The factory pattern in Java

From naive to functional

The factory pattern

About

The factory pattern is one of the most used design patterns and comes in different flavours: factory, factory method, or abstract factory.

There are more approaches you can choose from, when implementing the pattern, some of them are covered in this presentation.

As always, it helps to know them all and to choose the right tool for the job.

This presentation contains a simple, naive implementation, a more advanced one, a functional one and an object oriented example.

The code samples

Code examples are written in Java. All the samples are stored in a git repository and can be found here:

http://github.com/mw1980/onfactorypattern

The examples are kept easy, just to make a point. They are probably not ready to use in a real world productive scenario.

The factory pattern

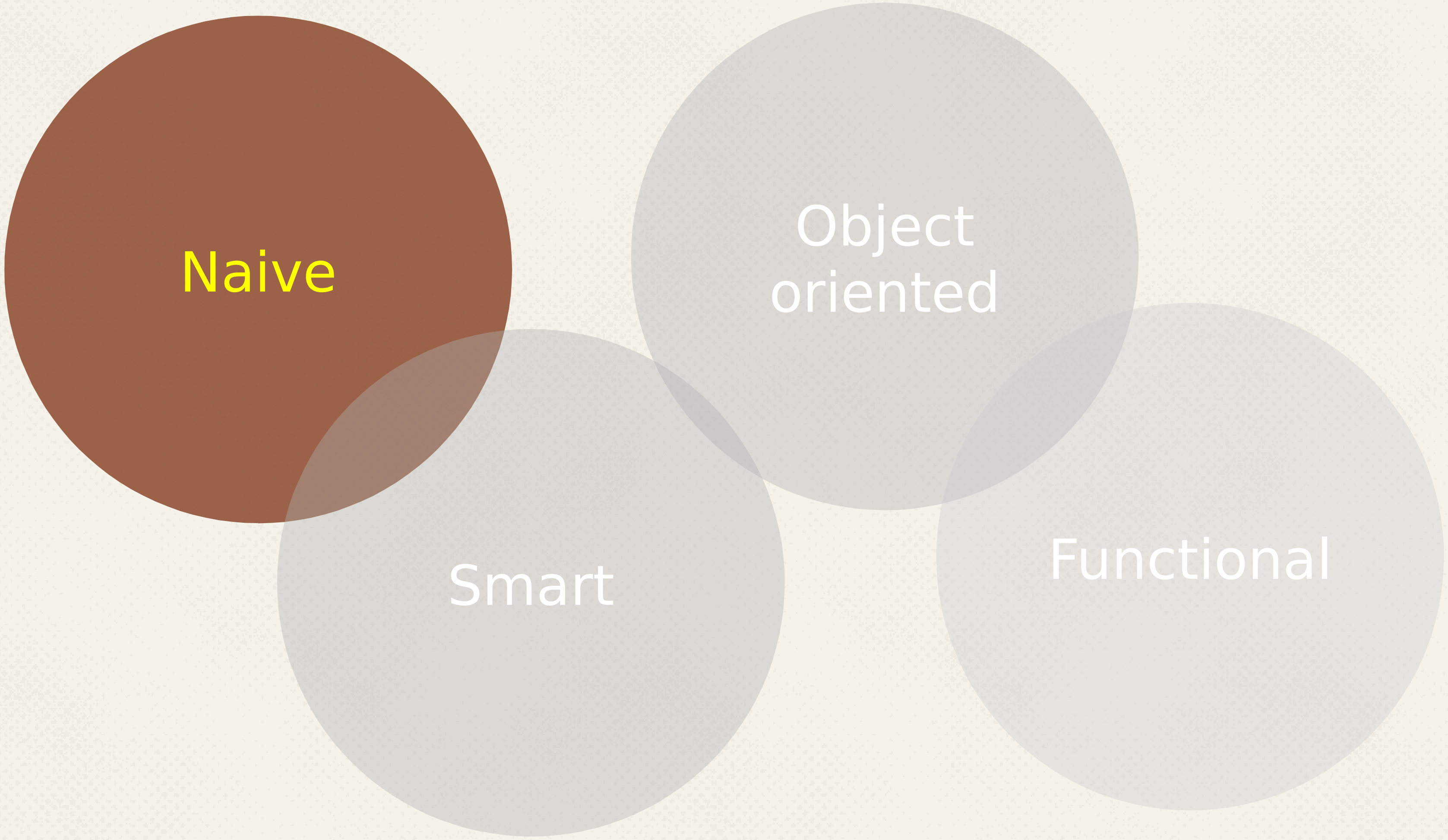
The idea:

In the factory pattern there are two actors involved: the client and the factory

Given a Product interface and some concrete implementations, the client knows only how to identify a product, for instance by type, or id

The complete logic is stored in the factory and therefore decoupled from the client

Factory pattern approaches



Factory pattern, naive implementation

The basics

The factory pattern

Naive implementation, the code

```
The factory:
Product createProductOfType(ProductType type) {
       switch (type) {
           case FIRST:
               return new FirstProduct();
           case SECOND:
               return new SecondProduct();
           default: // throw some runtime exception
}
The client:
new Factory().createProductOfType(ProductType.FIRST)
```

https://github.com/mw1980/onfactorypattern/ in the “naive” package

The factory pattern

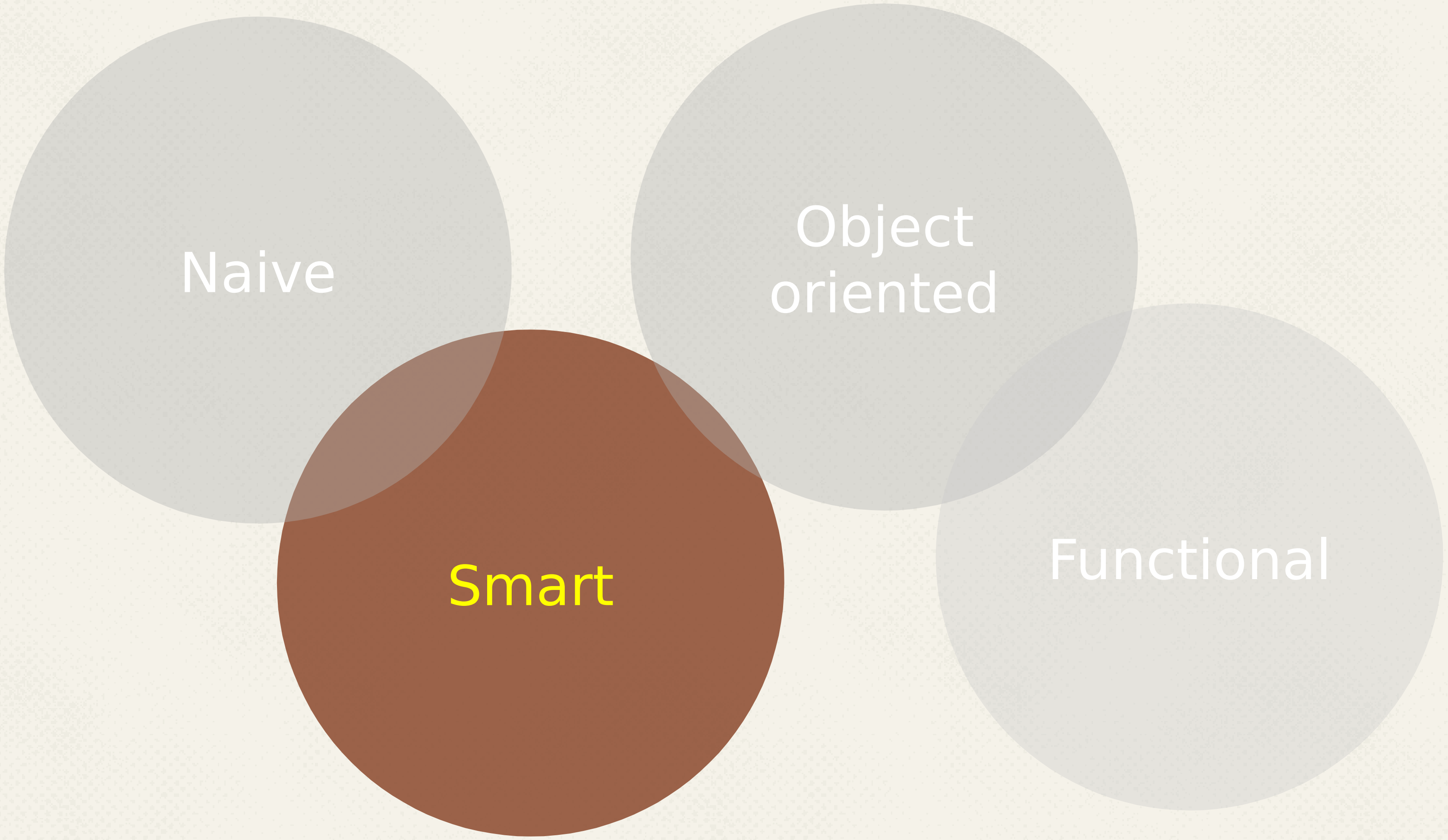
Naive implementation, some thoughts

The problem with the naive implementation is that it breaks the open-close-principle

