3. Factory Pattern

1. Redegør for hvad software design pattern er
2. Redegør for opbygningen af Factory Method og Abstract Factory
    a. Abstrahere oprettelsen af objekter væk
    b. Nemmere oprettelse af objekter (med mange dependencies)
    c. Returnere bestemt klasse (ud fra Main input)
    d. Interface kender til implementeringer, Main kender kun til Interface
    e. Opretter objekter uden at kende til hvilke objekter der skal oprettes
    f. Factory Method
        i. Definere interface til at oprette objekter
        ii. Klasser bestemmer selv hvilke objekter de vil kende til
        iii. En klasse oprettes som sørger at oprette et ”Document”
            1. Kan være Short eller Long version. (se eksempel i disposition)
            2. returnere new Factory
        iv. Kan styres med en switch-case
        v. Se kodeeksempler i disposition
    g. Abstract Factory
        i.

3. Factory Method/Abstract Factory

Redegør for hvad et software design pattern er.

Et software design pattern er en general genbrugelig løsning til problemer der tit opstår i en given kontekst i software design. Det er ikke et færdigt design der kan laves direkte til kilde kode. Det er en beskrivelse eller skabelon for hvordan et problem kan løses i mange forskellige situationer. Det er formaliserede bedste praksisser som en programmør kan bruge til at løse problemer med.

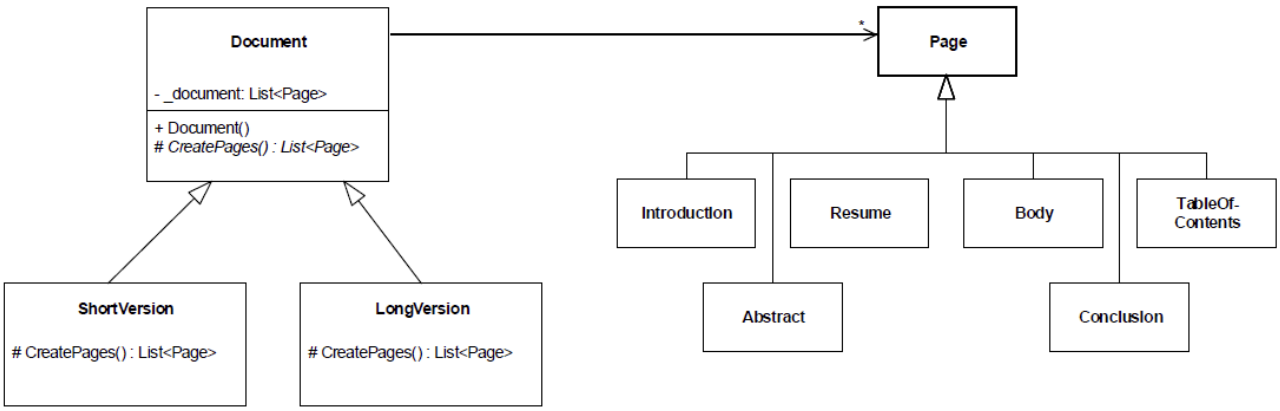
Redegør for opbygningen af GoF Factory Method og GoF Abstract Factory.

Ideen med factories er at **abstrahere oprettelsen af objekter væk** fra dem der bruger objekterne. Det er også lavet for at gøre det **nemmere at oprette objekter** der har mange dependencies på den rette måde. Den er lavet for at returnere en bestemt klasse, ud fra et input fra Main. Interfacet til afhængighederne kender til implementeringerne, men Main kender kun til interfacet. Det kaldes et factory da mønsteret opretter objekter uden nødvendigvis, at kende hvilke objekter, som skal oprettes eller hvordan.

Man kan bygge klasser på run-time.

Factory method

Her defineres et interface til at oprette et eller flere objekter, men vi lader klasserne der implementere interfacet bestemme hvilke objekter de vil initiere. Udover dette, så oprettes der en klasse, som sørger for at oprette dette "Document" med enten en **short** eller **longversion** ved at returnerer en **new factory** af enten short eller longversion af dokument**.** Dette kan gøres med enten en switch case eller en if.



