Design Patterns

Objekt-orienterad programmering och design

Alex Gerdes, 2018

Vad är ett design pattern?

- Ett *design pattern* (designmönster) är en (ofta namngiven) generell lösning av en vanligt återkommande situation inom (mjukvaru-)design.
  - Termen och konceptet kommer ursprungligen ifrån arkitektur.
  - Populariserades inom objekt-orientering med boken ”Design Patterns”, ofta idag refererad till som ”Gang of Four” (efter de fyra författarna).
  - Vi pratar mest om design patterns inom objekt-orienterad design, men de existerar (mer eller mindre uttalade och erkända) inom alla paradigmer.

- Ett design pattern har ingen färdig kod – de är abstrakta mallar för hur ett problem kan lösas.
  - ”Formaliserade” best-practices.
  - I en given situation och kontext kan vi instansiera ett design pattern med specifika klasser, metoder, etc, och få en färdig lösning.

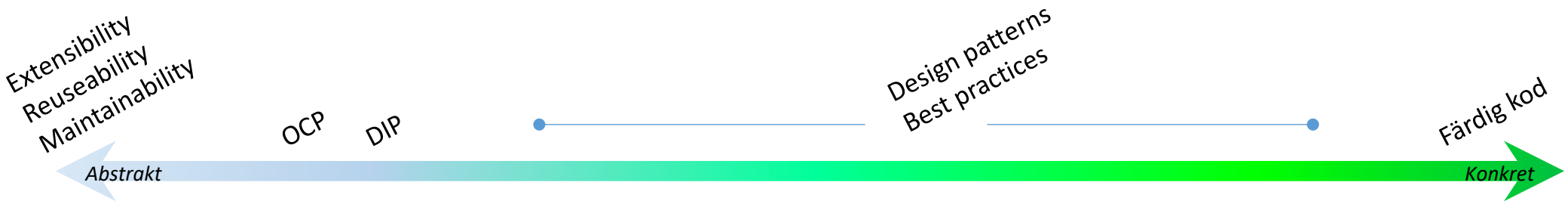
S.O.L.I.D.

- Inom objekt-orientering finns ett antal övergripande principer som vi arbetar efter, e.g.:
  - Open-Closed Principle: ”A component should be open for extension but closed for modification”

- De fem viktigaste (”the first five” – Robert C. Martin) har fått en minnesregel – SOLID – och man pratar ofta om SOLID objekt-orientering. Kan ni nämna fyra av de fem?
  - **S**ingle Responsibility Principle, **O**pen-Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, **D**ependency Inversion Principle.
  - (Vi har inte tagit upp Interface Segregation Principle ännu.)

Design patterns vs principer

- Vårt mål är att följa principerna – design patterns ger oss generella mönster för hur vi kan uppnå dem i specifika situationer.
  - Ingen skarp gräns, snarare en glidande skala; från principer i den mest abstrakta änden, till färdig kod i den mest konkreta, med design patterns någonstans däremellan.



Quiz: Vilket design pattern?

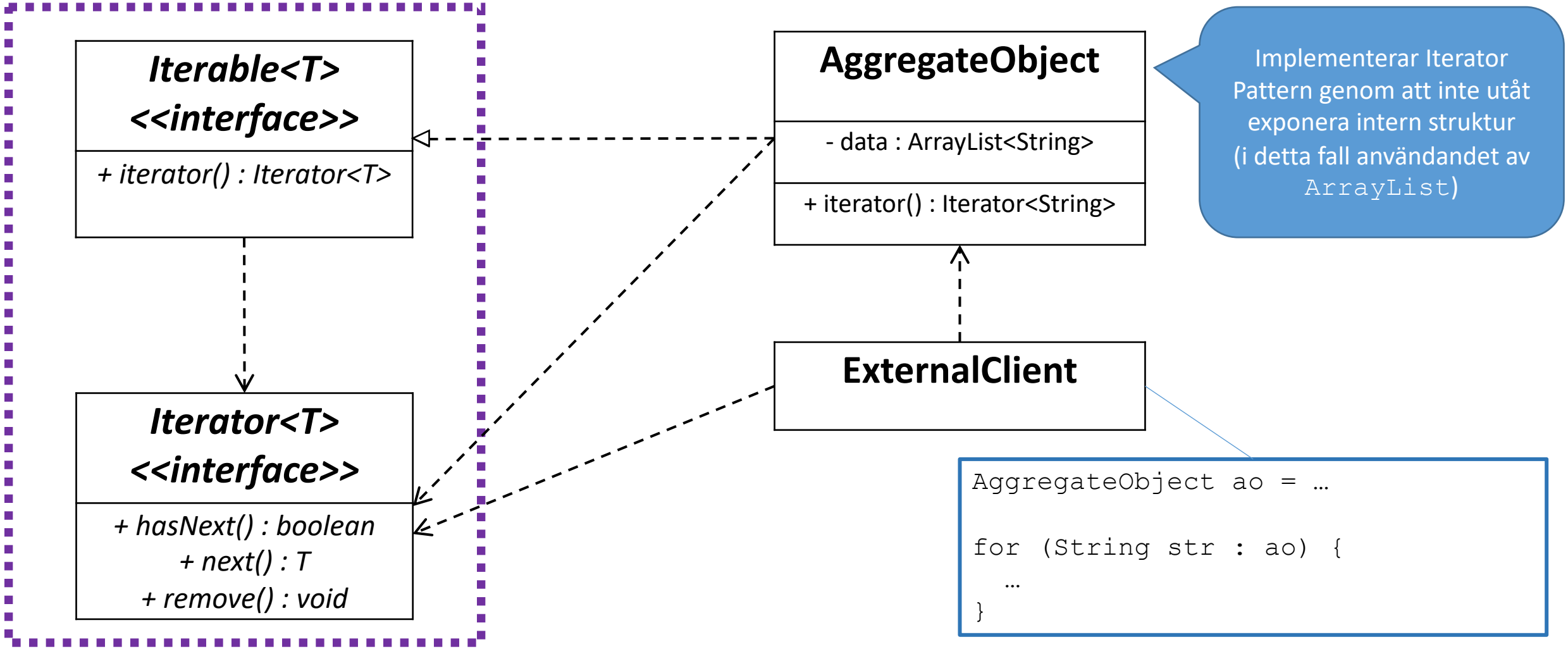
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- Ge klienter tillgång till elementen i ett sammansatt objekt som en sekvens, utan att visa hur objektet internt är uppbyggt.
- Svar: Iterator Pattern

