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

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.

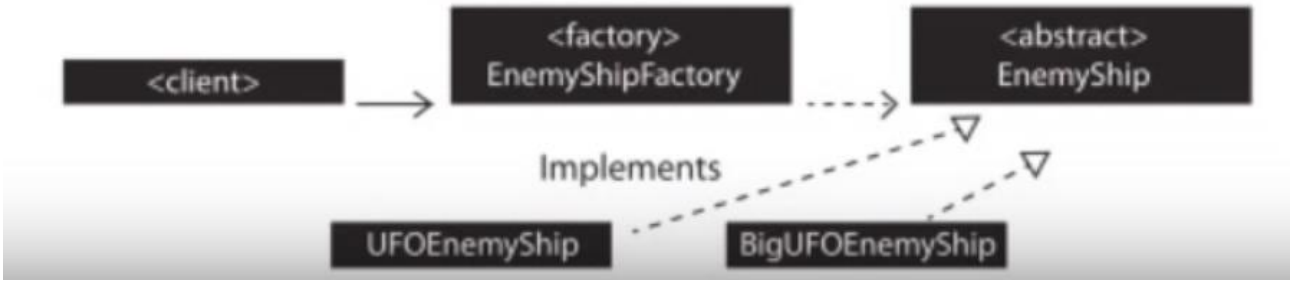
Redegør for opbygningen af GoF Factory Method og GoF Abstract Factory.

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.

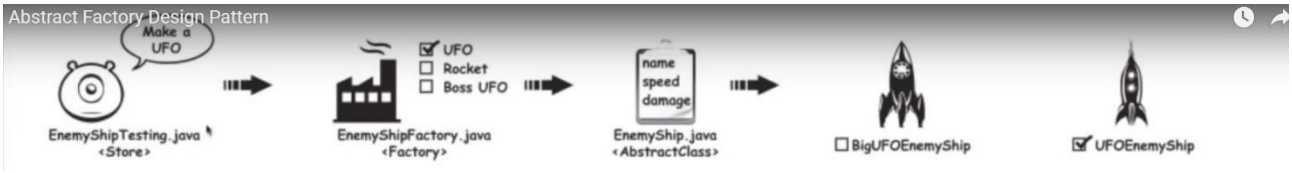
Factory method

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 initialisere factory(EnemyShipFactory), som kender interfacet(EnemyShip) som implementeres af klasserne(UFOEnemyShip og BigUFOEnemyship)



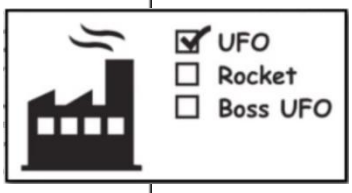
Solid

S: Overholdes da factory kun opretter objekter, og hver concreteObject implementere kun et objekt.
O: Åben for flere concreteObjects men lukket for at ændre i de eksisterende.
L: **IKKE** opfyldt da factory ikke kan substitueres
I : **IKKE** opfyldt da der ikke er en factory interface.
D: **IKKE** opfyldt da der ikke er abstraktion lag mellem client & factory.




```
public static void main(String[] args){

  // Create the factory object
  EnemyShipFactory shipFactory =
      new EnemyShipFactory();

  // Enemy ship object

  EnemyShip theEnemy = null;

  Scanner userInput = new Scanner(System.in);

  System.out.print("What type of ship? (U / R / B)");

  if (userInput.hasNextLine()){

    String typeOfShip = userInput.nextLine();

    theEnemy = shipFactory.makeEnemyShip(typeOfShip);

    if(theEnemy != null){

      doStuffEnemy(theEnemy);

    } else System.out.print("Please enter U, R,
           or B next time");
}
```

```
public EnemyShip makeEnemyShip(String newShipType)
{

  EnemyShip newShip = null;

  if (newShipType.equals("U")){

    return new UFOEnemyShip();

  } else

  if (newShipType.equals("R")){

    return new RocketEnemyShip();

  } else

  if (newShipType.equals("B")){

    return new BigUFOEnemyShip();

  } else return null;

}
```



