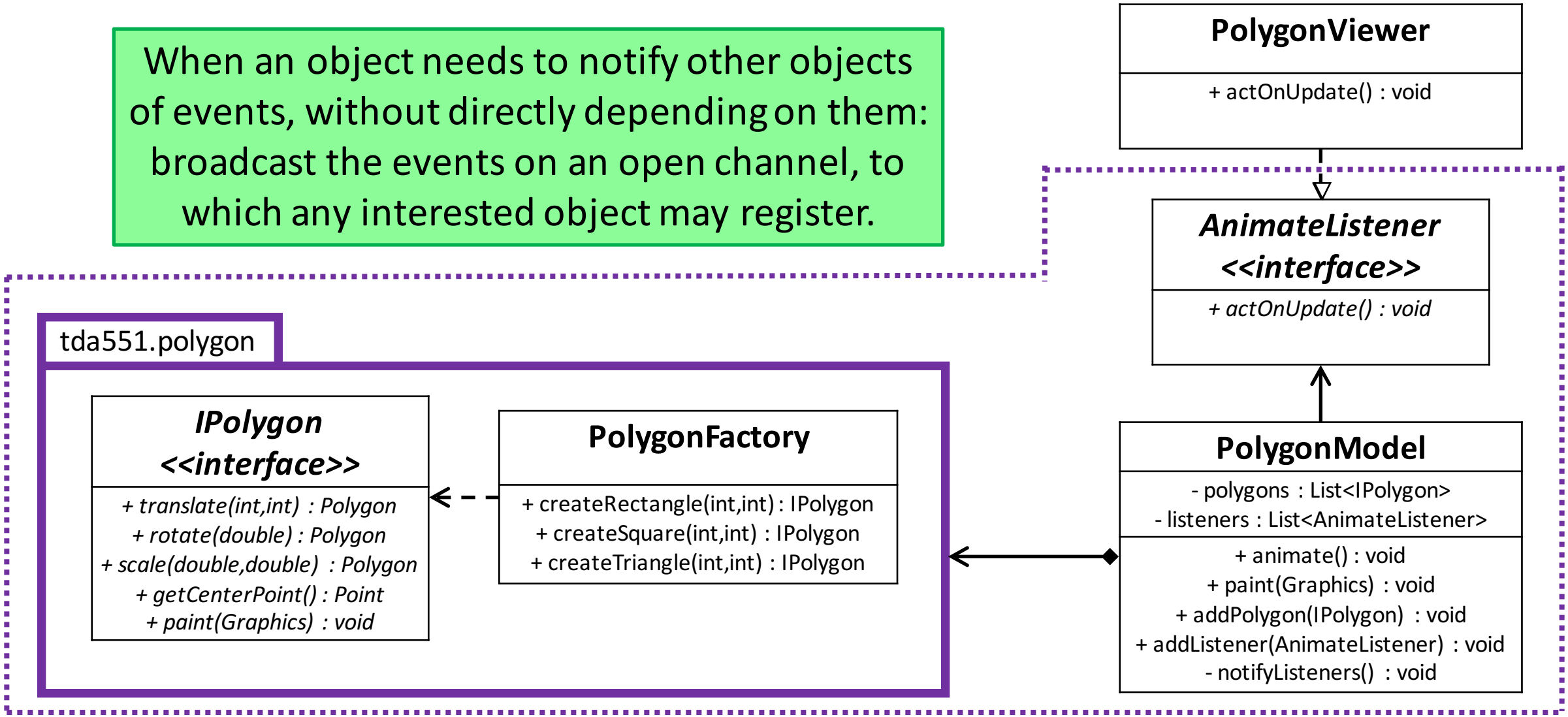


Module Pattern

- Module Pattern syftar till att skapa sammanhängande enheter på högre nivå än e.g. klasser.
  - Finns olika tolkningar: en tolkning är att det inte är Module Pattern ”på riktigt” om man inte har en *klass* som representerar ”modulen”, dvs en klass som inrymmer hela beteendet.
    - Anledningen är att det i språk som inte har språk-stöd för att skapa moduler (e.g. Javas packages) är så man skulle göra, och det är för dessa språk mönstret ursprungligen utvecklats.

