



The Creational Problem

The Creational Series 1

Serge Beaumont, May 2006

Polymorphism is the main mechanism to create maintainable object-oriented code. Client code uses other code through an abstract interface, and as long as we don't break the behavioral contract, we can change the implementation behind that interface at will.

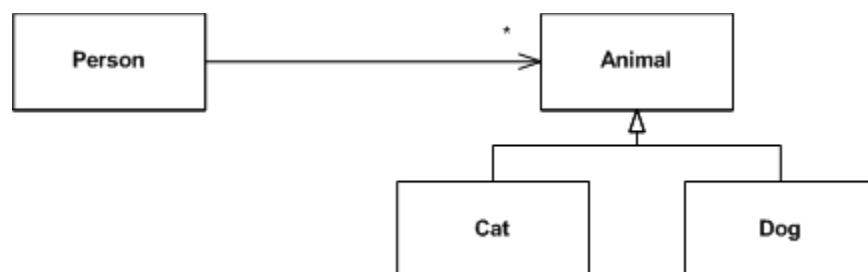
You will only achieve real maintainability if client code has dependencies to other code through an abstract interface *and nothing else*. Unfortunately **creational logic will - by its very nature - break this rule**. When you create an instance, at some point you will need call the constructor on the actual implementing class. This creates a dependency that prevents you from changing the implementation transparently, effectively cancelling the advantages of polymorphism.

The solution to the creational problem is to **protect the abstraction by hiding the creational logic** from the client code. Dependency injection is the ideal solution, in this case the client does not use creational logic at all. But in its turn the injecting code needs to create real instances.

This article is the first of a series of three: this article will illustrate how the creational problem manifests itself, the second will deal with some advanced implementations of creational code, and the third will deal with dependency injection.

Programming against abstractions

I will illustrate the creational problem with an an example. We have a - very original - class hierarchy with a base class `Animal` and subclasses `Cat` and `Dog`. Our client class `Person` is only interested in the general characteristics of an animal, like its name.





When we use *existing* animal instances we can list all the names by getting a collection from somewhere and using it:

```
public class Person {

    public List<Animal> getAnimals() {
        ...code that returns a list of animals from somewhere...
    }

    public void listAnimalNames() {
        for (Animal animal : getAnimals()) {
            System.out.println(animal.getName() + " says: " +
animal.speak());
        }
    }
}
```

This code does not know anything about the `Animal` class hierarchy. It just uses the `Animal` abstraction, and we can add or remove any subclass without breaking this code. In the example code we can add an arbitrary number of different behaviors behind the (polymorphic) calls to `animal.getName()` and `animal.speak()` by adding a `Animal` subclasses *without ever touching the calling code*.

It is crucial that we keep this desirable property intact. The fundamental trick to creating well-maintainable object-oriented code is by pushing the variable part behind an interface, which allows us to change the implementations, and polymorphism takes care of the rest.

So far we have maintainable code, but **we get into trouble when we need to create new instances**. We have to call the constructor of the actual types we are instantiating. The only way to create a dog is to call `new Dog()`, and a cat `new Cat()`. If we make `Person` responsible for creating the instances, we'll have calls to the constructors in the `Person` class itself. We will also see extra import statements showing up in the code:

```
package com.xebia.factoryarticle;

import com.xebia.factoryarticle.animals.Animal;
import com.xebia.factoryarticle.animals.Dog;
import com.xebia.factoryarticle.animals.Cat;

public class Person {
    private List<Animal> animals = new ArrayList<Animal>();

    public void listAnimalNames() {
        for (Animal animal : getAnimals()) {
            System.out.println(animal.getName());
        }
    }
}
```



```
public void addCat(String name) {
    animals.add(new Cat(name));
}

public void addDog(String name) {
    animals.add(new Dog(name));
}
}
```

The `Person` class now has a **dependency to every subclass of `Animal`**, just for creating instances. The whole point of programming to an abstraction has just been rendered useless, since every change to the `Animal` hierarchy is likely to break the `Person` class. You could just as well use the subclasses directly if you wanted: you're dependent on them anyway...

