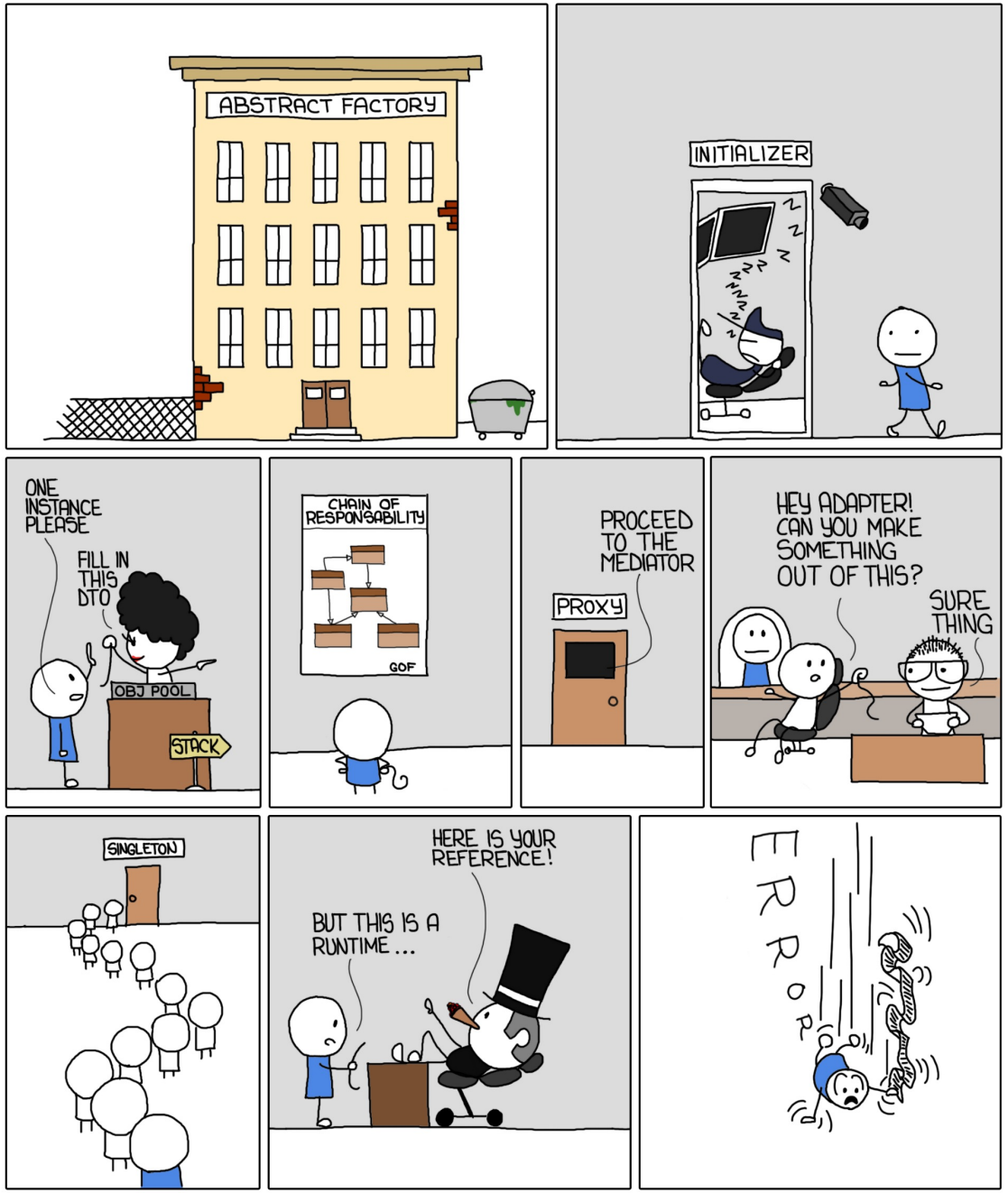
DESIGN PATTERNS - BUREAUCRACY



Software design patterns

GoF Factory Method and
GoF Abstract Factory

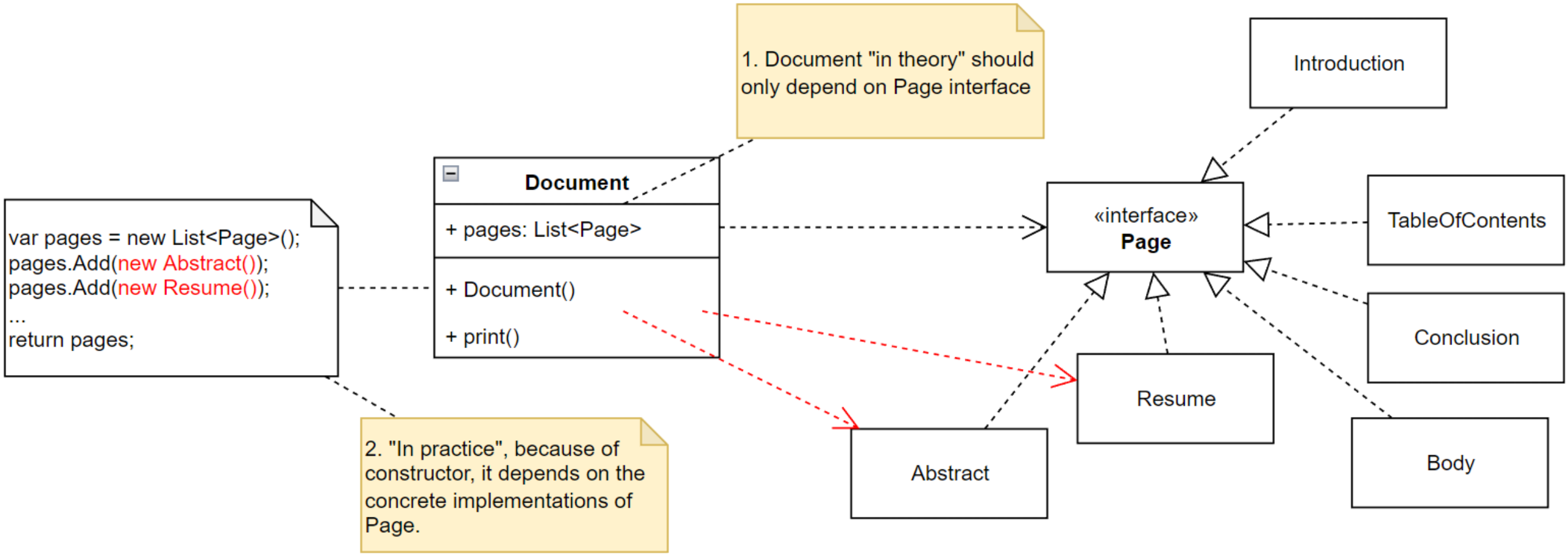
version: 1.0.2

AARHUS
UNIVERSITY
SCHOOL OF ENGINEERING

Factories in software architecture

- The general goals of factories are:
  - to separate the creation of an object from its use – SRP
  - Make creation code open for extension but closed for modification (OCP)
- Factories come in many different flavors – we will look at two classics:
  - GoF Factory Method: Define an interface for creating an object, but let the classes that implement the interface decide which object to instantiate.
  - GoF Abstract Factory: Define an interface for creating families of related or dependent objects without specifying their concrete classes