Quiz: Vilket Design Pattern?

When an object needs to notify other objects of events, without directly depending on them: broadcast the events on an open channel, to which any interested object may register.



Observer Pattern

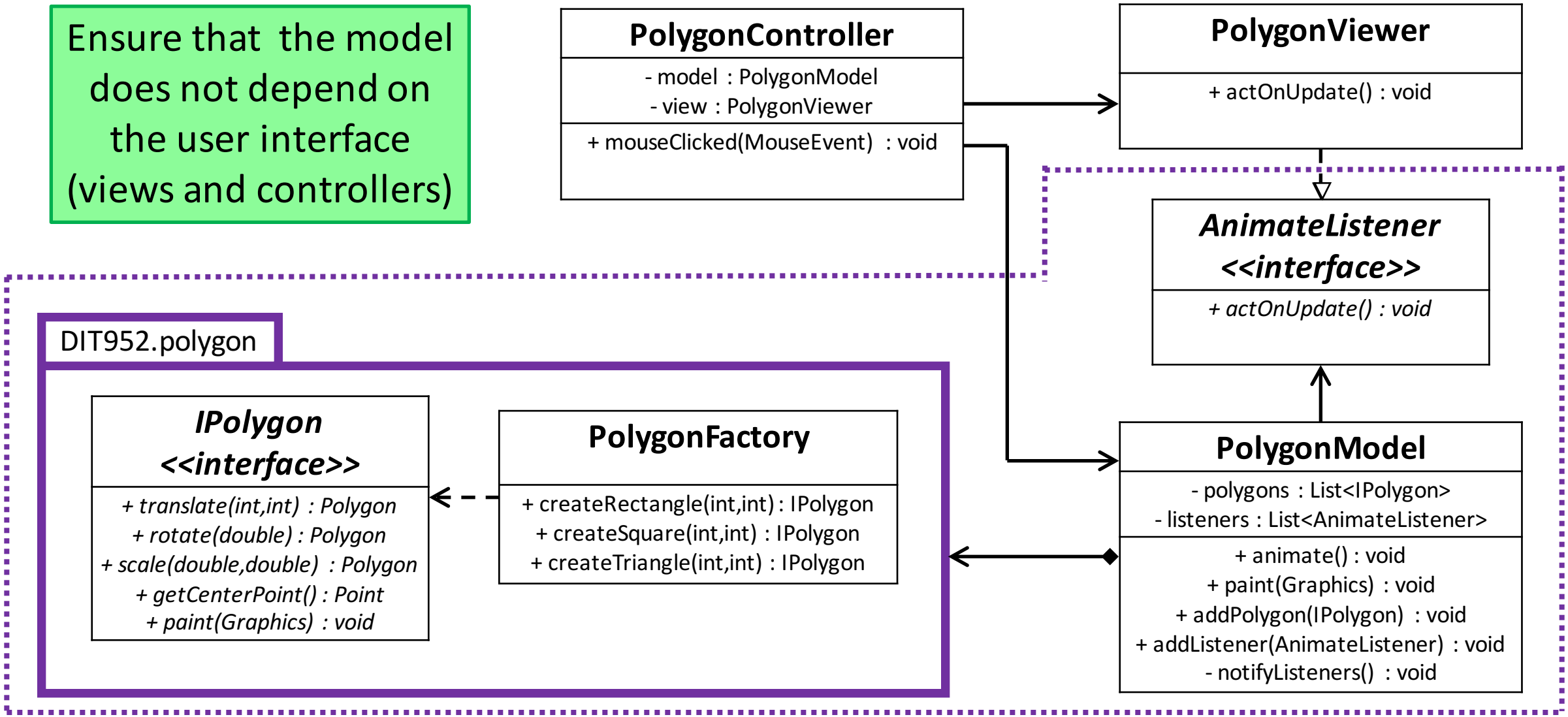
- DIP: Definiera ett interface enligt vilket objektet kommer att broadcasta sina events, och låt vem som vill implementera detta interface och registrera sig för att lyssna.