Every time there is another product added, or removed from factory portfolio, the factory class must be updated

Therefore the class is by design open for modifications, which requires constant maintenance

Factory pattern approaches



Factory pattern, smart products

Power to the products

The factory pattern

Smart products implementation, the idea

To avoid breaking the open close principle you might consider intelligent products, that register themselves to the factory

The factory stores a list of supported products and exposes a method to register new products.

Each product class contains a static initializer block, where it adds itself to the supported products list

The factory pattern

Smart products implementation, the code

```
The product:
public class FirstSmartProduct implements Product {
static {
Factory.instance()
       .registerProduct(ProductType.FIRST,
                        new FirstSmartProduct()
        );
}
//… other methods
}
```

https://github.com/mw1980/onfactorypattern/ , the “smart” package

The factory pattern

Smart products implementation, the code

```
The factory:
class Factory {
    private static Factory instance;
    private Map<ProductType, Product> registered = new
HashMap<>();
    static Factory instance() {
        if (instance == null) {instance = new Factory();}
        return instance;
    }
    void registerProduct(ProductType type, Product product){
         registered.put(type, product);
    }
```

https://github.com/mw1980/onfactorypattern/ , the “smart” package

The factory pattern

Smart products implementation, the code

```
The factory:
    Product create(ProductType type) {
        if (instance.registered.containsKey(type)) {
            return instance.registered.get(type);
        } else {
            throw new IllegalArgumentException("No product
available with type" + type);
        }
    }
```

https://github.com/mw1980/onfactorypattern/ , the “smart” package

The factory pattern

Smart products implementation, the code

```
The main method:
    public static void main(String[] args) {
try {
    Class.forName("complete.path.to.FirstSmartProduct");
    Class.forName("complete.path.to.SecondSmartProduct");
} catch (ClassNotFoundException any) {...}

Product first = Factory.instance().create(ProductType.FIRST);
}
```

https://github.com/mw1980/onfactorypattern/ , the “smart” package

The factory pattern

Smart products implementation, some thoughts

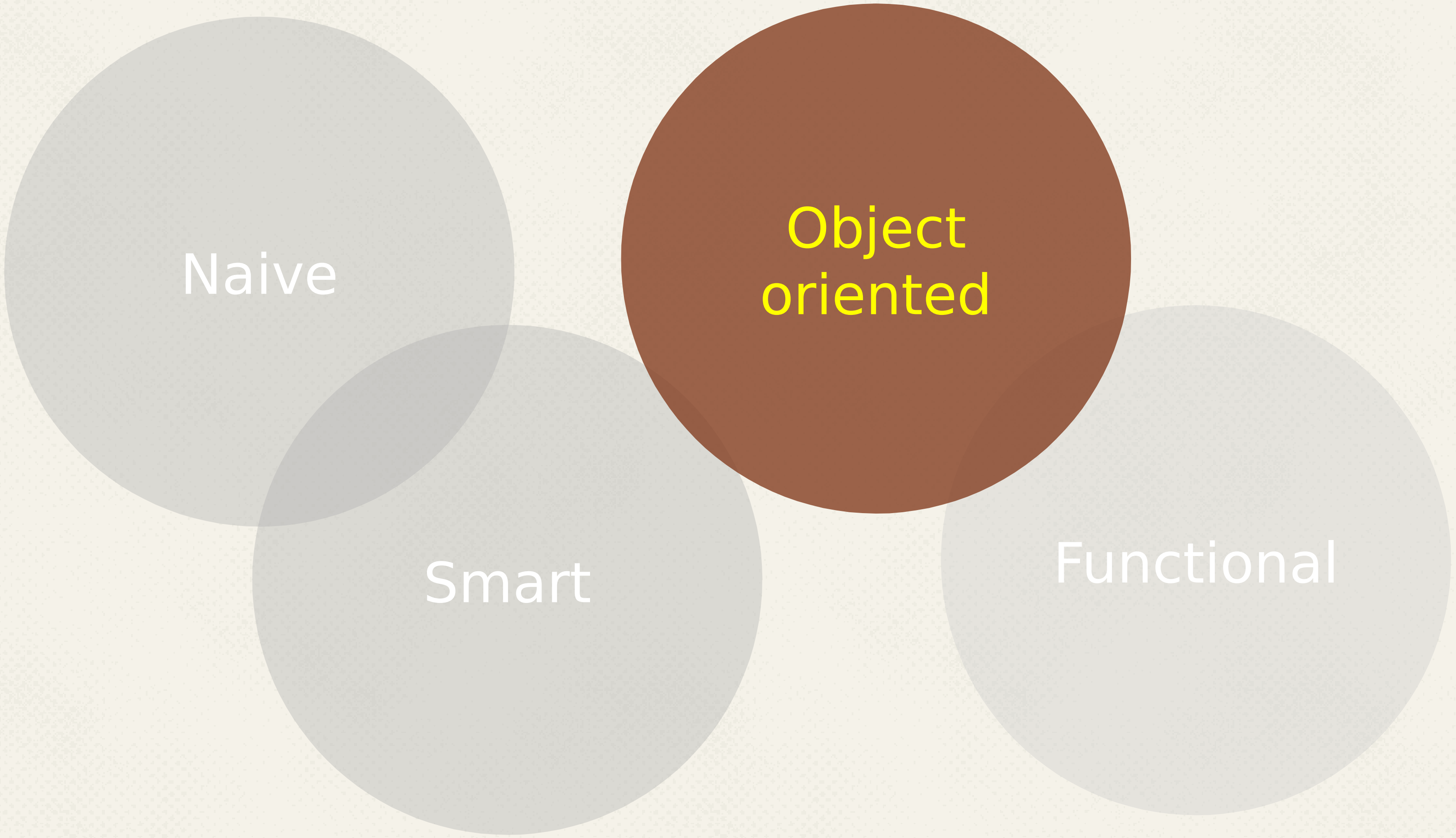
The factory is a singleton. On a love-and-hate scale, the singleton pattern is way closer to the “hate” end among developers