Iterator Pattern

- Syftet med Iterator Pattern är abstraktion – att dölja intern representation från externa klienter.
  - I Java finns interfacen `Iterator<T>` och `Iterable<T>`, som är de rekommenderade verktyget att använda när man applicerar Iterator Pattern i Java.
  - Kärnan i detta pattern är dock den klass som implementerar `Iterable<T>` - det är den som verkligen använder Iterator Pattern, genom att dölja sin interna struktur.

Iterator Pattern



Quiz

Varför implementerar ett aggregate objekt inte direkt `Iterator` interfacet men `Iterable` istället (som returnerar en `Iterator`)?

Svar: ett objekt kan itereras flera gånger samtidigt och man måste ha tillståndet (på vilken plats man är i samlingen) separat.

Live code

- `Map`

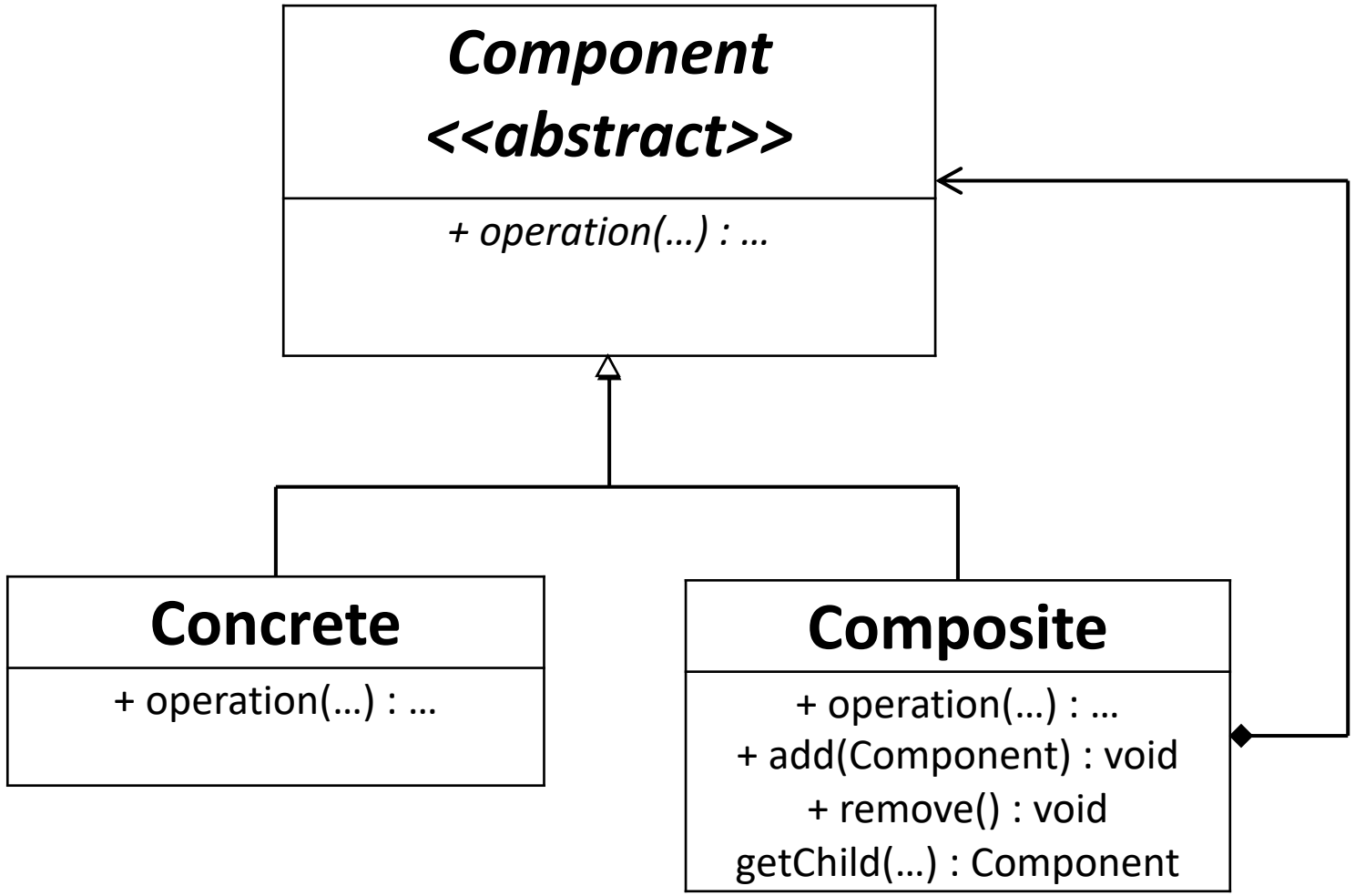
Composite Pattern

Treat a group of objects uniformly as if they were a single instance of an object.

- Gör det möjligt att använda grupper av objekt på samma sätt som ett enskilt objekt av den typen.

- I övningen försöker vi halvvägs behandla en lista av polygoner på samma sätt som en ensam polygon, e.g. vi definierar `paint` (men inte `updateCenter` – än) för en hel lista.

Composite 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.