- Den som lyssnar kallas *observer*; den som skickar events kallas *observable.*

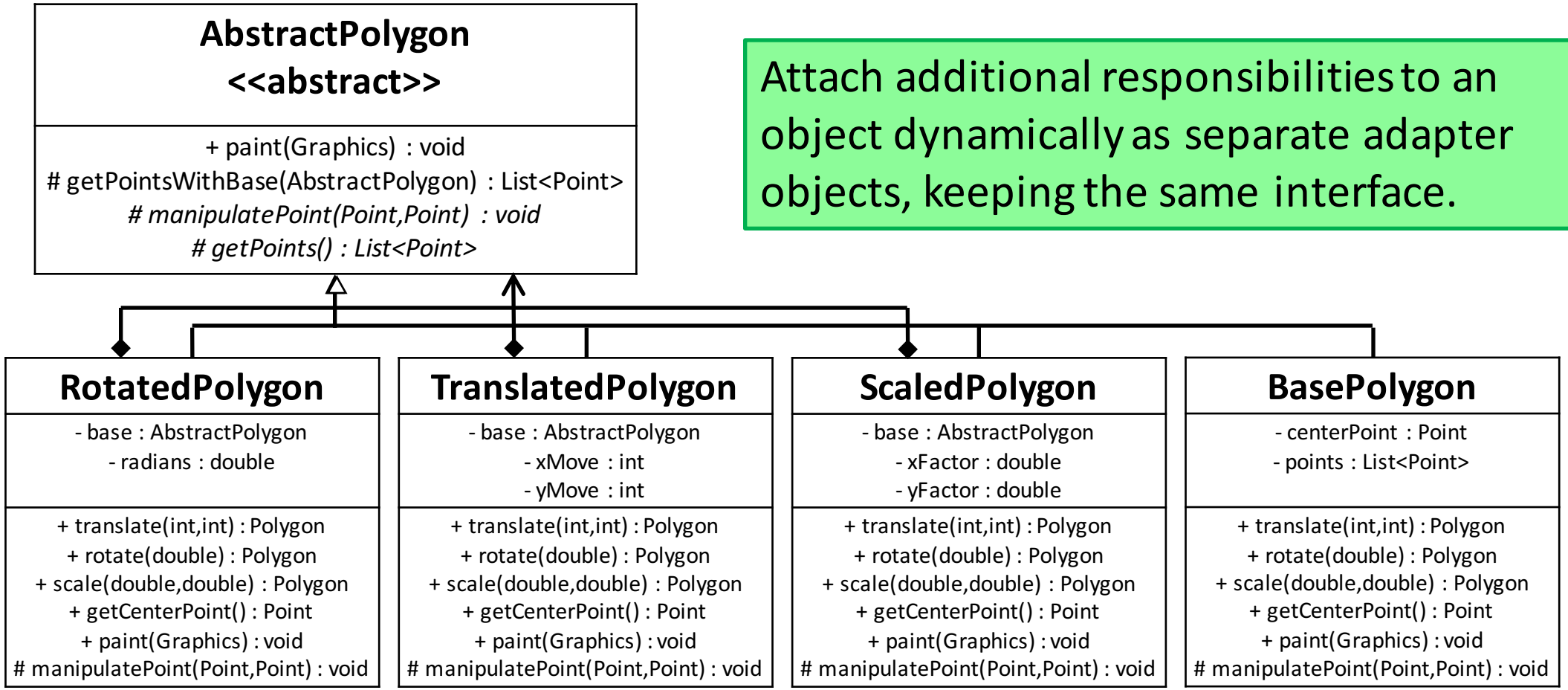
- ”Hollywood Principle”: Don’t call us, we’ll call you.
- ”Tell, don’t ask”

Quiz: Vilket Design Pattern?