The static initializers are generally avoided, initializing the fields in the constructor is the readable and intuitive solution

The product static initializers must run before the factory creates any product, therefore the need for the brittle:

Class.forName(“pathAsPlainString”) calls

Factory pattern approaches



Object oriented

Magic beans

The factory pattern

Object oriented implementation, the idea

In a real-world scenario you most likely use dependency injection and you probably already have a container that stores all your beans

Therefore all the products are already available in the container for any factory implementation

The factory can find the products in the container, eliminating the need to maintain the registered products list, like in the smart products example

The factory pattern

Object oriented implementation, the code

```
The factory:
public class TextileFactory implements Factory,
ApplicationContextAware {
public Product createProduct(ProductType type) {
 return this.applicationContext.getBeansOfType(Product.class)
        .values().stream()
        .filter(product -> type == product.productType())
        .findFirst()
        .orElseThrow(() -> new IllegalArgumentException(..));
}
```

https://github.com/mw1980/onfactorypattern/ , the “oo” package

The factory pattern

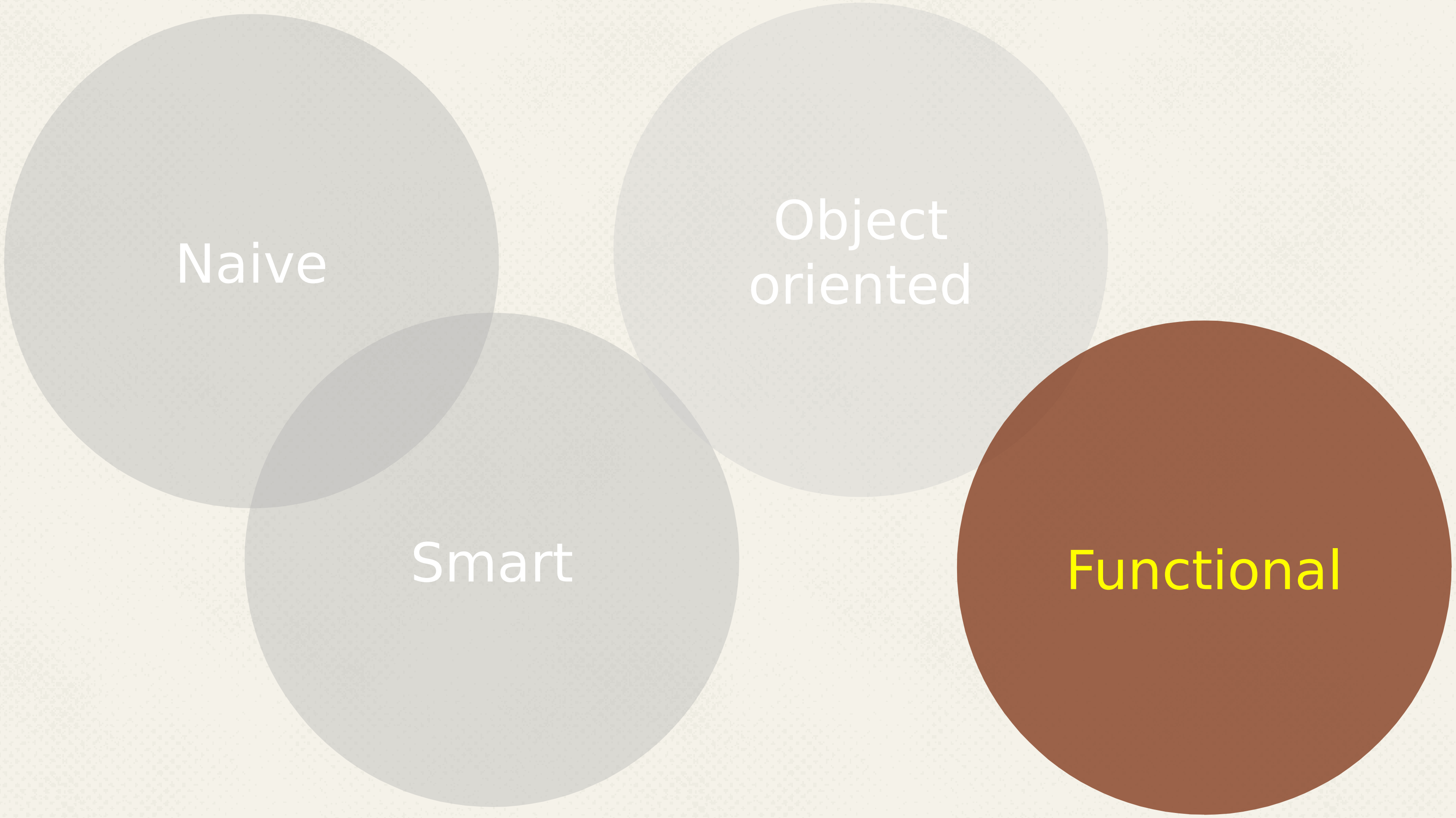
Object oriented implementation, some thoughts

This solution avoids writing and maintaining a lot of boilerplate and error prone code

You need, of course, dependency injection and the container in your project

The github repository contains a simple spring boot example

Factory pattern approaches



Functional

Look at all these beautiful functions

The factory pattern

Functional implementation, the idea

Since Java 8 you can design your factories also in a functional way

You can implement for each product type a function that takes the product key as parameter and returns the product