The creational problem negates the advantages of polymorphism.

Solving the creational problem – dependency injection

One way of solving the creational problem is to burden somebody else with it:

```
public class Person {
    private List<Animal> animals = new ArrayList<Animal>();

    public void addAnimal(Animal animal) {
        animals.add(animal);
    }
}
```

This is dependency injection (DI): something else creates the correct instances and gives (injects) them to the client object, in this case through the `addAnimal()` method.

Dependency injection is a very elegant mechanism, but **in the end there is always a place in the system where instances are created**. For the purpose of this article we're not going to inject our dependencies, but create them ourselves.

Solving the creational problem – factories

Let's improve the first version of `Person` by centralising the creational code (DRY principle).

The choice for the necessary subclass has been abstracted with a simple `String` parameter that is used to determine the type. There are more advanced ways to select the correct subclass, but the point here



is that the clients of `Person` are **protected from dependency on the `Animal` subclasses**.

```
public interface Animal {
    public static final CAT = "cat";
    public static final DOG = "dog";

    ...etc.
}

public class Person {
    private List<Animal> animals = new ArrayList<Animal>();

    public void addAnimal(String type, String name) {
        if (Animals.CAT.equals(type)) {
            animals.add(new Cat(name));
        } else if (Animals.DOG.equals(type)) {
            animals.add(new Dog(name));
        }
    }
}
```

Centralising the creational code is a first step, but the `Person` class still has dependencies to all the subclasses. This is where a factory comes in. We factor out the creational code into an `AnimalFactory` by moving the creational code in the `Person.addAnimal()` method.

```
public class Person {
    private AnimalFactory factory = new AnimalFactory();
    private List<Animal> animals = new ArrayList<Animal>();

    public void addAnimal(String type, String name) {
        animals.add(factory.create(type, name));
    }
}

public class AnimalFactory {
    public Animal create(String type, String name) {
        Animal result = null;

        if (Animals.CAT.equals(type)) {
            result = new Cat(name);
        } else if (Animals.DOG.equals(type)) {
            result = new Dog(name);
        }

        return result;
    }
}
```



You might be thinking: "Hey, all you did was move the problem. The `if..else if...` will still break with every change to the subclass hierarchy!". Yes, that's right, but there is a valid reason for the move.

`Person` is a client class, and every other client class that needs to instantiate an `Animal` subclass would need to implement a method like the `create()` method. Moving the code to a separate class means that all client classes can use the same creational code, which conforms to the DRY (Don't Repeat Yourself) principle.

The crucial point is that the system **will only break in one place** when you add a subclass. You can write code that enables you to configure the factory in a configuration file, or do some reflection wizardry, but you'll need to change *something* to tell your system that there's a new subclass. A humble `if...else if...` can be a perfectly good solution, as long as it's the only one of its kind. For the purposes of this example it suffices: we'll see some advanced techniques in the next article.