Ensure that the model does not depend on the user interface (views and controllers)



Quiz: Vilket Design Pattern?



Attach additional responsibilities to an object dynamically as separate adapter objects, keeping the same interface.

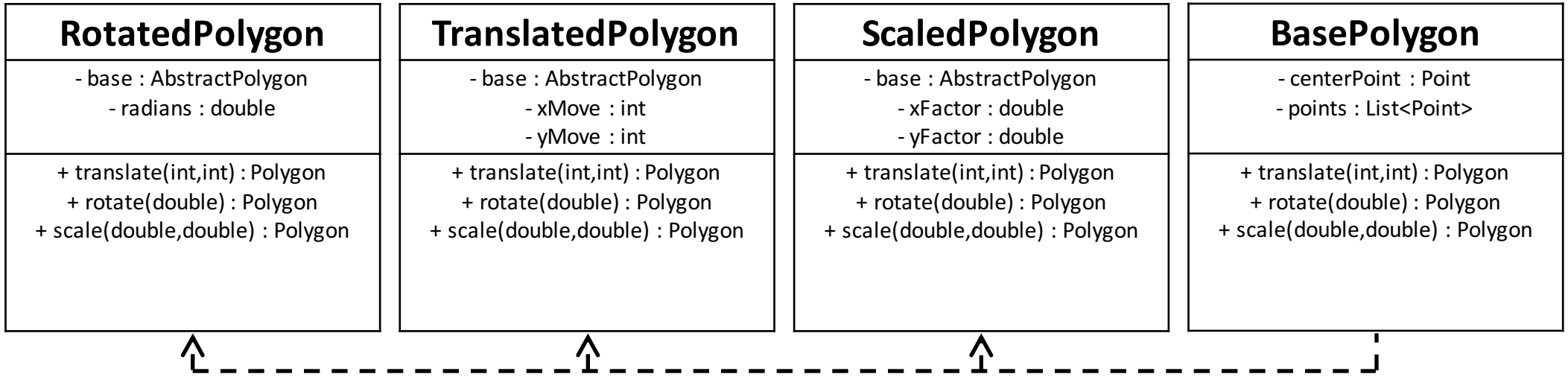
Decorator Pattern

- Genom att dela upp ett komplext objekt i flera beståndsdelar, som kan sättas ihop lager på lager med samma gränssnitt, så kan vi uppfylla Single Responsibility Prinicple.
  - ”An object should have at most one reason to change”
  - Varje ”reason to change” – förändringsdimension – och vilka tillstånd som kan antas inom en sådan, kan särskiljas från övriga. Detta minskar kognitiv komplexitet, och minskar risken för felaktig sammanblandning mellan dimensioner.

- Med Decorator Pattern representeras ett ”objekt” i domänen vi modellerar av en sammansättning av objects.
  - E.g. En polygon kan representeras av en sammansättning av en `BasePolygon`, en `RotatedPolygon` adapter och en `TranslatedPolygon` adapter.

Quiz: Vilket Design Pattern?

Chain the receiving objects of a request, and pass the request along until a suitable handler is found.



Chain of Responsibility Pattern

- Syftet med Chain of Responsibility Pattern är att undvika beroenden, från det objekt som skickar metod-anropet, till alla de objekt som skulle kunna hantera anropet.
  - Genom att skapa en enda ”entry point” där metod-anropet görs (helst via ett interface), så ser det från anroparen ut som att den pratar med ett enda objekt, trots att anropet internt kan skickas mellan flera olika objekt innan det hanteras.