Live code

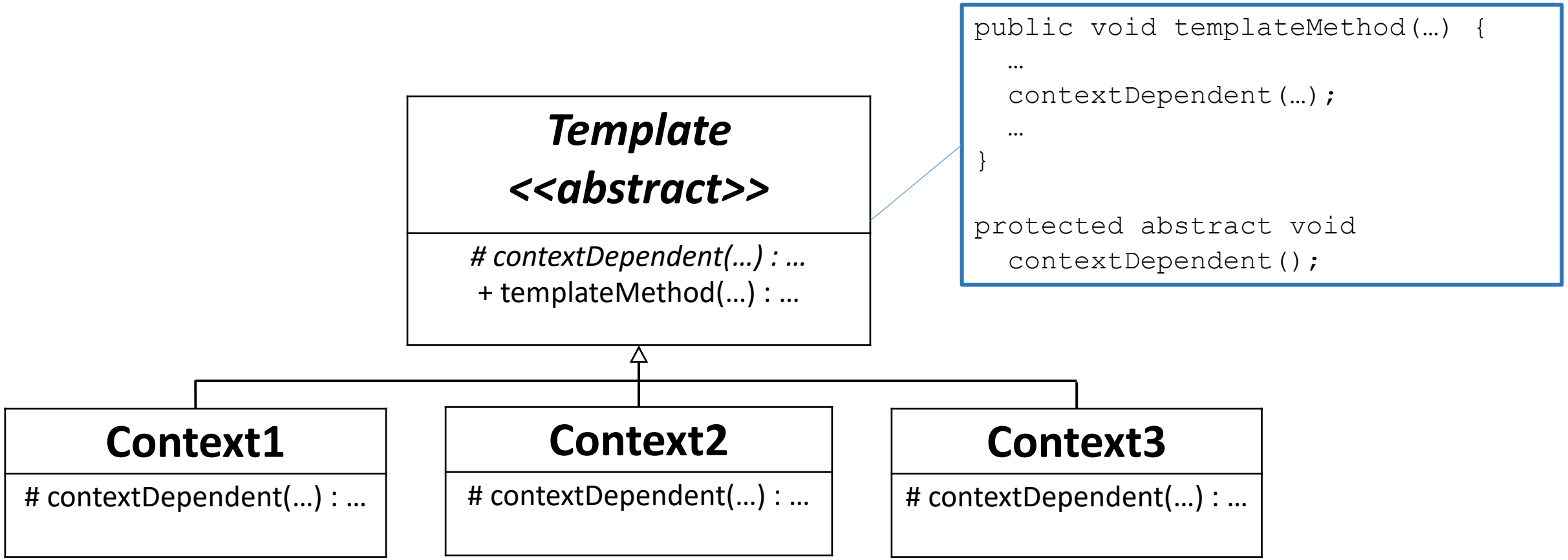
- `tda551.shapes.Polygon`

Template Method Pattern

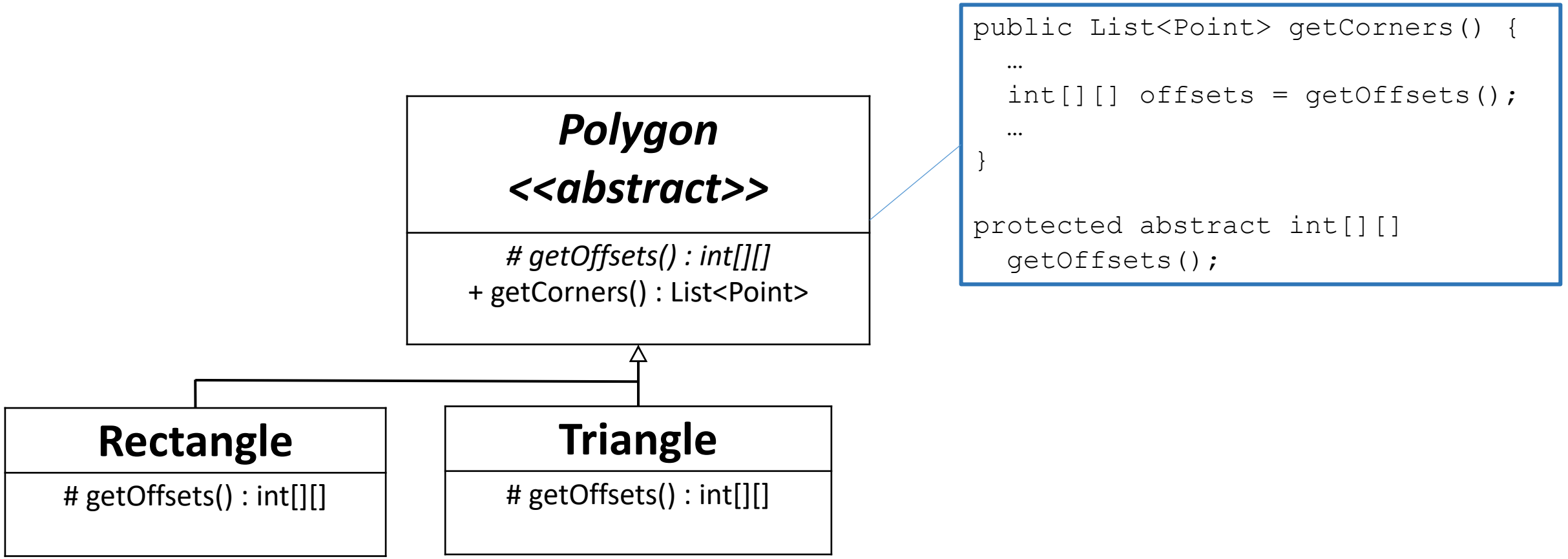
- Syftet med Template Method Pattern är att åstadkomma återanvändning av kod, trots att denna kod till vissa delar skiljer sig åt mellan de olika kontexten där den används.

- I grundutförandet pratas om ”abstract superclass” och kontext som ”subclasses” (som i exemplet) – men det går lika bra att använda Template Method Pattern med delegering och interfaces också.