Motivation

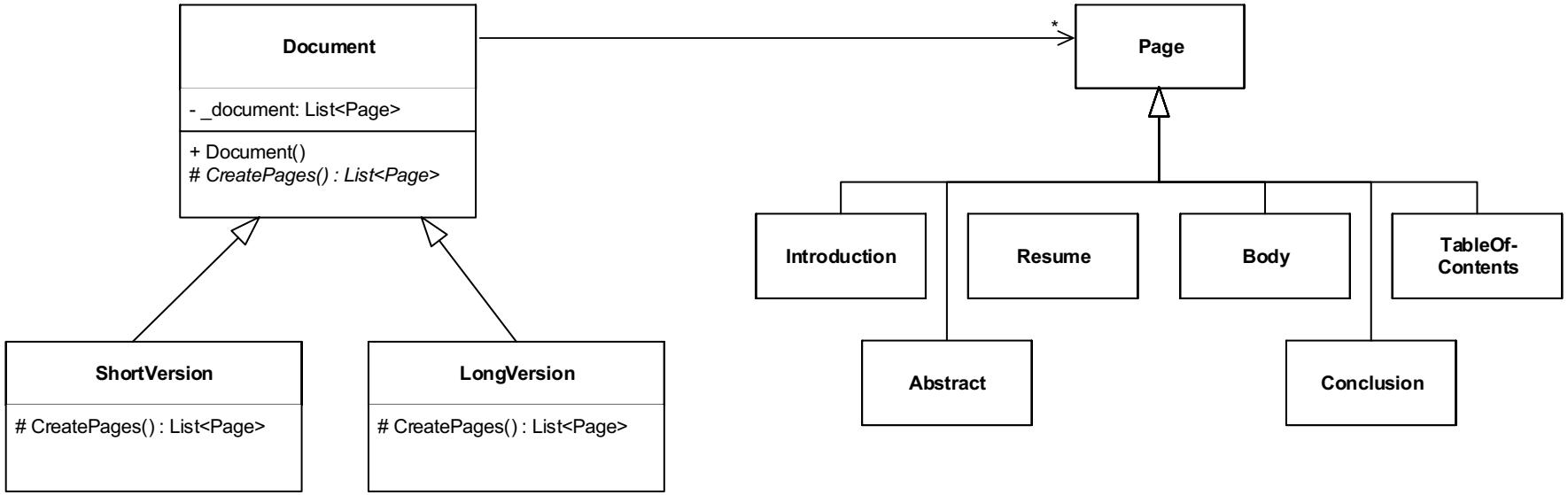


Source

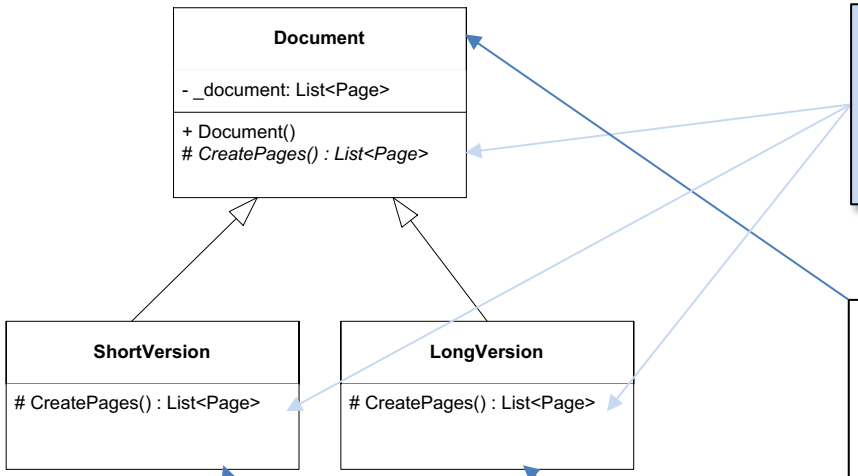
GoF Factory Method

GoF Factory Method

- *Factory Method*: Define an interface for creating an object(s), but let the classes that implement the interface decide which object(s) to instantiate.



GoF Factory Method



Factory Method: Define an interface (`CreatePages()`) for creating an object(s), but let the classes that implement the interface decide which object(s) to instantiate.

```
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;
  }
}
```

```
class LongVersion : Document
{
  protected override List<Page> CreatePages()
  {
    var pages = new List<Page>();
    pages.Add(new Abstract());
    pages.Add(new TableOfContents());
    pages.Add(new Introduction());
    pages.Add(new Body());
    pages.Add(new Conclusion());
    return pages;
  }
}
```

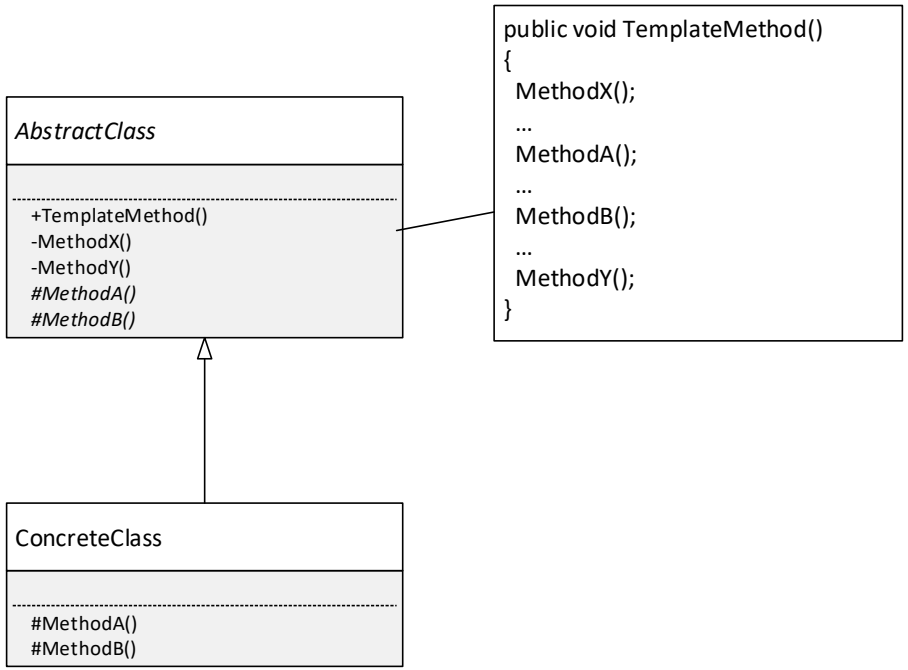
Recap Goals:

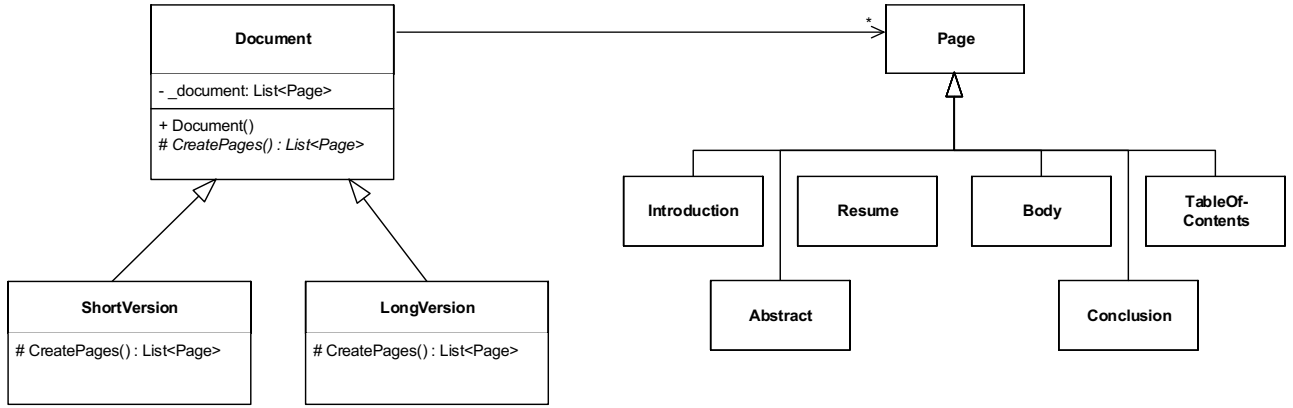
to separate the creation of an object from its use – SRP ✓

Make creation code open for extension but closed for modification (OCP) ✓

Discussion

What's the difference between template method and factory method patterns?

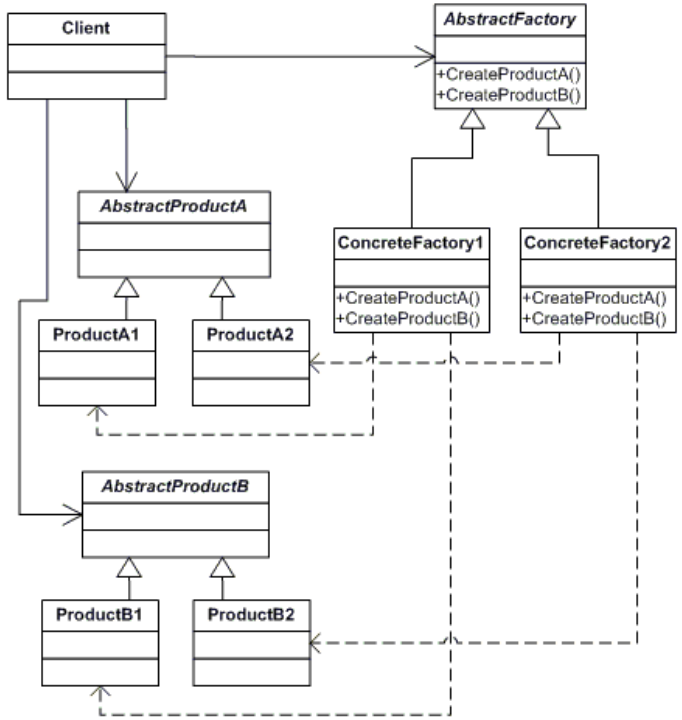




GoF Abstract Factory

GoF Abstract Factory

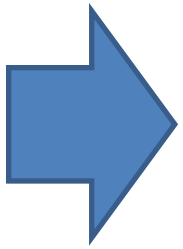
- Define an interface for creating *families* of related or dependent objects without specifying their concrete classes



GoF Abstract Factory

- An example system: The `WeighingSystem`

```
public class FøtexWeighingSystem
{
    private FøtexWeighingUnit _weighingUnit;
    private FøtexPrinter _printer;
    private FøtexDisplay _display;

    public WeighingSystem()
    {
        _weighingUnit = new FøtexWeighingUnit();
        _printer = new FøtexPrinter();
        _display = new FøtexDisplay();
    }
}
```