Live code

- `translate`, `scale`, `rotate`

Method Chaining

- Method chaining är namnet på den språkfeature i Java (och andra språk) som låter oss sätta ihop flera metodanrop i en ”kedja”, utan att introducera lokala variabler för mellan-stegen.
  - E.g.

```
myPoly.translate(10,10).rotate(Math.pi).scale(2,2);
```

istället för

```
IPolygon p1 = myPoly.translate(10,10);
Ipolygon p2 = p1.rotate(Math.pi);
p2.scale(2,2);
```

Method Cascading

- *Method Cascading* är en specifik användning av Method Chaining, där objektet självt returneras från varje anrop i kedjan.
  - Smidig teknik: Låt alla mutators returnera `this` istället för att ha void som retur-typ. Detta möjliggör Method Cascading. E.g.:

```
public IPolygon rotate(double radians){
  this.radians += radians;
  return this;
}
```

istället för

```
public void rotate(double radians){
  this.radians += radians;
}
```

Quiz

- Varför är ett anrop som följande dåligt:

```
view.getFrame().repaint();
```

medan följande är en bra teknik?

```
myPoly.translate(10,10).rotate(Math.pi).scale(2,2);
```

- Svar: Det första involverar olika typer, det senare bara en typ. Det första introducerar därför extra beroenden.

Law of Demeter

- *Law of Demeter* (LoD), eller *Principle of Least Knowledge*, säger att vi ska undvika att introducera beroenden genom att hålla oss till följande riktlinjer:
  - En klass ska bara känna till sina närmaste ”vänner” (dvs ofrånkomliga beroenden).
  - En klass ska inte anropa metoder hos andra än sina vänner.

- ”Don’t talk to strangers”

Law of Demeter

- Vi uppfyller LoD genom att introducera metoder som agerar på interna objekt, istället för generiska getters:

```
public class View {
   private JFrame frame;
   public JFrame getFrame() {
    return frame;
   }
}

view.getFrame().repaint();
```

```
public class View {
   private JFrame frame;
   public void repaintView() {
      frame.repaint();
   }
}

view.repaintView();
```

Genom att inte exponera den interna användningen av JFrame, kan vi senare ändra valet utan att externa klienter påverkas.

ISP: Interface Segregation Principle

No client should be forced to depend on methods it does not use.

- Om ett objekt A är stort, med mycket funktionalitet, och objekt B beror på en liten del av denna funktionalitet: Introducera en abstraktion vars gränssnitt är precis den del av A som B behöver bero på.

Live code

- `IPolygon`

Local state

- Muterbara objekt har per definition olika (kanske oändligt många) tillstånd (states) de kan anta. Ett objekts interna tillstånd kallar vi för *local state*.

