## Creational Design Patterns

Five creational patterns have been documented by GoF:

**Abstract Factory** provides an interface for creating families of related objects, without specifying concrete classes.

**Factory Method** defines an interface for creating objects, but lets subclasses decide which classes to instantiate.

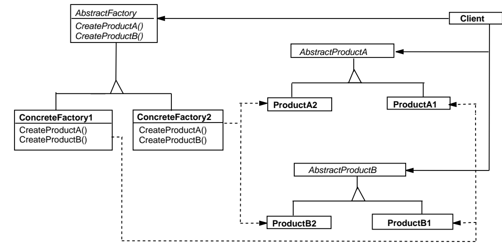
**Builder** separates the construction of a complex object from its representation, so that the same construction process can create different representation.

**Prototype** specifies the kind of objects to create using a prototypical instance.

**Singleton** ensures that a class has only one instance, and provides a global point of access to that instance.



## *Abstract Factory* Review

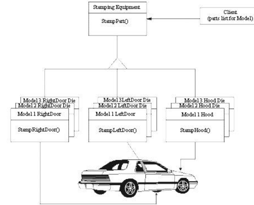



**When?**

- Need to abstract from details of implementation of products.
- Need to have multiple families of products.
- Need to enforce families of products that must be used together.
- Need to hide product implementations and just present interfaces.



## *Abstract Factory* Example 1



The stamping equipment is an Abstract Factory which creates car body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Concrete classes produced by the machinery can be changed through the use of rollers.

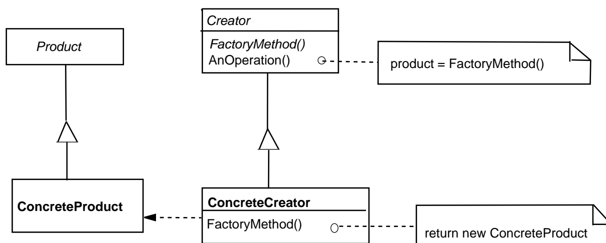


## *Abstract Factory* Example 1 (Continued ...)

- The Master Parts List corresponds to the client, which groups the parts into a family of parts.
- The Stamping Equipment corresponds to the Abstract Factory, as it is an interface for operations that create abstract product objects.
- The dies correspond to the Concrete Factory, as they create a concrete product.
- Each part category (Hood, Door, etc.) corresponds to the abstract product.
- Specific parts (i.e., driver side door for 1998 Sedan) corresponds to the concrete products.



## *Factory Method* Review

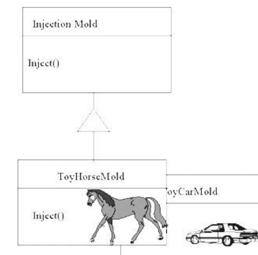



**When?**

- A class can't anticipate the class of the objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Need to delegate responsibility to helper subclasses.



## *Factory Method* Example 1




Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.

## *Factory Method* Example 1 (Continued ...)

- The Injection Mold corresponds to the Product, as it defines the interface of the objects created by the factory.
- A specific mold (ToyHorseMold or ToyCarMold) corresponds to the ConcreteProduct, as these implement the Product interface.
- The toy company corresponds to the Creator, since it may use the factory to create product objects.
- The division of the toy company that manufactures a specific type of toy (horse or car) corresponds to the ConcreteCreator.



## *Abstract Factory* versus *Factory Method*

Sometimes it is difficult to differentiate between abstract factory and factory method.

**Solution?**

- In the factory method, your class looks down (inheritance) for help in creating the object of specific type.
- In abstract factory, it looks to the left or right (association) for that help.

So use factory method if

1. your class depends on a subclass to decide what object to create.
2. inheritance plays a vital role.

However, if your class depends on another class, which may be abstract and you are associated with it, then use abstract factory.



## *Abstract Factory* Example 2

In this example, `AbstractPizzaFactory` defines the method names and return types to make various kinds of pizza.

There are two concrete factories, namely `PizzaPizzaConcreteFactory` and `PizzaHutConcreteFactory`, extending `AbstractPizzaFactory`.

An object can be defined as an `AbstractPizzaFactory` and instantiated as either a `PizzaPizzaConcreteFactory` (PPCF) or a `PizzaHutConcreteFactory` (PHCF). Both PPCF and PHCF have the `makeVeggiePizza` method, but return instances of different `VeggiePizza` subclasses: PPCF returns a `PizzaPizzaVeggiePizza` whereas PHCF returns a `PizzaHutVeggiePizza`.