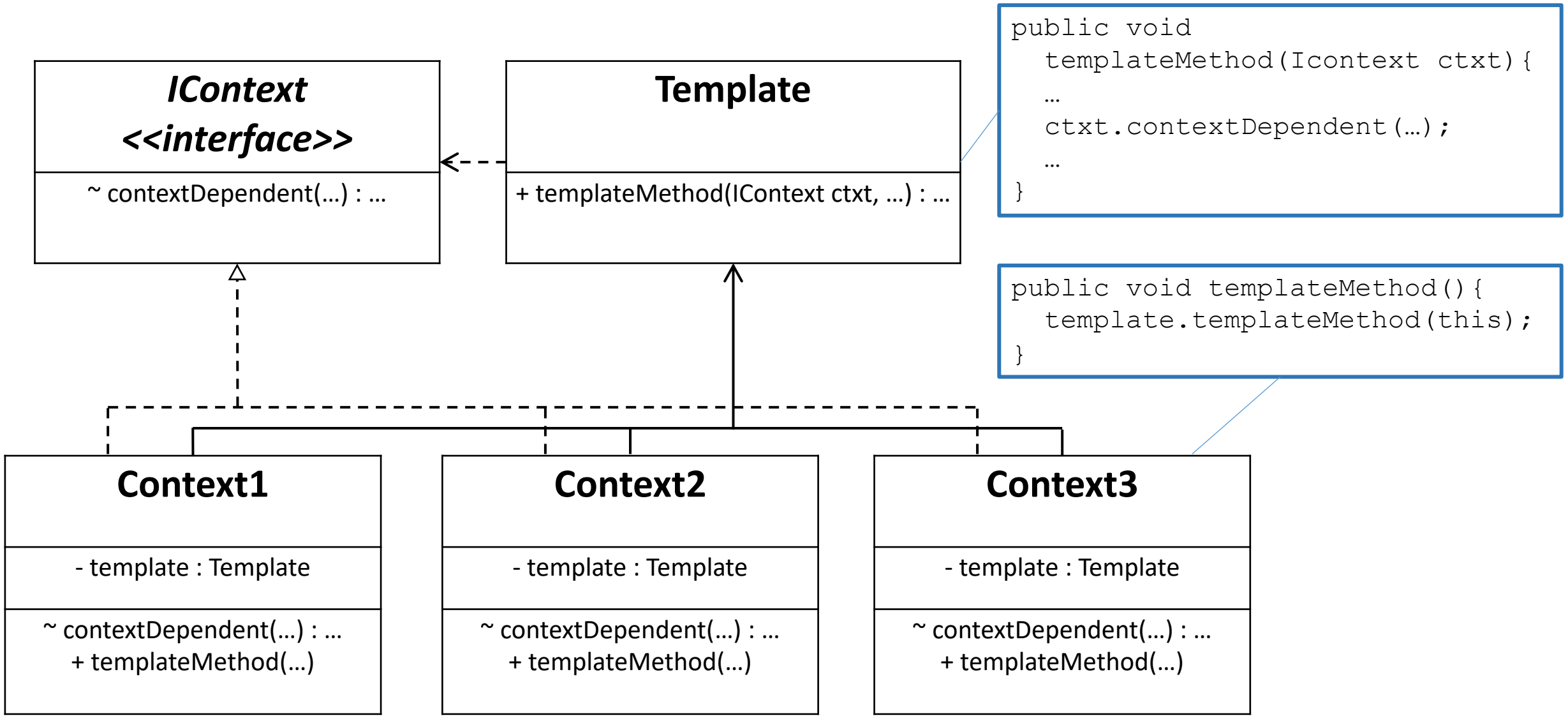
Template Method Pattern



Template Method Pattern i `tda551.shapes.Polygon`

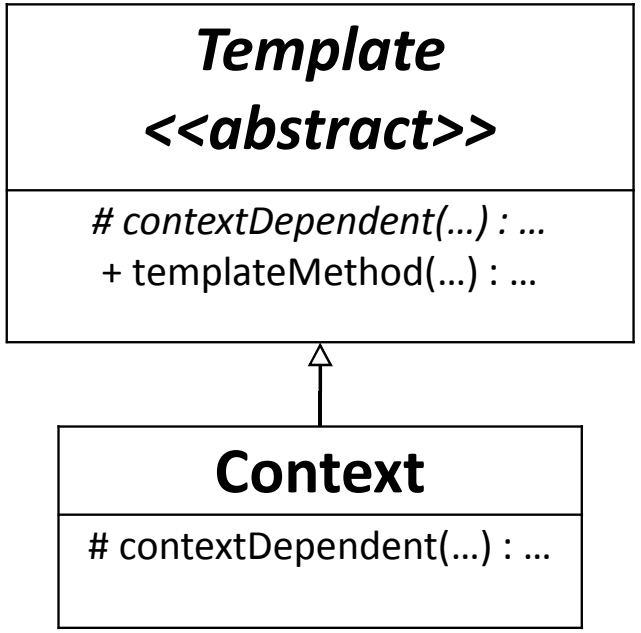


Template Method Pattern (delegering)



Viktigt om Template Method Pattern

- OBS: För att det ska röra sig om Template Method Pattern måste vi ha *både* den abstrakta metoden som implementeras i subklasserna (e.g. `getOffsets`), *och* den konkreta `templateMethod` som anropar den abstrakta (e.g. `getCorners`).
- Att bara ha en abstrakt metod (e.g. `paint` i Shape) är *inte* en instans av Template Method Pattern – det är helt vanlig användning av en abstrakt klass för polymorfism.