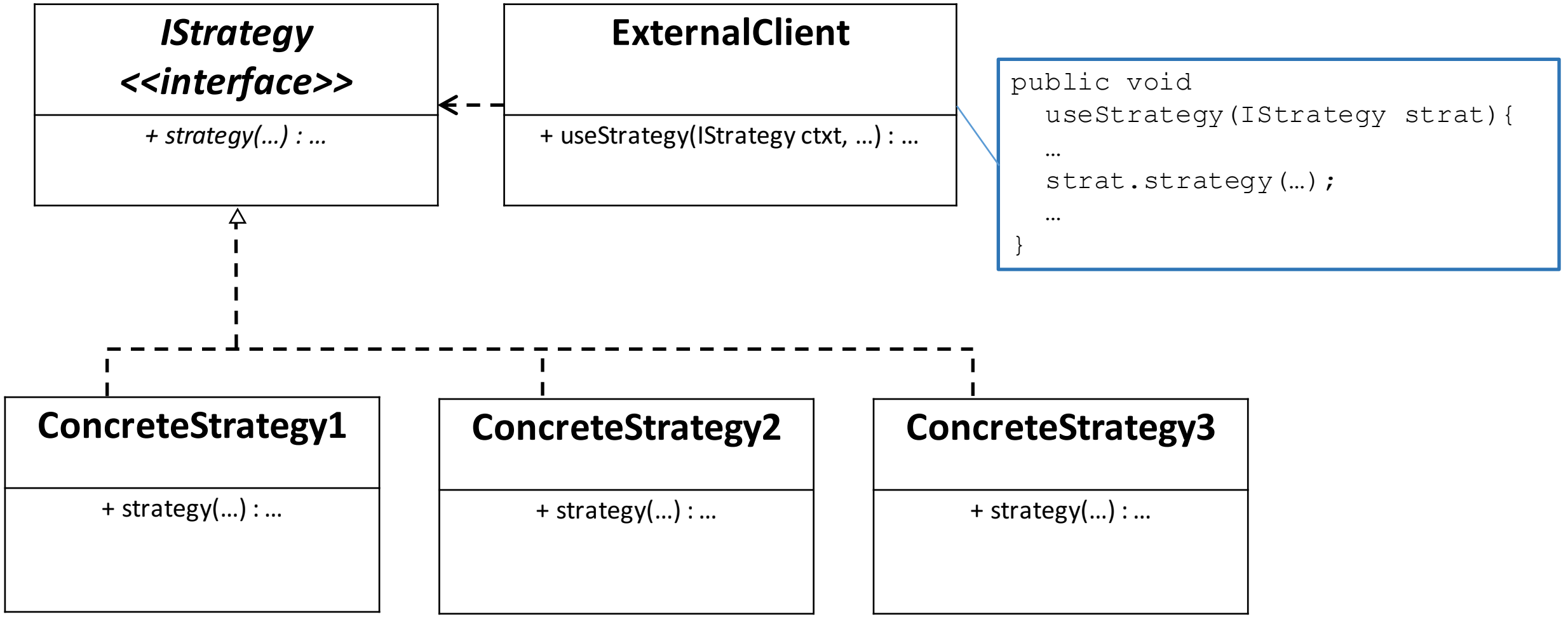
- Även med local state kan vi råka i trubbel:
  - Tillståndet (state) kan uppdateras oväntat.
  - Dåligt designade objekt kan hamna i *inkonsekventa* (inconsistent) tillstånd
    - E.g. flera attribut där värdet på ett av dem ska kunna härledas ur de andra, men håller fel värde.
    - `bedIsDown == true | bedAngle > 0`

Quiz: Vilket design pattern?

When a mostly generic algorithm can be varied in the details:
Create an interface with methods that represent the variability;
Use these methods within the generic algorithm;
Define different concrete behaviors as separate classes that implement the interface, and pass these to the generic algorithm as necessary.

- Svar: Strategy Pattern (se föreläsning 5-1).