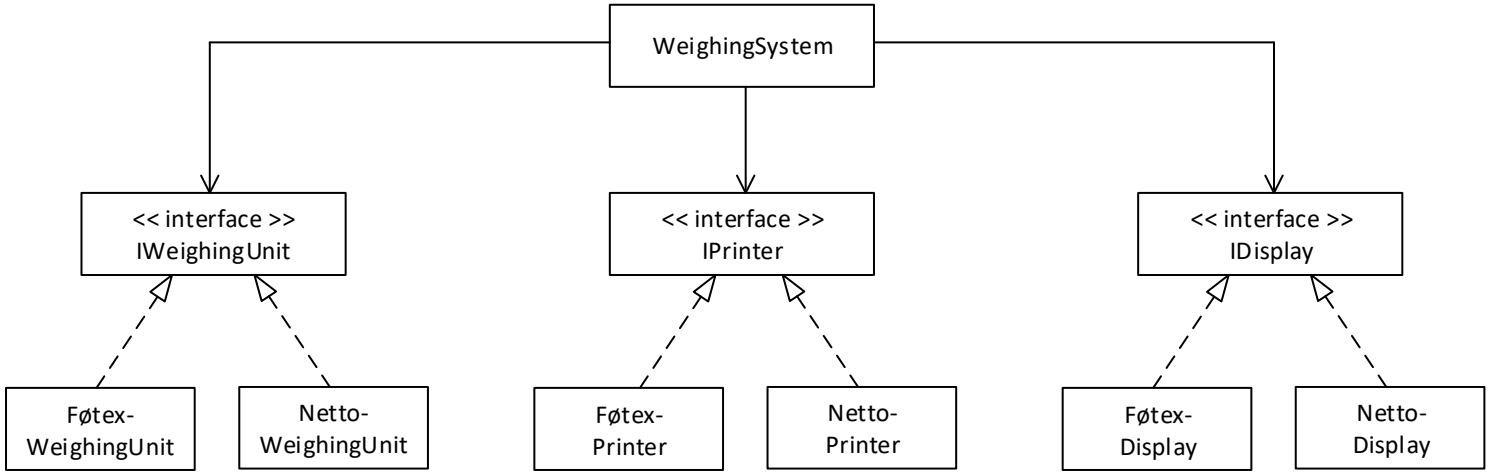


```
public class WeighingSystem
{
    private IWeighingUnit _weighingUnit;
    private IPrinter _printer;
    private IDisplay _display;

    public WeighingSystem(
              IWeighingUnit weighingUnit,
              IPrinter printer,
              IDisplay display)
          {
              _weighingUnit = weighingUnit;
              _printer = printer;
              _display = display;
          }
      }
```

```
public class Application
{
    public static void Main()
    {
        var føtexWs =
                    new FøtexWeighingSystem();
    }
}
```

```
public class Application
{
    public static void Main()
    {
        // Create a Føtex weight
        var føtexWs = new WeighingSystem(
              new FøtexWeighingUnit(),
              new FøtexPrinter(),
              new FøtexDisplay());
      }
      }
```

WeighingSystem after DIP is applied



Question: how many different WeighingSystems can now be created?

Without abstract factory

- Creation of `WeighingSystem` variants is complex and error prone

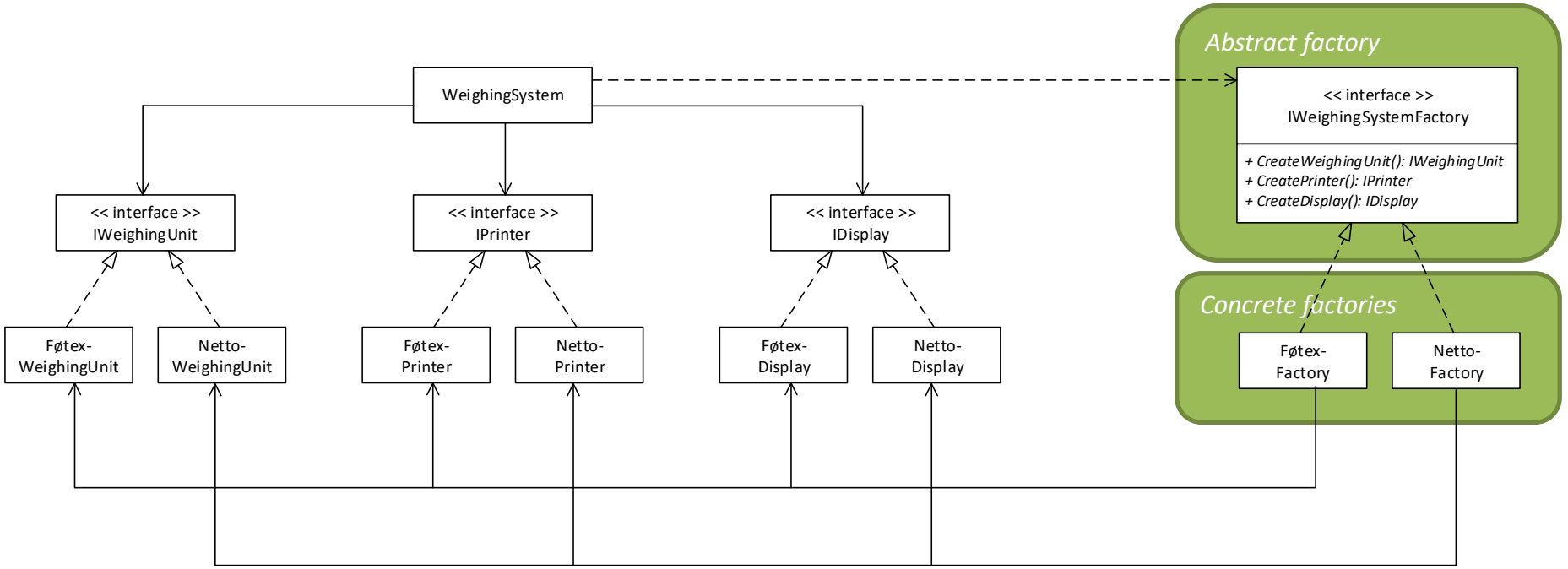
```
public class Application
{
    public static void Main()
    {
        // Create a Føtex weight
        var føtexWs = new WeighingSystem(
              new FøtexWeighingUnit(),
              new FøtexPrinter(),
              new FøtexDisplay());
       }
       }
```

```
public class Application
{
    public static void Main()
    {
        // Create a Netto weight
        var nettoWs = new WeighingSystem(
              new NettoWeighingUnit(),
              new NettoPrinter(),
              new FøtexDisplay());
       }
       }
```

```
public class WeighingSystem
{
    private IWeighingUnit _weighingUnit;
    private IPrinter _printer;
    private IDisplay _display;

    public WeighingSystem(
              IWeighingUnit weighingUnit,
              IPrinter printer,
              IDisplay display)
          {
              _weighingUnit = weighingUnit;
              _printer = printer;
              _display = display;
          }
      }
```