Factories protect abstractions

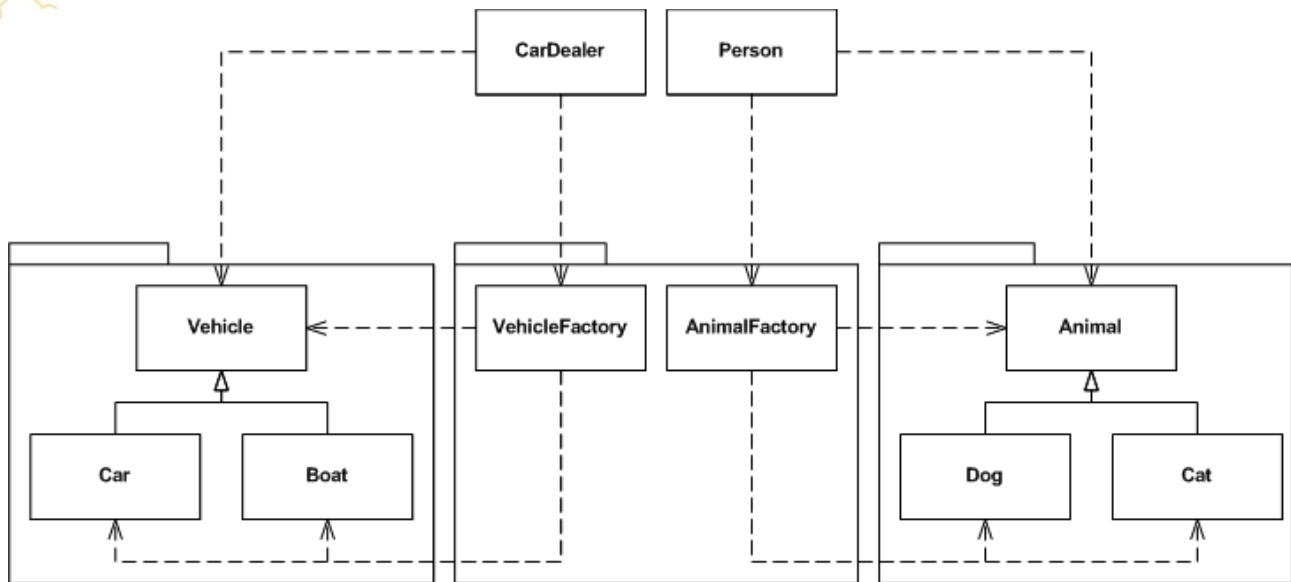
So here we have the purpose of a factory: they are not just for creating stuff, they **protect an abstraction**. Factories protect client classes from dependencies to the whole class hierarchy just because they need to create instances.

Clients of a factory should not start downcasting to subclass types. The point of the factory is to protect clients from knowledge of subclass types, and downcasting means that the reason for a factory's existence was nullified. In such a case, either fix the abstraction so clients can do everything they need without downcasting, or don't use a factory.

A Factory belongs with the abstraction it creates

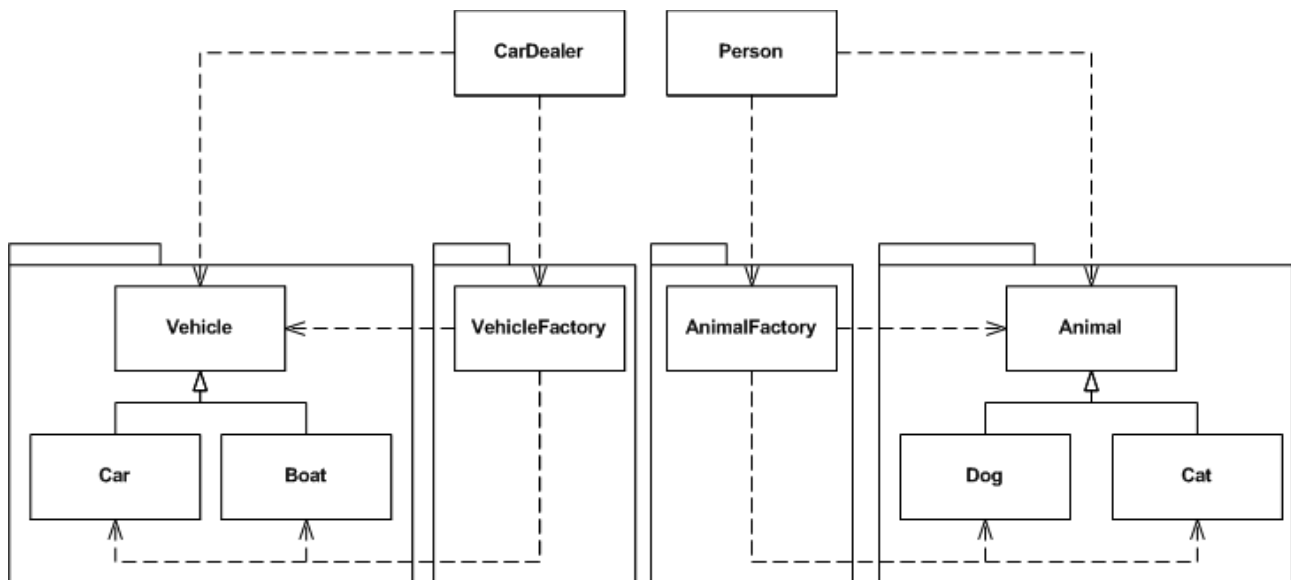
When you have a factory, where should it go? With the clients that use it, with the class hierarchy it creates, or separate from both?