Strategy Pattern

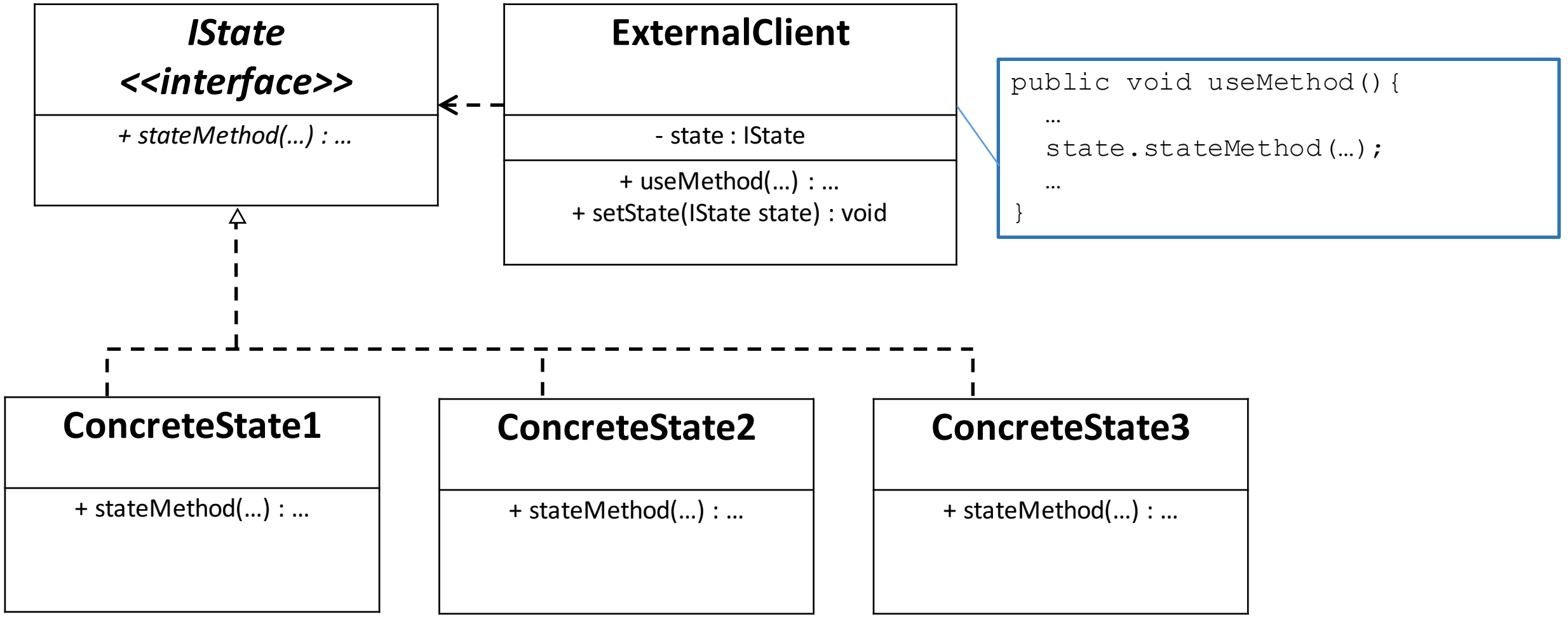


State Pattern

When an object can vary between a finite number of different states:
Create an interface with methods that represent the variability;
Use these methods within the object;
Define different concrete states as separate classes that implement the interface, and use these internally in the object as necessary.

- Definiera det som skiljer mellan olika states som objekt av olika klasser.
  - Kombineras gärna med Singleton Pattern, då det bara bör finnas ett objekt som representerar varje specifikt tillstånd.
- Växla mellan tillstånd med hjälp av specifika metoder, antingen i huvud-objektet eller hos de olika state-objekten (e.g. finite state machine).

State Pattern



State vs Strategy Pattern

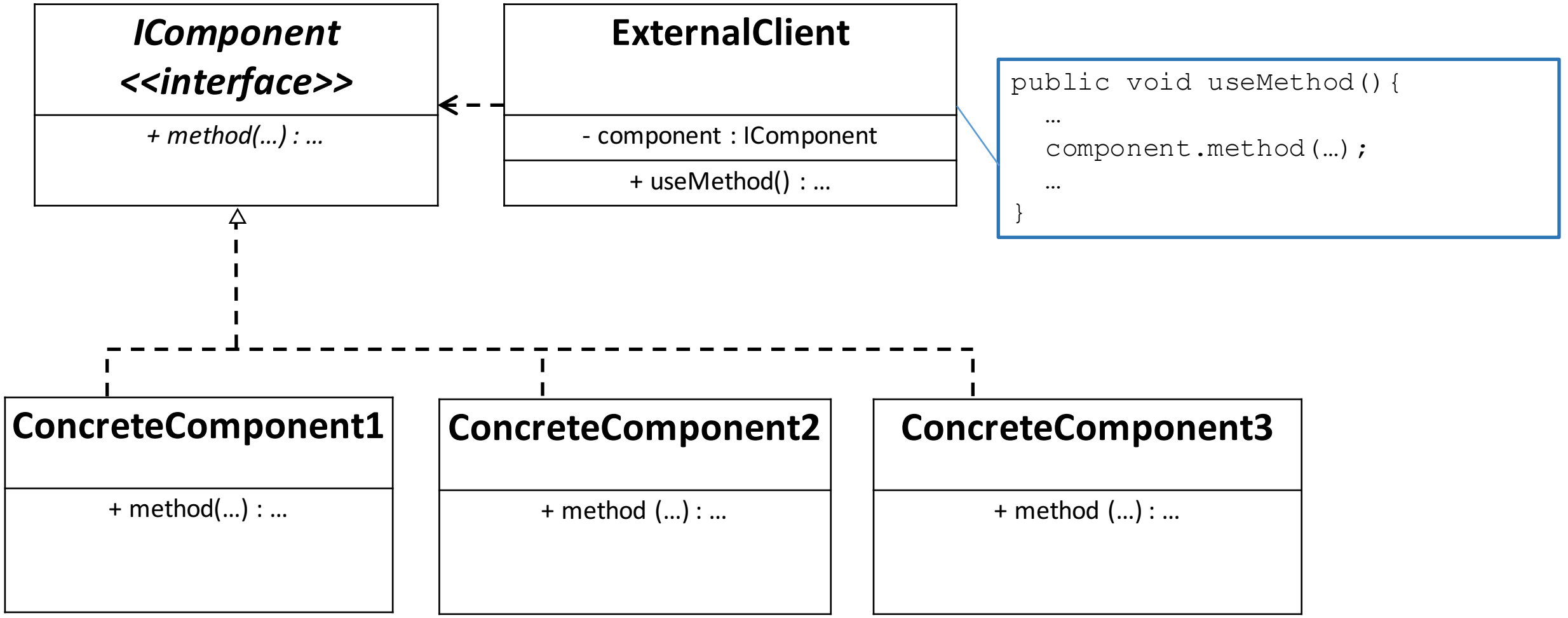
- State Pattern är implementationsmässigt detsamma som Strategy Pattern – men sättet vi använder det på skiljer sig lite.
  - I Strategy Pattern väljer vi *en* av många strategies, och kör sen vår algoritm med den.
  - I State Pattern vill vi kunna *byta* mellan olika states över tid.