Live code

- `Plotter`

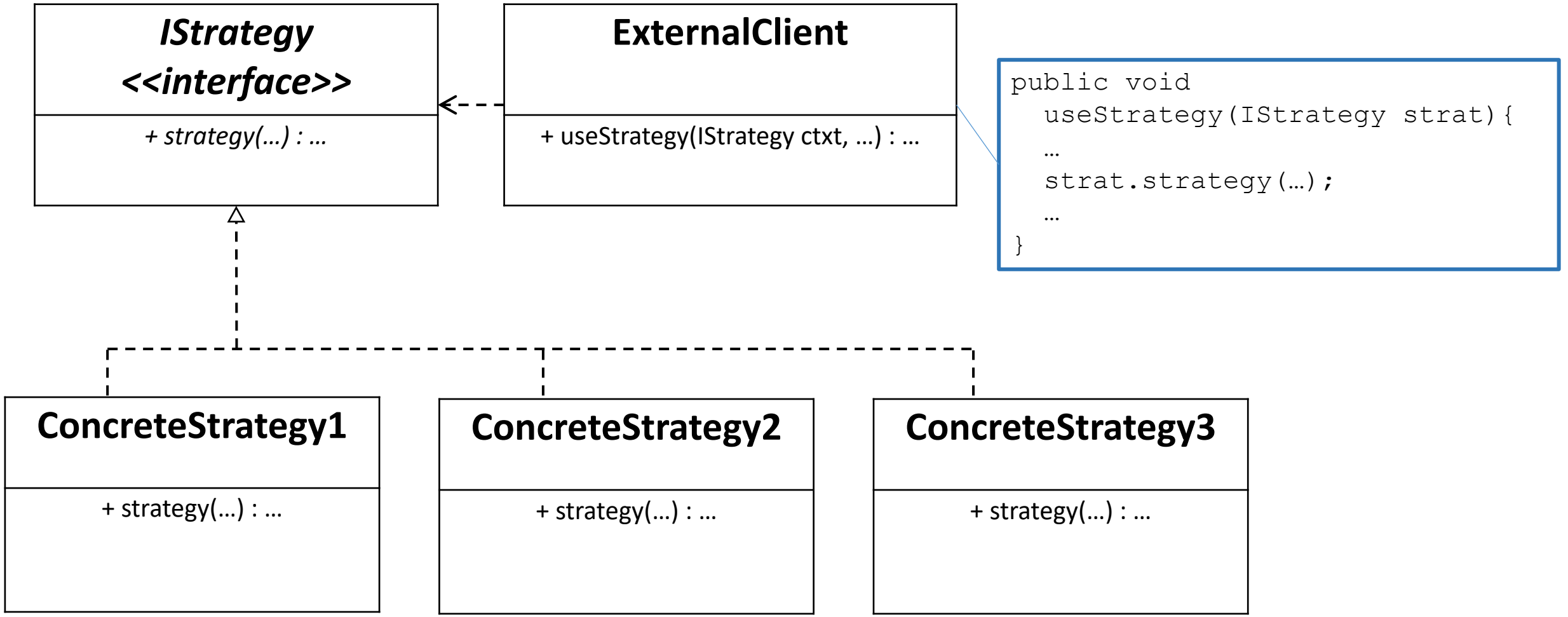
Strategy Pattern

- Strategy Pattern går steget längre än Template Method Pattern.
  - Vi knyter inte det som varierar till en specifik subclass (kontext), utan definierar ett fristående interface. Vi kan sen definiera helt olika beteenden som fristående klasser som implementerar detta interface.

- Konkret exempel i Java:
  - `Comparator<T>` med metoden `compare(T o1, T o2)`. Vi kan t ex implementera olika `Comparator<String>` med olika beteende – ascending/descending order, case (in-)sensitive, längd, … - för att jämföra strängar.
  - I Collections finns metoden `sort` som utnyttjar detta:

```
public static <T> void sort(List<T> list, Comparator<? super T> c) {…}
```