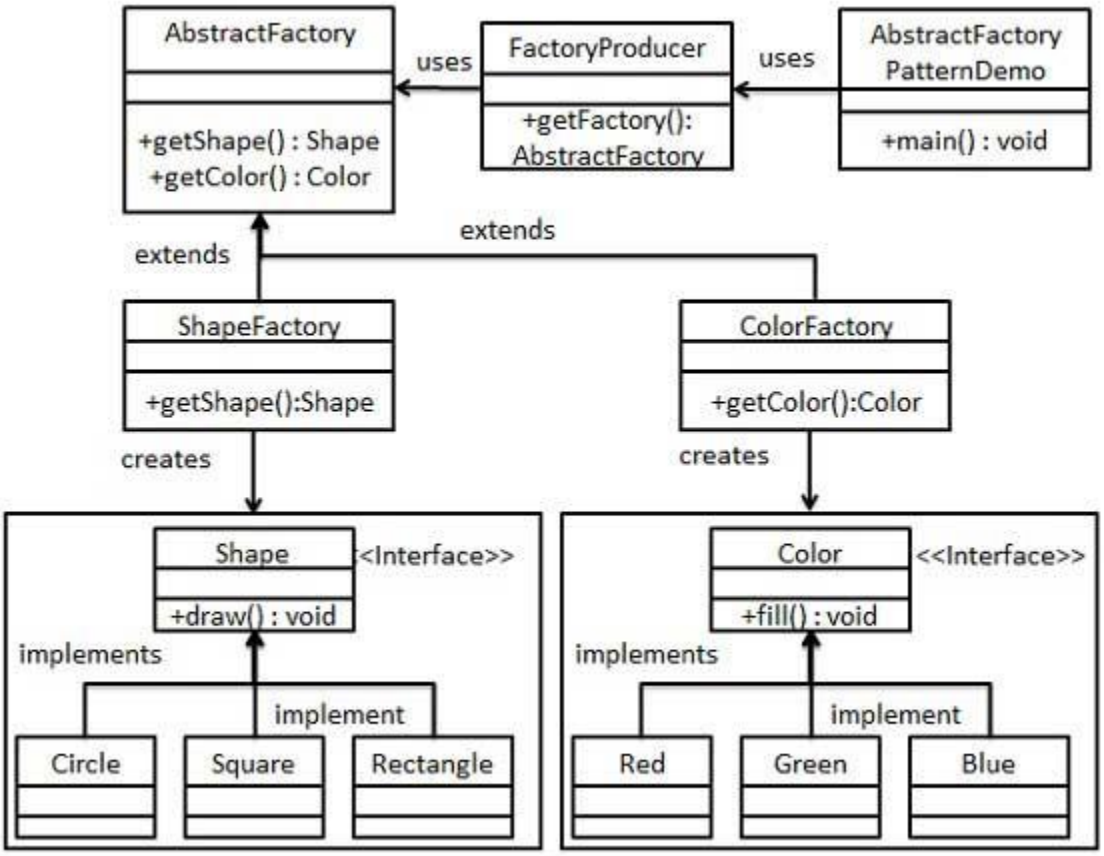

```
public abstract class EnemyShip
{

  private String name;
  private double speed;
  private double damage;

  public String getName()
  { return name; }

  public void setName(String newName)
  { name = newName; }

  public double getDamage()
  { return amtDamage; }

  public void setDamage(double newDamage)
  { amtDamage = newDamage; }
}
```


```
public class BigUFOEnemyShip extends EnemyShip
{
  public BigUFOEnemyShip(){

    setName("Big UFO Enemy Ship");
    setDamage(40.0);
    setSpeed(10.0);
  }
}
```

```
public class UFOEnemyShip extends EnemyShip
{
  public UFOEnemyShip(){

    setName("UFO Enemy Ship");
    setDamage(20.0);
    setSpeed(20.0);
  }
}
```