Bridge Pattern

When an aspect of an object can vary independently of the object itself:
Create an interface with methods that represent the aspect;
Use these methods within the object;
Define different concrete implementations of the aspect and use these with the object as necessary.

- När ett objekt är sammansatt av olika komponenter, och dessa komponenter kan tänkas variera oberoende av objektet självt, då bör vi representera dessa med ett separat (SRP) interface (DIP) och låta konkreta implementationer av komponenterna implementera detta.

Bridge Pattern



State vs Strategy vs Bridge Pattern

- Bridge Pattern är i princip samma sak ytterligare en gång, men fyller ett annat syfte.
  - State och Strategy är båda *behavioral patterns* som säger hur vi bör tänka för att åstadkomma ett specifikt beteende.
  - Bridge pattern är ett *structural pattern* som förklarar hur vi bör tänka för att åstadkomma mesta möjliga variabilitet och polymorfism för objekt bestående av olika komponenter.

Exempel: State vs Strategy vs Bridge Pattern

- En bil kan ha olika sorters motorer. Det finns olika sorters bilar. Vi kan sätta ihop en motor med en bil till en specifik kombination, men konkreta bilar och motorer kan variera oberoende av varandra.
  - Bridge Pattern
- En amfibiebil kan växla mellan att köra på land eller köra på vatten. Den har samma uppsättning beteenden (e.g. köra, svänga, backa), men dessa implementeras olika för olika tillstånd (land/vatten).
  - State Pattern
- En bil kan köra längs en sträcka på olika sätt – t ex så fort som möjligt, eller så bränslesnålt som möjligt. Funktionaliteten ”köra” kan, vid olika tillfällen, ta en sådan strategi som argument, och anpassa beteendet därefter.
  - Strategy Pattern

Sammanfattning

- Principer
  - Single Responsibility Principle
  - Open-Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

  - Separation of Concern
  - Law of Demeter
  - High Cohesion, Low Coupling
  - Command-Query Separation
  - Composition over Inheritance

- Patterns och Tekniker
  - Template Method, Strategy, State
  - Bridge, Decorator, Adapter
  - Factory Method, Abstract Factory
  - Singleton
  - Chain of Responsibility
  - Iterator, Composite
  - Module, Facade
  - MVC, Observer, Mediator

  - Defensive Copying
  - Method Cascading

What’s next

Block 7-1:
Trådar
(lite lambdas)