Her ses koden for document hvor den der implementere dette ”interface” så overskriver createpages og tilføjer de pages til listen som de ønsker.

```
class Document
{
  private list<Page> _pages;
  public Document()
  {
    pages = CreatePages();
  }

  protected abstract List<Page> CreatePages();
}
```

```
class ShortVersion : Document
{
  protected override List<Page> CreatePages()
  {
    var pages = new List<Page>();
    pages.Add(new Abstract());
    pages.Add(new Resume());
    return pages;
  }
}
```

Page er altså vores product, et interface, de forskellige typer som fx abstract er concrete product som arver fra Page. Vores Document er Creator og vores long eller short version er concrete creator. Vores factory method afleverer en liste af vores product.

Alternativt eksempel

```
class A {
    public void doSomething() {
        Foo f = makeFoo();
        f.whatever();
    }

    protected Foo makeFoo() {
        return new RegularFoo();
    }
}

class B extends A {
    protected Foo makeFoo() {
        //subclass is overriding the factory method
        //to return something different
        return new SpecialFoo();
    }
}
```

Klassen ’RegularFoo’ er en nedarvning af ’Foo’, og fordi dette er tilfældet, kan man returnere den som ’Foo’.

Abstract Factory

Bruges når der er brug for at oprette familier af relaterede eller afhængige objekter uden at specificere deres konkrete klasser.

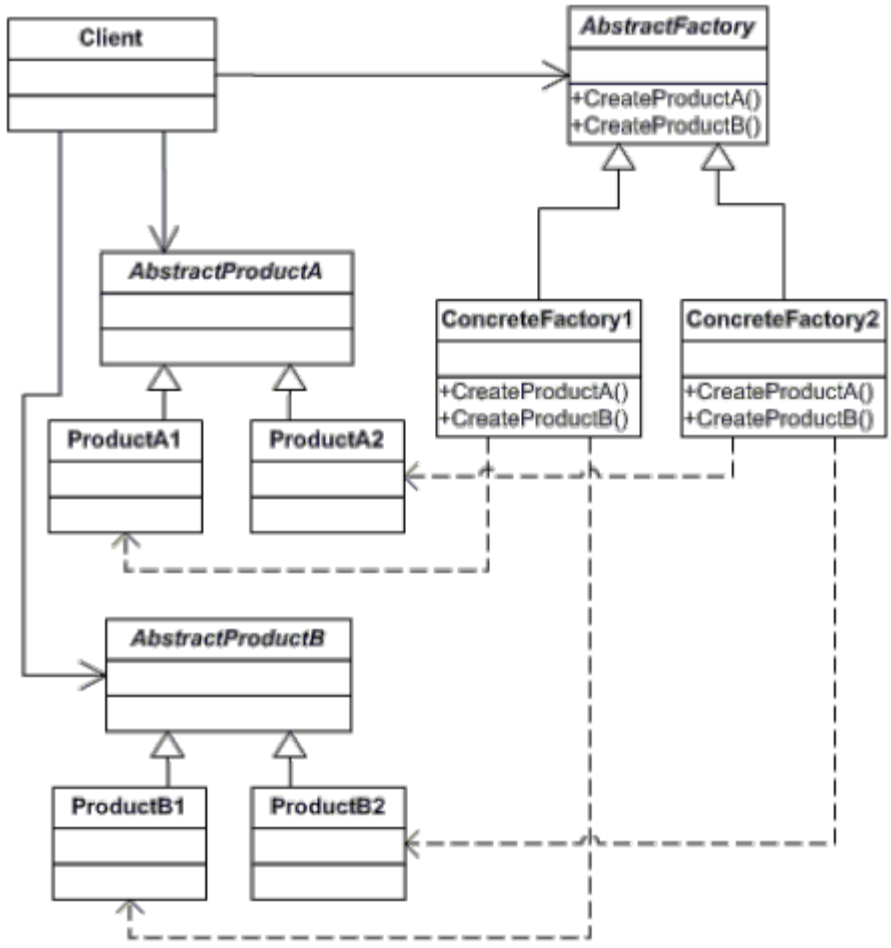
Brugeren opretter en konkret implementering af den abstrakte fabrik som så bruges til at få alle de konkrete implementeringer.

"Client" bruger AbstractFactory og AbstractProductA og AbstractProductB, for at kunne få oprettet sine objekter. Klienten skal vide om han ønsker noget oprettet fra enten AbstractProductA eller AbstractProductB.

