Factory Method
och andra Design Patterns

Objekt-orienterad programmering och design

Sólrún Halla Einarsdóttir och Alex Gerdes, 2018

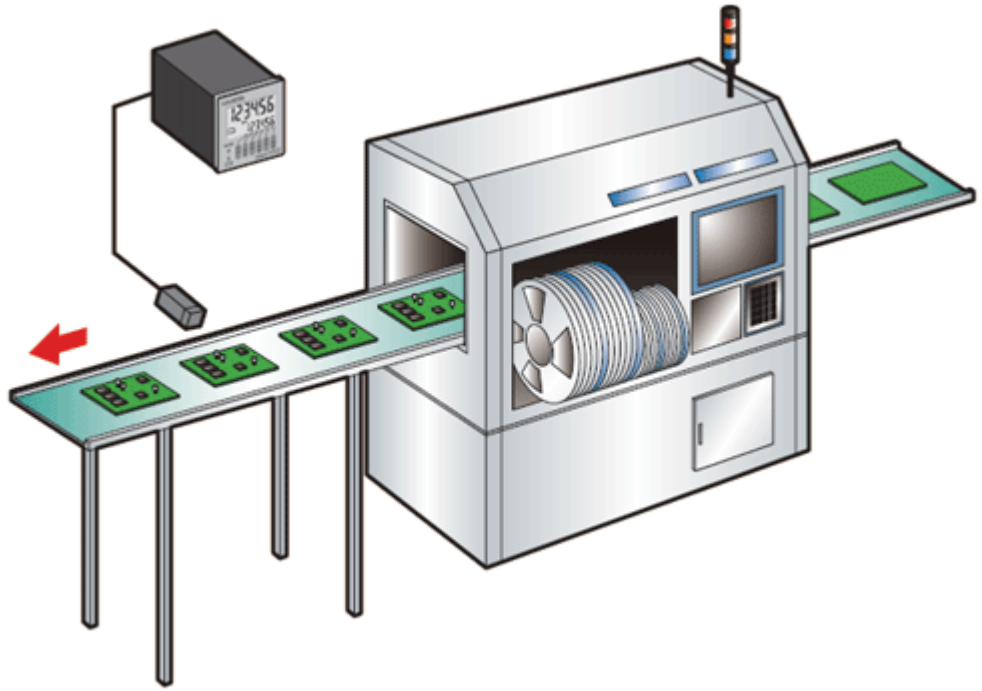
Class vs Object (static vs non-static)

Analogi: En Class är en produktionsmaskin. Maskinen är programmerad med ritningar för en sorts objekt, som den kan skapa efter önskemål (dvs anrop av konstruktor).

Maskinen kan ha ytterligare funktionalitet:

- Anrop av konstruktorer är meddelanden till maskinen att skapa nya objekt.
- Attribut märkta `static` tillhör maskinen, inte objekten den skapar.
- Anrop av metoder märkta `static` är meddelanden till maskinen, inte objekt.
- Allt som *inte* är märkt `static` (utom konstruktorer) är en del av ritningen för objekten som skapas.

Objekten vet alltid vilken maskin de kommer från (kan använda saker märkta `static`). Maskinen håller inte automatiskt reda på de objekt den skapat (kan inte använda saker som *inte* är märkta `static`).



Konstruktorer

- Ett anrop till en konstruktor är ett sorts metodanrop. Typ.

- Konstruktorer är hårt knutna till den exakta klassen.
  - Konstruktorer ärvs inte.
  - Måste explicit nämna klassnamnet när man anropar med `new`.
  - Effekten blir att konstruktorer inte kan använda polymorfism. Alls.

- Andra komponenter som behöver objekt av typen i fråga får ett beroende på den specifika klassen när de anropar konstruktorn.

Design Pattern: Factory Method

- 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.
    - 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.

Exempel: Factory Method

```
class MyObject {
  private MyObject() { ... }

  public static MyObject createMyObject(){
    MyObject obj = new MyObject();
    // possibly do smart stuff, and then...
    return obj;
  }
}

MyObject myObj = MyObject.createMyObject();
```

Design Pattern: Factory

- En *Factory* är en *klass* vars (statiska) metoder abstraherar (döljer) beteendet hos en eller flera konkreta konstruktorer.
  - Minskar beroendet på paket-interna konkreta klasser.
  - Ofta kopplad till en abstraktion (i.e. superklass eller interface) snarare än konkreta subklasser.
  - Kan ha ett mer sofistikerat beteende än en Factory Method, t ex genom att välja mellan konstruktorer (eller Factory Methods) från flera olika konkreta klasser.
  - Kan fortfarande vara så enkel att den bara delegerar till en viss konstruktor.
    - ”Framtidssäkring” precis som för Factory Method – men ger ännu svagare bindning.

Exempel: Factory

```
interface IObject { ... }

class MyObject implements IObject {
   MyObject(){ … }
}

public class IObjectFactory {
  public static IObject createIObject(){
    IObject obj = new MyObject();
    // possibly do smart stuff, and then
    return obj;
  }
}

IObject someObj = IObjectFactory.createIObject();
```

Övning: setup

- Börja från koden som finns att ladda ner från hemsidan.
  - Vår gamla kod från förra veckan är uppdaterad enligt vad vi gjorde på tidigare föreläsningen: Vi har nu ett separat sub-paket `tda551.polygons.polygon` som innehåller data-representationen. Vi har inga beroenden från detta paket på `DrawPolygons` som ligger utanför. Dock har vi ett beroende från `DrawPolygons` direkt på subklasserna `Triangle`, `Rectangle` och `Square`, via konkreta anrop av dessas konstruktorer. `DrawPolygons` har också fortfarande problem med att försöka göra för mycket – denna klass är inte uppdaterad helt.
  - Koden för `tda551.shapes` är nu fullt uppdaterad enligt Separation of Concern. Allt gemensamt beteende för polygoner är lagt i en abstrakt superklass `Polygon` (subklass till det ännu mer generella `Shape`), och de specifika klasserna `Rectangle` och `Triangle` innehåller nu enbart det som skiljer dem åt.

Övning

- Introducera en factory-class, med factory methods för att skapa trianglar, rektanglar och kvadrater.
  - Vilka argument behöver metoderna ta?
  - Vilken returtyp vill vi ge de olika metoderna?
  - Vår nya factory bör bo i sub-paketet `tda551.polygons.polygon`. Varför?
  - Rita ett UML-diagram över vår nya design. Vilket gränssnitt (publika klasser och metoder) har sub-paketet nu?
  - Kan du göra gränssnittet ännu mer abstrakt, dvs exponera ännu mindre av paketets konkreta interna implementation, utan att förlora funktionalitet?
- Nu är det dags att fundera över hur vi skulle kunna byta representation av polygoner i vårt `DrawPolygons`-program, till `tda551.shapes`.
  - Vilka refactoring-steg skulle behövas för att vi, när vi gör bytet, inte skulle behöva ändra i koden i `DrawPolygons`, mer än dess imports?
  - Vilka komponenter behöver läggas till när vi gör bytet, för att inte behöva ändra i `tda551.shapes`?
  - Implementera förändringarna, och byt paket.