Using GoF Abstract Factory

- By using GoF Abstract Factory we can isolate the complex creation of object families in factory classes …



Using GoF Abstract Factory

```
public class Application
{
  public static void Main()
  {
    // Create a Føtex weight
    var føtexWs = new WeighingSystem(new FøtexFactory());
  }
}
```

```
public class Application
{
  public static void Main()
  {
    // Create a Netto weight
    var nettoWs = new WeighingSystem(new NettoFactory());
  }
}
```

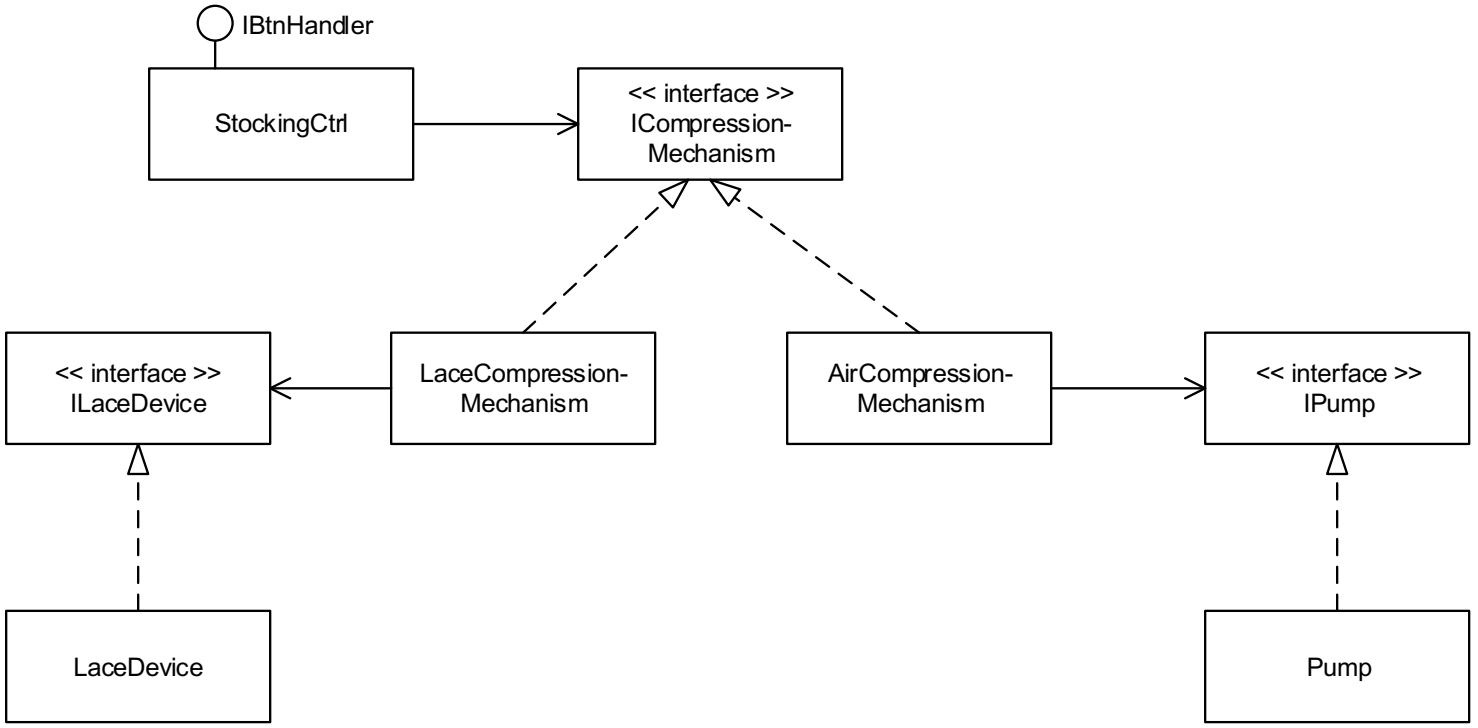
```
public class FøtexFactory : IWeighingSystemFactory
{
    public IWeighingUnit CreateWeighingUnit() {
        return new FøtexWeighingUnit();
    }
    public IDisplay CreateDisplay() {
        return new FøtexDisplay();
    }
    public IPrinter CreatePrinter() {
        return new FøtexPrinter();
    }
}
```

```
public class NettoFactory : IWeighingSystemFactory
{
    public IWeighingUnit CreateWeighingUnit() {
        return new NettoWeighingUnit();
    }
    public IDisplay CreateDisplay() {
        return new NettoDisplay();
    }
    public IPrinter CreatePrinter() {
        return new NettoPrinter();
    }
}
```

```
public class WeighingSystem
{
    private IWeighingUnit _weighingUnit;
    private IPrinter _printer;
    private IDisplay _display;

    public WeighingSystem(IWeighingSystemFactory factory)
    {
        _weighingUnit = factory.CreateWeighingUnit();
        _printer = factory.CreatePrinter();
        _display = factory.CreateDisplay();
    }
}
```