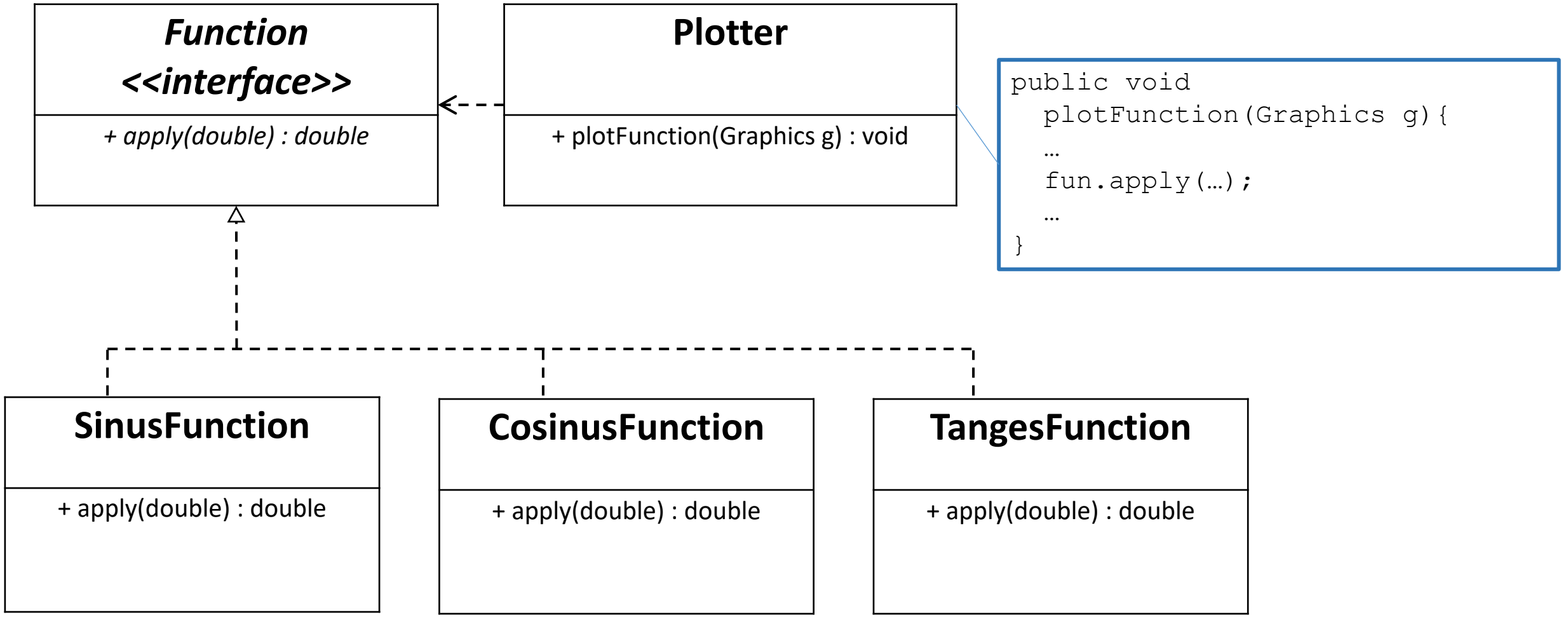
Strategy Pattern



Live code

- `Plotter`

Strategy Pattern i `Plotter`



Local state

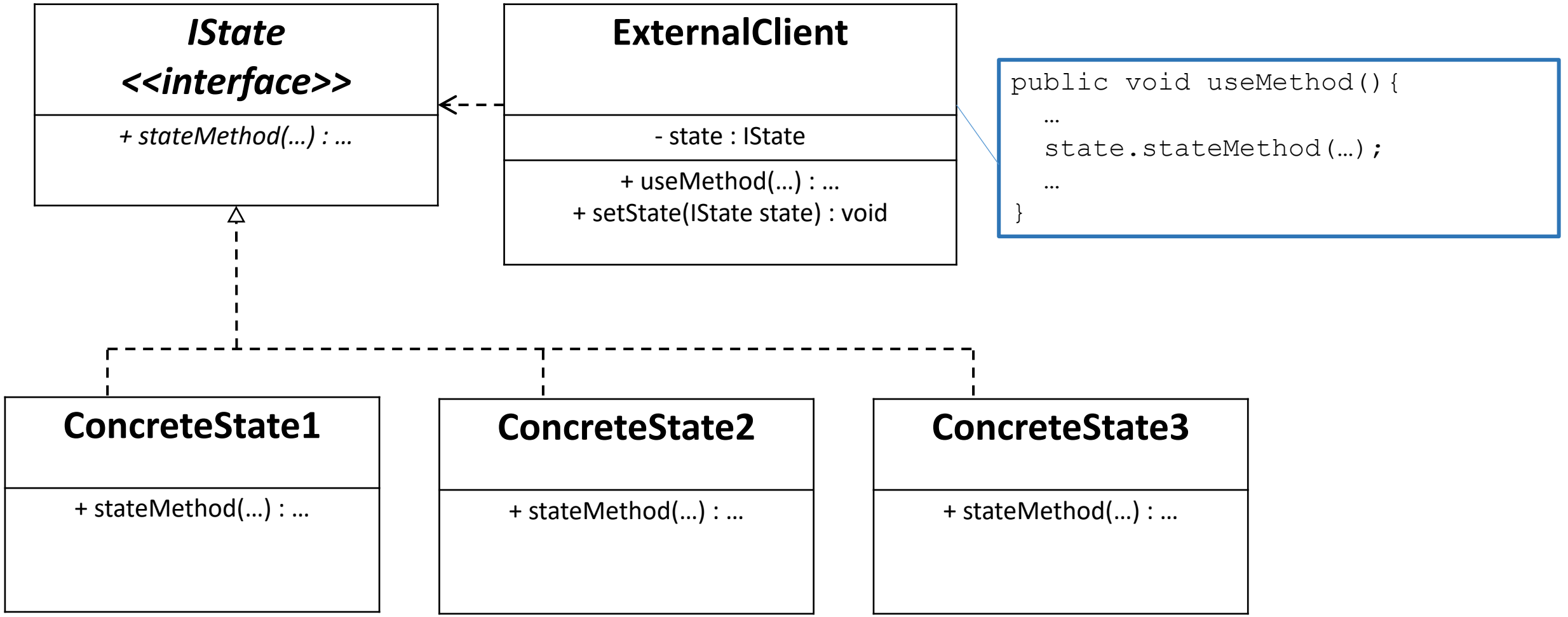
- Muterbara objekt har per definition olika (kanske oändligt många) tillstånd (states) de kan anta. Ett objekts interna tillstånd kallar vi för *local state*.
- Med naiv local state kan vi råka i trubbel:
  - Tillståndet (state) kan uppdateras oväntat.
  - Dåligt designade objekt kan hamna i *inkonsekventa* (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`

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

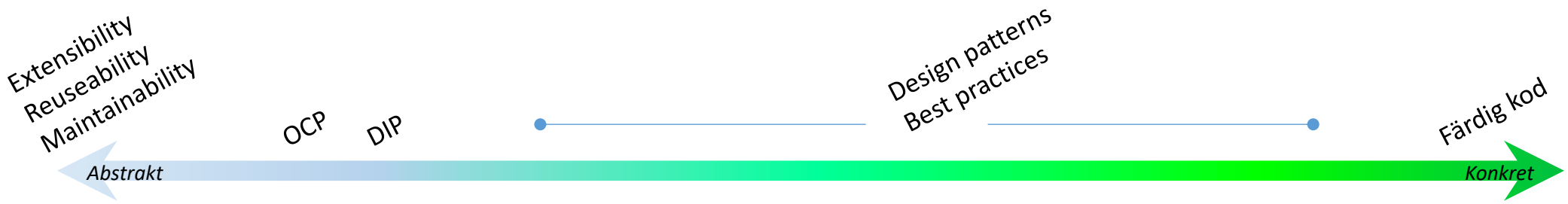


State vs Strategy Pattern

- 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.

Olika sorters design patterns

- Begreppet Design Patterns spänner över en stor del av spektrat av lösningsmetoder, från väldigt konkret till hyfsat abstrakt.



- Vi kan skilja på olika sorters design patterns, baserat på vilken sorts problem de syftar till att lösa:
  - Architectural, Structural, Behavioral, Creational, …

Architectural design patterns

- *Architectural patterns* behandlar hur kod bör struktureras på en övergripande nivå, utan att gå in på specifika classes etc.
  - Behandlas ibland separat från *software design patterns* – de är ofta snarare *system design patterns*. Men gränsen är flytande.

- Exempel: Model-View-Controller; Module (i viss användning)
  - Mer om MVC nästa föreläsningen.