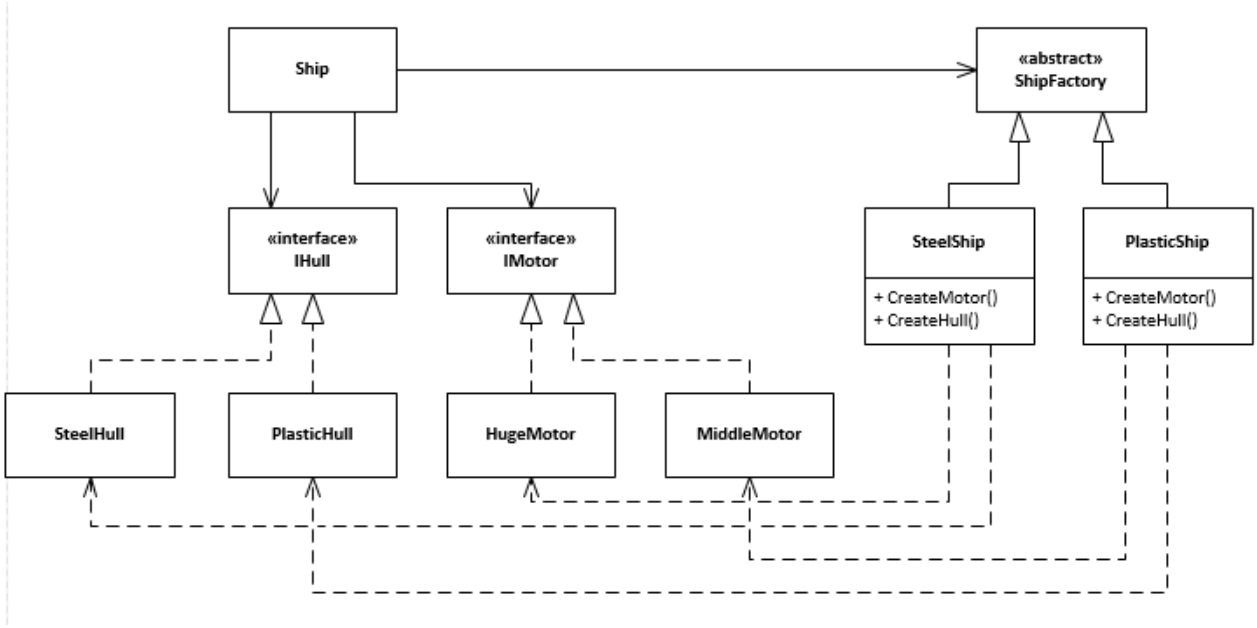
"AbstractProductA&B" bruges til at kunne definer metoderne til "ProductA" og "ProductB". Da det er et interface kan der tilføjes så mange products, som man ønsker, så længe de indeholder implementering af de metoder, som er defineret i "AbstractProduct"

"AbstractFactory" er et interface, som bruges til at kunne oprette "ConcreteFactories"

"ConcreteFactory" implementerer interfacet fra AbstractFactory, så der kan oprettes nogle produkter (ProductA og ProductB).



Giv et designeksempel på anvendelsen af GoF Abstract Factory



With the Factory pattern, you produce implementations (Apple, Banana, Cherry, etc.) of a particular interface -- say, IFruit.

With the Abstract Factory pattern, you produce implementations of a particular Factory interface -- e.g., IFruitFactory. Each of those knows how to create different kinds of fruit.

```
class A {
    private Factory factory;

    public A(Factory factory) {
        this.factory = factory;
    }

    public void doSomething() {
        Foo f = factory.makeFoo();
        f.whatever();
    }
}

abstract class Factory {
    Foo makeFoo();
}
```

Til dette eksempel vil der være en klasse som nedarver fra 'Factory', og som dermed laver en instans af 'makeFoo'. Når man bruger denne Abstract factory pattern, så laver man en factory, der indeholder alle de nødvendige informationer til at kunne oprette et objekt 'Foo'. Dette betyder at hvis man har forskellige typer af Foo's, kan man oprette en factory til hver type, og dermed ikke tænke på om den bruger den gør det rigtige.

https://stackoverflow.com/questions/5739611/differences-between-abstract-factory-pattern-and-factory-method