GoF Abstract Factory

- GoF Abstract Factory is especially handy when the family of products *depend* on each other in different ways
- Consider the `CompressionStocking` exercise (SOLID 2)



GoF Abstract Factory

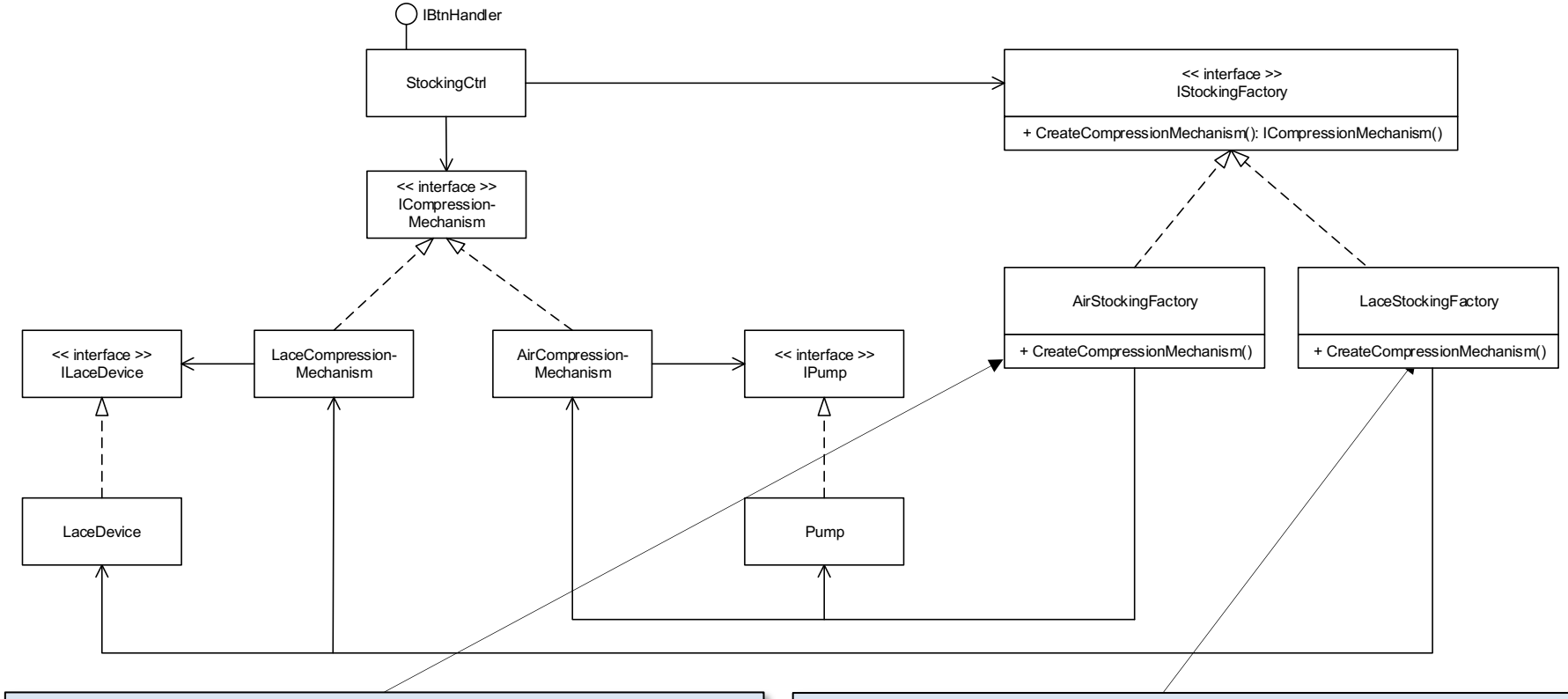
- Creating the different versions of the compression stocking within `StockingCtrl` requires changes in the creation of the mechanisms and devices.

```
class StockingCtrl
{
  enum CompressionMethod { AIR, LACES }
  ICompressionMechanism _compressionMechanism;



  public StockingCtrl(CompressionMethod method)
  {
    switch(method)
    {
      case AIR:
        _compressionMechanism = new AirCompressionMechanism(new Pump(), 5000, 2000);
        break;

      case LACES:
        _compressionMechanism = new LaceCompressionMechanism(new LaceDevice(), 40, 100);
        break;
    }
  }
```

