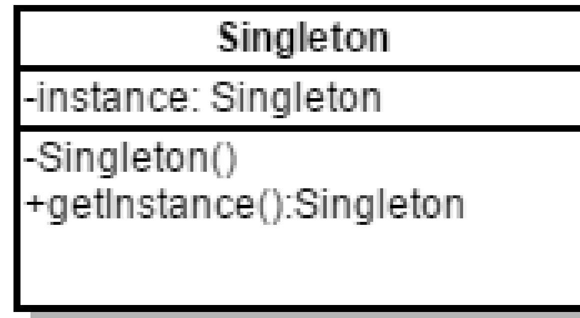
Design Patterns
Part 1

by Vlad Ungureanu

Agenda

Design Patterns

- Singleton
- Abstract Factory
- Builder
- Factory Method


Singleton
-instance: Singleton
-Singleton()
+getInstance():Singleton

Singleton – Implementation 1

```
public class Singleton implements Serializable{

	// thread safe static resource
	private static ThreadLocal<HeavyResource> heavyResource = new ThreadLocal<>();
	// double-checked locking pattern implementation
	public static HeavyResource getHeavyResource() {
		// lazy initialization
		if (heavyResource.get() == null) {
			System.out.println("Initialize resource!");
			synchronized (HeavyResource.class){
				if (heavyResource.get() == null) {
					heavyResource.set(new HeavyResource());
				}
			}
		}
		return heavyResource.get();
	}
}
```

Singleton – Implementation 2

```
public enum SingletonEnum {
        Instance;

        private SingletonEnum() {
        }

        public void dostuff() {
                System.out.println("Some heavy operation!");
        }

      }
}

// in main
SingletonEnum.Instance.dostuff();
```

Singleton – Why ?

- Global single instance of heavy resources
- Lazy loading of heavy resources
- Service locators are usually Singletons
- The Abstract Factory and Builder patterns can use Singletons in their implementation.
- Facade objects are often singletons because only one Facade object is required.
- State objects are often singletons.
- Singletons are often preferred to global variables because:
    - They do not pollute the global namespace (or, in languages with namespaces, their containing namespace) with unnecessary variables.
    - They permit lazy allocation and initialization, whereas global variables in many languages will always consume resources.

Singleton – Pitfalls: Serializable Duplicates

```
public static void obtainTwoSingletonInstances(){
    StreamOperations.writeObject(new Singleton());

    Singleton s1 = StreamOperations.readObject();
    Singleton s2 = StreamOperations.readObject();

    System.out.println(s1 + " and " + s2 + " are different instances");
}
```