Structural design patterns

- *Structural patterns* behandlar hur saker bör struktureras och hänga ihop på e.g. klass-nivå.
  - För structural patterns ger vi typiskt ett UML-diagram med boxar med abstrakta namn, som vi ger konkreta namn när vi använder dem. Vi bryr oss om vilka classes och interfaces vi har, och vilka beroenden vi har mellan dem, men inte alltid vilka metoder de har.

- Exempel: Composite
  - Specificerar ett specifikt sätt att ordna klasser inbördes för att åstadkomma en effekt – i den här fallet att kunna använda grupper av objekt på samma sätt som enskilda objekt.

Behavioral design patterns

- *Behavioral pattern* behandlar hur vi bör tänka kring hur klasser och objekt kommunicerar med varandra.
  - E.g. vilka argument- och retur-typer bör vi ge metoder.
  - Ges ofta med mer detaljerade UML-diagram, där vi även bryr oss om vilka metoder vi har, och hur de interagerar.

- Exempel: Iterator Pattern, Template Method, Strategy
  - Iterator Pattern säger att om externa klienter ska ges tillgång till intern aggregerad data (e.g. en intern lista), ge dem en abstrakt representation (en `Iterator`) och inte den konkreta listan (e.g. `ArrayList`).

Creational design patterns

- *Creational patterns* behandlar hur vi bör tänka när vi skapar nya objekt.
  - Kan ha många olika syften, alltifrån hur vi kan göra skapande av många objekt så effektivt som möjligt, till hur vi kan dölja intern representation så mycket som möjligt.

- Exempel: Factory Method Pattern
  - Factory Method Pattern säger att vi kan introducera metoder som skapar objekten internt och sen returnerar dem.

Oviktig uppdelning

- Vilka patterns som tillhör vilken kategori är tämligen oviktigt, och inte alltid självklart.

- Det viktiga är insikten att design patterns – dvs mallar och best practices – kan finnas och vara användbart på många olika nivåer.

- Att förstå på vilken nivå ett design pattern agerar kan vara en hjälp att förstå det.

Quiz: Design pattern?

Define an interface for creating an object, but let subclasses decide which concrete class to instantiate. Let a class defer instantiation it uses to subclasses.

- Definiera en metod som säger sig skapa objekt av en abstrakt typ (abstract class, interface). Låt subtyper avgöra vilken faktisk konkret typ som instansieras, genom overriding/implementering av metoden.

- Svar: Factory Method Pattern (ursprunglig formulering)