The main factory calls all the registered functions, passing the product key and returns the created product

The factory pattern

Functional implementation, the code

```
The collection of functions:
public class Factories {
  @Producer
  Function<String, Optional<Product>> FIRST_FACTORY =
            productKey -> (“first”).equals(productKey)
                    ? Optional.of(new FirstProduct())
                    : Optional.empty();
  }
  //… for each product a function annotated as @Producer
}
```

https://github.com/mw1980/onfactorypattern/ , the “functional” package

The factory pattern

Functional implementation, the code

```
The main factory class:
public class Factory {
  Optional<Product> createProduct(String productKey) {
         return prodFactories().stream()
                .map(factory -> factory.apply(productKey))
                .filter(Optional::isPresent)
                .findFirst()
                .orElse(Optional.empty());
    }
  List<Function<String, Optional<Product>>> prodFactories{
     //returns all functions annotated with @Producer
}}
```

https://github.com/mw1980/onfactorypattern/ , the “functional” package

The factory pattern

Functional implementation, some thoughts

No container magic, advanced set up, or external framework is needed

You always have the overview on the producer functions and you can easily unit test them

The code to load all functions annotated with Producer uses reflection and is kind of ugly

THANKS!

Any questions?

You can find me at:
marian.wamsiedel@gmail.com

linkedin
https://www.linkedin.com/in/marian-wamsiedel-4a1b792b/

CREDITS

- http://www.blackwasp.co.uk/gofpatterns.aspx
- http://www.oodesign.com/factory-pattern.html
- https://github.com/mariofusco/from-gof-to-lambda

CREDITS

- Presentation template by SlidesCarnival
- Photographs by Unsplash
- Backgrounds by SubtlePatterns
- Photographs https://www.pexels.com/