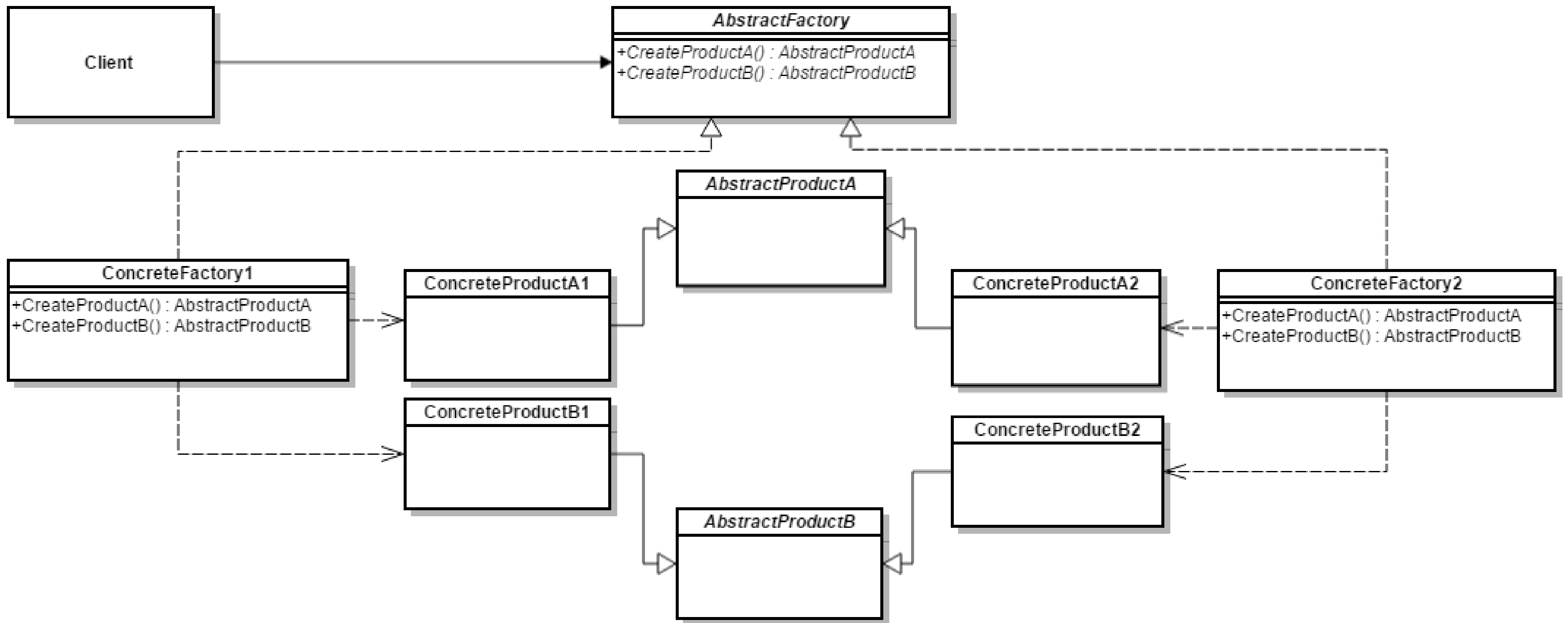
Singleton – Pitfalls: Continuous Session

```
public static Connection getConnection() {
        if (connection.get() == null) {
                Properties props = getPorperties();
                try {
                        Class.forName(props.getProperty("Driver"));
                        // open this connection
                        // !we never close this connection
                        connection.set(DriverManager.getConnection(props.getProperty("Url"),
                                        props.getProperty("User"),
props.getProperty("Password")));
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
        return connection.get();
    }
```

LEARN stuff IO

Singleton – Why not ?

- Can be easily abused - to many Singleton classes
- Cannot use any layer of abstraction - cannot use DI or substitution methods
- “I know where you live” anti-pattern - changes to Singleton have ripple effect
- “Continuous Session” anti-pattern - heavy resources might never get released
- Singletons can be designed for single user in multiuser systems


Client
AbstractFactory
+CreateProductA() : AbstractProductA
+CreateProductB() : AbstractProductB
AbstractProductA
ConcreteFactory1
+CreateProductA() : AbstractProductA
+CreateProductB() : AbstractProductB
ConcreteProductA1
ConcreteProductA2
ConcreteFactory2
+CreateProductA() : AbstractProductA
+CreateProductB() : AbstractProductB
ConcreteProductB1
ConcreteProductB2
AbstractProductB

Abstract Factory - Implementation

```
public interface AbstractFactory {

	SimpleWorker  getSimpleWorker();

	ComplexWorker  getCompleWorker();
}
```

Abstract Factory - Implementation

```
public class FinancialFactory implements AbstractFactory {
	@Override
	public SimpleWorker getSimpleWorker() {
		return new FinancialSimpleWorker();
	}
	@Override
	public ComplexWorker getCompleWorker() {
		return new FinancialComplexWorker();
	}
}
public class MedicalFactory implements AbstractFactory {
	@Override
	public SimpleWorker getSimpleWorker() {
		return new MedicalSimpleWorker();
	}
	@Override
	public ComplexWorker getCompleWorker() {
		return new MedicalComplexWorker();
	}
}
```

Abstract Factory - Implementation

```
private static AbstractFactory getDomainSpecificFactory() {
        if ("Pitfalls".equals(DOMAIN)) {
            return new SuperMedicalFactory();
        } else if ("Medical".equals(DOMAIN)) {
            return new MedicalFactory();
        } else {
            return new FinancialFactory();
        }
    }
}
```

LEARN stuff IO

Abstract Factory – Why ?