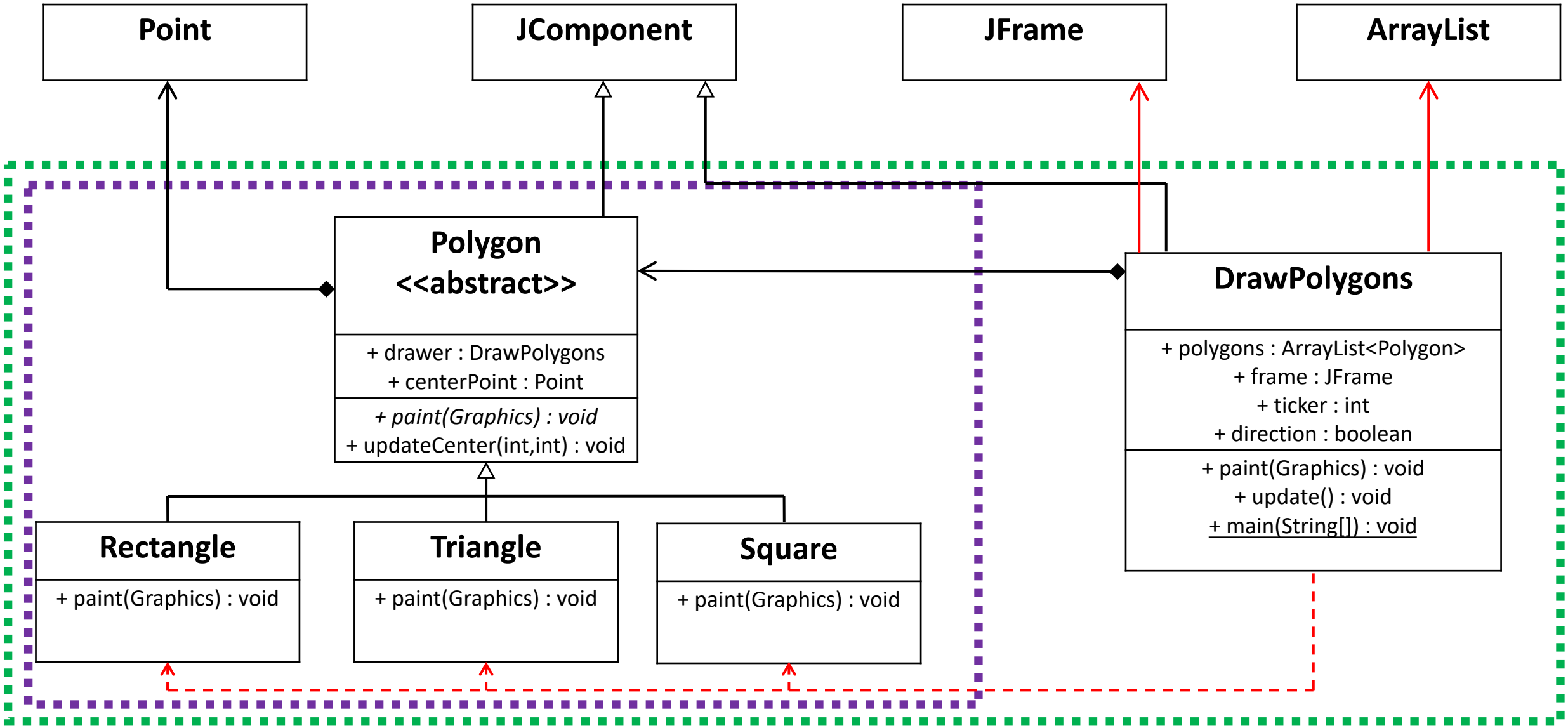
Factory Method Pattern

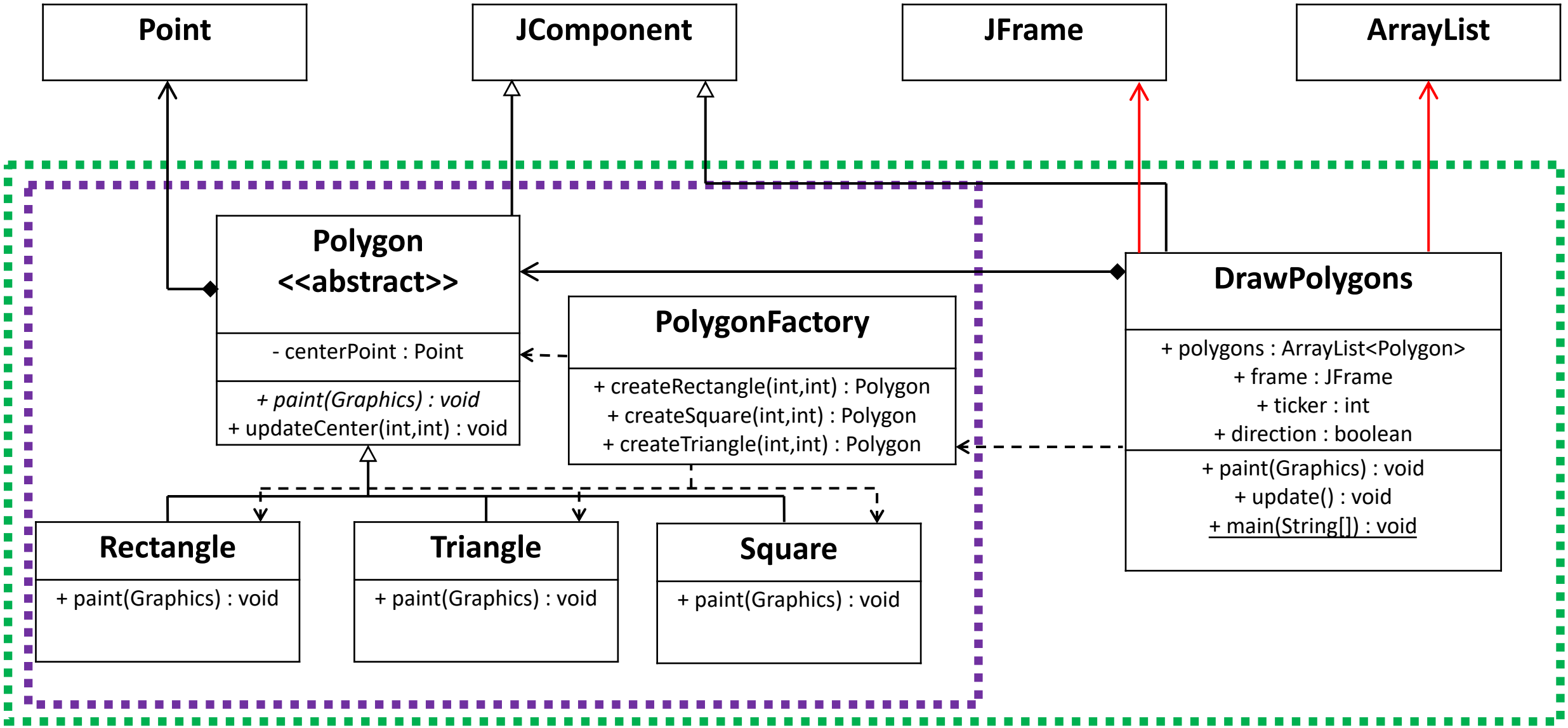
- Syftet med Factory Method Pattern är att dölja intern implementation:
  - Vilka konkreta konstruktorer som används.
  - Vilka specifika klasser som används – undviker beroende på konkreta implementationer (följer DIP).

- En Factory Method 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.

Live code

- `PolygonFactory`


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>>
- centerPoint : Point
+ paint(Graphics) : void
+ updateCenter(int,int) : void
PolygonFactory
+ createRectangle(int,int) : Polygon
+ createSquare(int,int) : Polygon
+ createTriangle(int,int) : Polygon
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

Quiz: Vilket design pattern?

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

- Dölj intern komplexitet och representation genom ett väldefinierat gränssnitt. Detta gränssnitt tillhandahålls av ett objekt, och ibland även ett antal associerade interfaces.

- Svar: Facade Pattern

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.

Live code

- `PolygonFactory` + `IPolygon`

Quiz: Vilket design pattern?

Allow otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

- Få inkompatibla komponenter att fungera ihop genom att konvertera gränssnittet hos den ena till det som den andra (klienten) förväntar sig.
- Hint: Hur kan du stoppa in e.g. en USB-kontakt i ett strömuttag?
- Svar: Adapter Pattern

Adapter Pattern

- Syftet med Adapter Pattern är att göra annars inkompatibla komponenter kompatibla med varandra.
  - Kallas ibland slarvigt för Wrapper – men den termen är tvetydig och kan syfta på många olika design patterns.

- Används typiskt när kod tidigare fungerat med komponent X, och nu (också) behöver fungera med komponent Y som har ett annat gränssnitt.
- I `tda551.polygons` vill vi byta ut subkomponenten `polygons` mot det nya `tda551.shapes`.

Live code

- Plugging in `tda551.shapes` into `tda551.polygons`

Summering

- Design patterns låter oss återanvända struktur och metodik som andra smarta designers redan kommit på, och som använts och prövats under lång tid.

- Ofta är design patterns konkret formulerade tekniker för att följa våra SOLID-principer i specifika fall.
  - Principerna är de mål vi vill uppnå, design patterns är verktyg.
    - E.g. Iterator Pattern gör att vi följer Dependency Inversion Principle.
  - Det är viktigt att förstå vad det är ett design pattern vill åstadkomma, för att veta hur och när det bör appliceras.

What’s next

Block 5-2:
Model – View – Controller