The place where each factory belongs is with its abstraction: I'll explain why with **a counter-example**. Let's say you have a bunch of factories, and you place those factories together in a package. If you can have a `stringutils` package, why not a `creationalutils` package? Let's add a `Vehicle` class hierarchy with a `CarDealer` user, and see what this choice does to the package dependencies:



We've just created **dependency hell**. All clients depend on the factory package, and the factory package depends on all class hierarchies. Now the `Vehicle` and `Animal` halves of our example are connected, even though they do not have anything to do with each other.

This is better:



The vehicle and animal sides are completely separate again. The conclusion so far is that **unrelated factories should not be bundled together**.



In the figure above you can see that the `AnimalFactory` and the abstraction `Animal` are in separate packages. Note that a client of `AnimalFactory` will always have a dependency to the `Animal` abstraction, since that is what the factory returns.

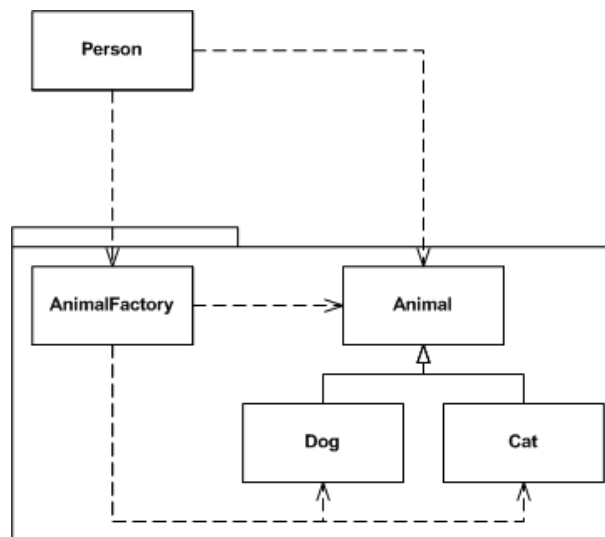
```
package com.xebia.factoryarticle;  
  
import com.xebia.factoryarticle.animals.Animal;  
  
public class AnimalFactory {  
    public Animal create(String type, String name) {  
        ... code ...  
    }  
}
```

The conclusion is that if a dependency to `AnimalFactory` automatically means a dependency to `Animal`, you might as well have `AnimalFactory` in the same package as `Animal`. Why introduce unnecessary cross-package dependencies?

If the dependency to one type automatically implies the dependency to another type, keep the types together.

In our example we do this by making sure that the dependency from `Person` to `AnimalFactory` points to the same package as the dependency from `Person` to `Animal`.

You now have this situation:



We could also state the previous principle as:



Creational dependencies should reuse behavioral dependencies, not introduce new ones.

You might have noticed a flaw in the last solution: the implementing classes are packaged with the interface (`AnimalFactory` and `Animal`) part: you can't swap implementation classes without opening up the package. In our current solution it wasn't really useful, since you need to change the `AnimalFactory` code for every change anyway, but in the next article we'll use more advanced techniques, and then the interface-implementation split will be useful.

Conclusion

This article discussed programming against abstractions, a very desirable property of good object-oriented code. We saw how the creational problem negates this, and how factories help to solve the creational problem.

You should now have a grasp of when to use factories: in the next article we'll discuss advanced ways of implementing factories.