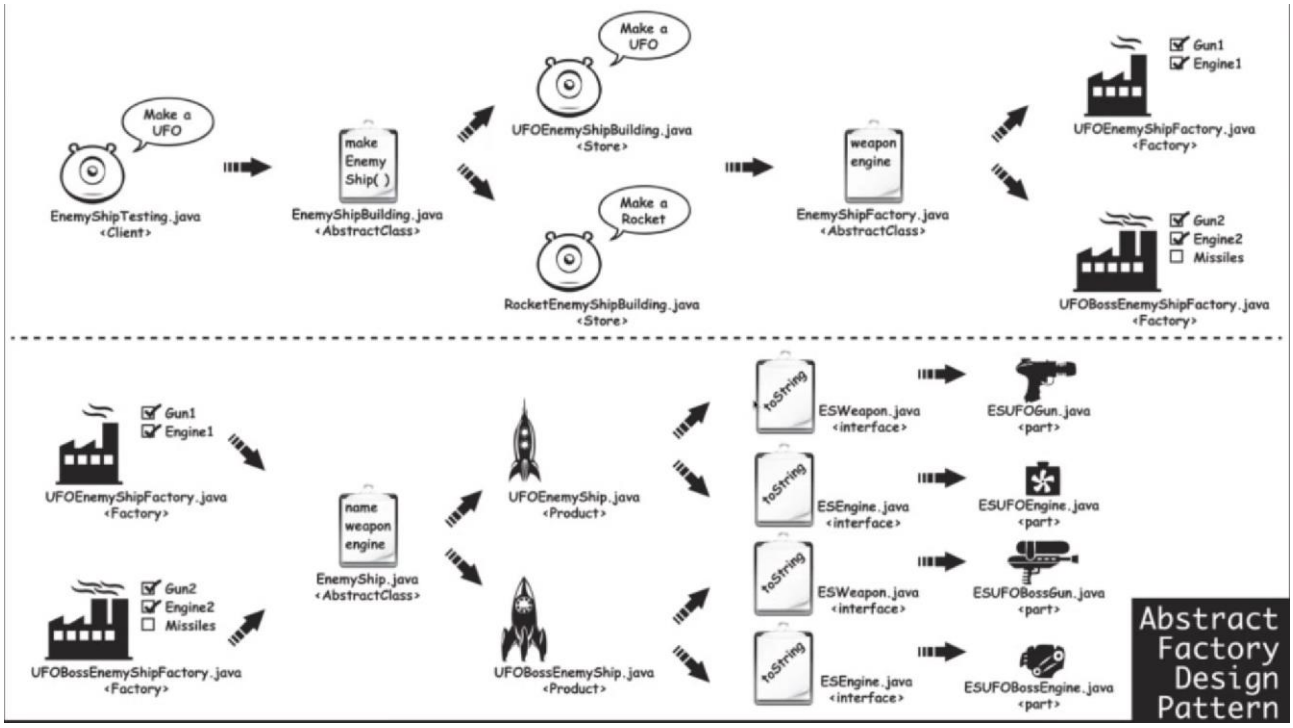
Abstract Factory

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



Vi har en superfactory som producerer andre factories baseret på AbstractFactory, hver af disse fabrikker kan producere en et produkt af en given type baseret på den enkelte fabriks abstrakte produkt interface. Hvor hvert konkret produkt har sin egen implementering, der overholder Interfacet.

Giv et designeksempel på anvendelsen af GoF Abstract Factory



Se “Derek Banas Abstract factory pattern” video for forklaring.

https://www.youtube.com/watch?v=xbjAsdAK4xQ&t=153s

Related pattern

Abstract Factory (99) is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.

Factory methods are usually called within Template Methods (360). In the document example above, NewDocument is a template method.

Prototypes (133) don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.