- Adding new compression methods is a mess – violates OCP

GoF Abstract Factory

- Using GoF Abstract Factory, we can encapsulate creation details and dependencies and remove them from `StockingCtrl`:




```
ICompressionMechanism CreateCompressionMechanism()
{
  // Connect and return AIR compression mechanism parts
  return new AirCompressionMechanism(new Pump(), 5000, 2000);
}
```

```
ICompressionMechanism CreateCompressionMechanism()
{
  // Connect and return LACE compression mechanism parts
  return new LaceCompressionMechanism(new LaceDevice(), 40, 100);
}
```

GoF Abstract Factory

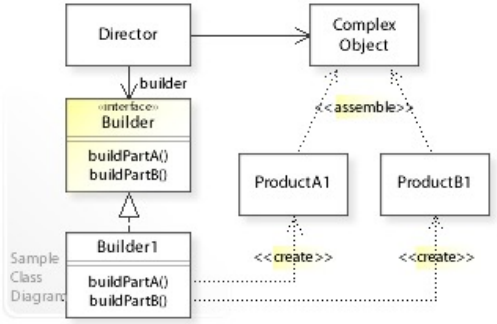
- Injecting the factory makes `StockingCtrl` oblivious to the compression mechanism (air or laces) and to the constructor arguments

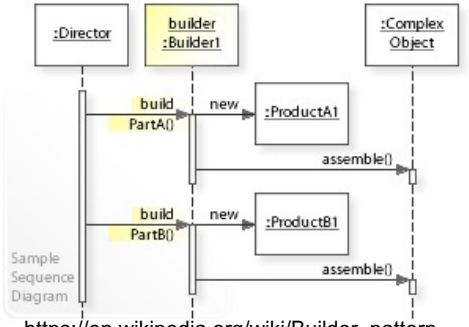
```
class StockingCtrl
{
  ICompressionMechanism _compressionMechanism;

  public StockingCtrl(IStockingFactory factory)
  {
    _compressionMechanism = factory.CreateCompressionMechanish();
  }
}
```

- Adding new compression methods is now done by adding new factories – adheres to OCP!

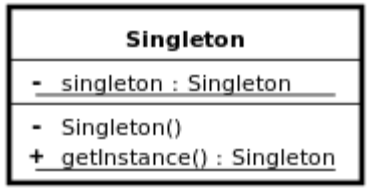
Other factories

- Builder
  - *The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so the same construction process can create different representations.*
- Singleton
  - *Ensure that only one instance of a class is created.*
  - *Provide a global point of access to the object.*
- Prototype
  - *Specifying the kind of objects to create using a prototypical instance.*
  - *Creating new objects by copying this prototype.*

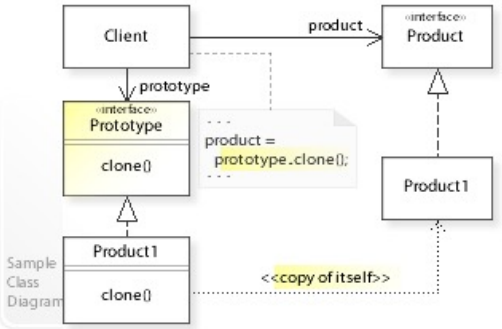


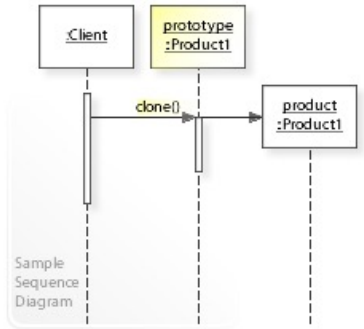


https://en.wikipedia.org/wiki/Builder_pattern



https://en.wikipedia.org/wiki/Singleton_pattern





https://en.wikipedia.org/wiki/Prototype_pattern

Questions?