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

- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".

- We don’t want to be responsible for creating new objects (we don’t use the new operator)

- The creation of a library of products is needed, for which is relevant only the interface, while the implantation is left for the person using the library.

- When object creation depends on provided parameters or contextual parameters.

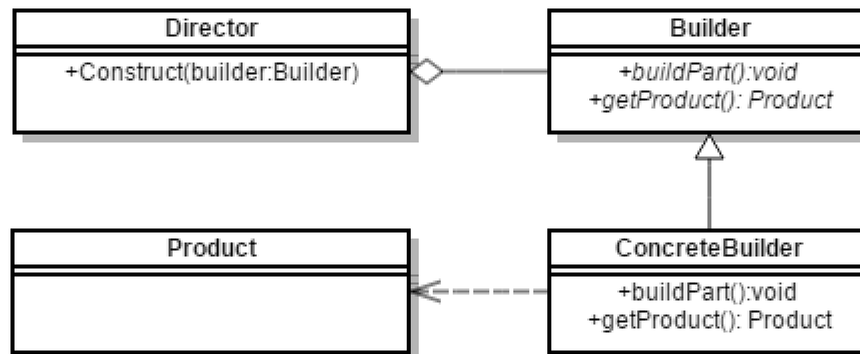
Abstract Factory – Pitfall: Abstraction Layer

```
public class SuperMedicalFactory implements AbstractFactory{

	@Override
	public SimpleWorker getSimpleWorker() {
		complexOperation();
		timeConsumingOperation();
		return new MedicalSimpleWorker();
	}

	@Override
	public ComplexWorker getCompleWorker() {
		cpuHeavyOperation();
		blockingOperation();
		return new MedicalComplexWorker();
	}
}
```

LEARN stuff IO

Abstract Factory – Why not?

- It is difficult to add new products to the factory
- Creation of objects might hide complex operations.
- Abstract factory implementation adds extra work especially in the first stages of development
- Extra layer of abstraction
- Abstract Factory is with concurrent with Prototype, Factory Method and Builder and might not be the best solution for the job.
- Most of the time Abstract Factories hide the previously mentioned patterns their implementation


Director
+Construct(builder:Builder)
Builder
+buildPart():void
+getProduct(): Product
Product
ConcreteBuilder
+buildPart():void
+getProduct(): Product

Builder – Implementation 1

```
public class FluentBuilder {
	protected ComplexClass  complexClass;

	private String authorization;
	private String credentials;
	private String roles;
	private String properties;
	// setter and getter
	public ComplexClass  build() {
		complexClass  = new ComplexClass();
		complexClass.setAuthorization(this.authorization);
		complexClass.setCredentials(this.credentials);
		complexClass.setRoles(this.roles);
		complexClass.setProperties(this.properties);
		return complexClass;
	}

}
```

Builder – Implementation 2

```
public class Builder {
	private BuilderInterface builderInterface;

	public Builder(BuilderInterface builderInterface){
		this.builderInterface = builderInterface;
	}

	public ComplexClass getComplexClass() {
		return this.builderInterface.getComplexClass();
	}
	public void build(String authorization, String credentaials, String roles, String properties) {
		this.builderInterface.buildAuthorization(authorization);
		this.builderInterface.buildCredentials(credentaials);
		this.builderInterface.buildRoles(roles);
		this.builderInterface.buildProperties(properties);
	}
}
```

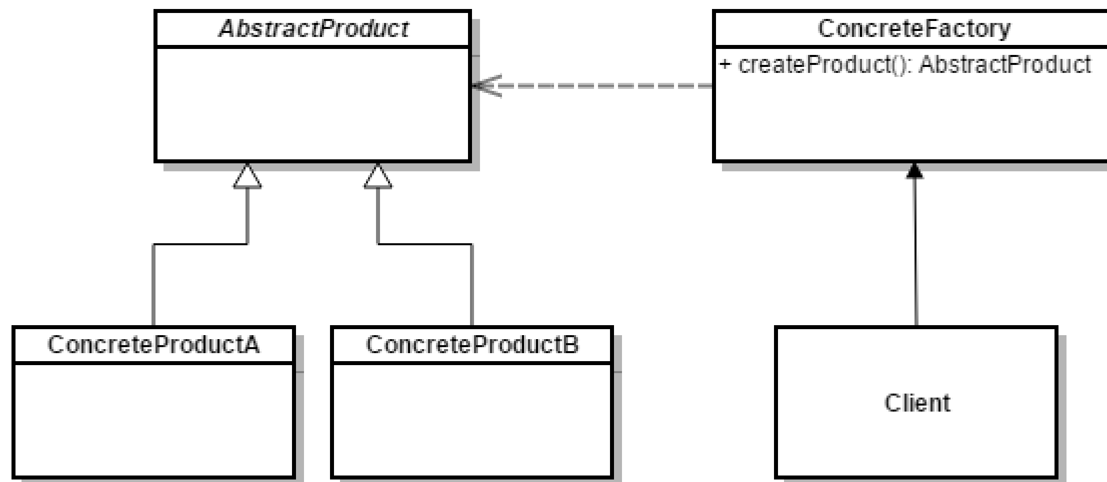
Builder – Why ?

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.
- A constructor could receive to many parameters
- The creation algorithm of a complex object is independent from the parts that actually compose the object
- The system needs to allow different representations for the objects that are being built
- Concurrent implementation with Abstract Factory with the particularity that it creates different types of products, not products of the same family

LEARN stuff IO

Builder – Why not?

- Builder might hide a composite object, or a very expensive one
- Extra layer of abstraction
- The number of builders is directly proportional to the number of classes we want to build, so if we have 100 DTO we will have 100 builders
- Some implementations of builders are great examples of Feature Envy or God Class code smells
- It encourages copy paste programming if used to obtain test stubs in unit testing or it increase unit testing code base


AbstractProduct
ConcreteFactory
+ createProduct(): AbstractProduct
ConcreteProductA
ConcreteProductB
Client

Factory Method – Implementation 1

```
public interface LogicalBuilder {
	ComplexClass buildComplexClass();
}
public class SimpleLogicBuilder implements LogicalBuilder{
	@Override
	public ComplexClass buildComplexClass() {
		return ComplexFactoryMethod.buildComplexClass("Simple One", "Simple Two", "Simple Three");
	}
}
public class ComplexLogicBuilder implements LogicalBuilder{
	@Override
	public ComplexClass buildComplexClass() {
		return ComplexFactoryMethod.buildComplexClass("Complex One", "Complex Two", "Complex Three");
	}
}
```

Factory Method – Implementation 1

```
public class LogicalFactoryMethod {
    public static ComplexClass buildClass(String option) {
        LogicalBuilder logicalBuilder = null;
        switch (option) {
        case "Simple":
            logicalBuilder = new SimpleLogicBuilder();
            break;
        case "Complex":
            logicalBuilder = new ComplexLogicBuilder();
            break;
        }
        return logicalBuilder.buildComplexClass();
    }
}
```

Factory Method – Implementation 2

```
public class ComplexFactoryMethod {

	public static String buildPartThroughComplexProcess(String content){
		return content;
	}

	public static ComplexClass buildComplexClass(String one, String two, String three){
		ComplexClass complexClass = new ComplexClass();
		complexClass.setComplexPartOne(buildPartThroughComplexProcess(one));
		complexClass.setComplexPartTwo(buildPartThroughComplexProcess(two));
		complexClass.setComplexPartThree(buildPartThroughComplexProcess(three));
		return complexClass;
	}

}
```

Factory Method – Why ?

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Defining a "virtual" constructor.
- You depend on an external resource but you don't know exactly which one yet.
- Construction of an object is expensive and you want to build once and reuse many times.

LEARN stuff IO

Factory Method – Why not?

- May provides a more complicated way of building something
- Extra level of abstraction
- May create indirect dependencies depending on the criteria it uses to build objects
- Sometimes a simple contractor would do the same thing
- If there are errors in the building process they may get masked and not get treated accordingly

THANK YOU!

Vlad Costel Ungureanu
ungureanu_vlad_costel@yahoo.com

This is a free course from LearnStuff.io
– not for commercial use –