## *Abstract Factory* Example 2 (Continued ...)

**The Abstract Factory**

```
public abstract class AbstractPizzaFactory {
    public abstract CheesePizza makeCheesePizza();
    public abstract VeggiePizza makeVeggiePizza();
    public abstract PepperoniPizza makePepperoniPizza();
}
```

**A Concrete Factory**

```
public class PizzaPizzaConcreteFactory extends AbstractPizzaFactory {
    public CheesePizza makeCheesePizza()
      {return new PizzaPizzaCheesePizza();}
    public VeggiePizza makeVeggiePizza()
      {return new PizzaPizzaVeggiePizza();}
    public PepperoniPizza makePepperoniPizza()
      {return new PizzaPizzaPepperoniPizza();}
}
```



## *Abstract Factory* Example 2 (Continued ...)

**An Abstract Product**

```
public abstract class VeggiePizza extends Pizza {
    public VeggiePizza() {
        pizzaName = "Veggie Pizza";
        toppings.add("Cheese");
    }
}
```

**A Concrete Product**

```
public class PizzaPizzaVeggiePizza extends VeggiePizza {
    public VeggiePizza() {
        pizzaName += "(Pizza Pizza)";
        toppings.add("Green Peppers");
        toppings.add("Sliced Tomatos");
        toppings.add("Mushrooms");
    }
}
```



## *Abstract Factory* Example 2 (Continued ...)

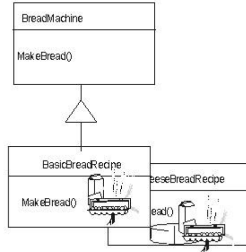
**Other**

In this particular example, the product hierarchy also has a root, namely `Pizza`.

```
public abstract class Pizza {
   String pizzaName;
   ArrayList toppings = new ArrayList();

   public String getPizzaName() {return pizzaName;}
}
```

## *Factory Method* Example 2




A Bread Machine allows its user to make bread. The recipe used determines the type of bread to be made.

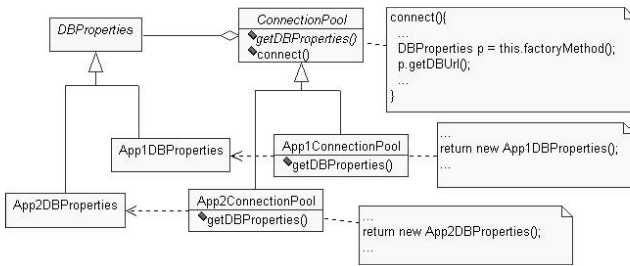


## *Factory Method* Example 2 (Continued ...)

- The Bread Machine corresponds to the Product, as it defines the interface of the objects created by the factory.
- A specific recipe (BasicBreadRecipe or CheeseBreadRecipe) corresponds to the ConcreteProduct, as these implement the Product interface.
- The user corresponds to the Creator, since he or she uses the factory to create product objects.



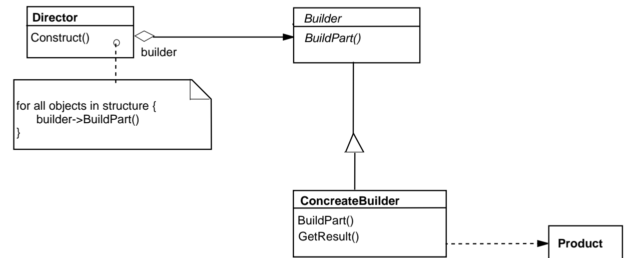
## *Factory Method* Example 3




In this example, ConnectionPool is the Creator and every DBProperties class acts like a Product. The ConnectionPool class istantiates a pool of logons and makes them available to a client.



## *Builder* Review

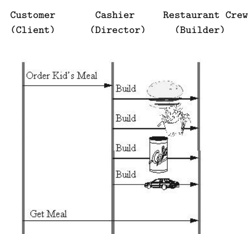



**When?**

- Need to isolate knowledge of the creation of a complex object from its parts.
- Need to allow different implementations/interfaces of an object's parts.



## *Builder* Example



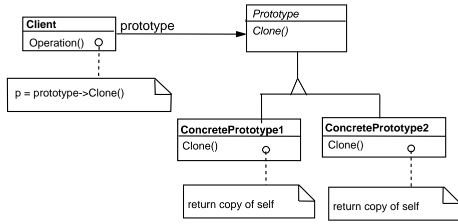

This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, coke, and toy car). Note that there can be variation in the contents of the children's meal, but the construction process is the same.



## *Builder* Example (Continued ...)

- The Kid's Meal concept corresponds to the builder, which is an abstract interface for creating parts of the Product object.
- The restaurant crew corresponds to the ConcreteBuilder, as they will assemble the parts of the meal (i.e. make a hamburger).
- The cashier corresponds to the Director, as he or she will specify the parts needed for the Kid's Meal, resulting in a complete Kid's meal.
- The Kid's Meal package corresponds to the Product, as it is a complex object created via the Builder interface.

## *Prototype* Review

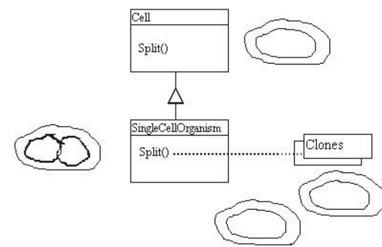



**When?**

- Need to be independent of how its products are created, composed, and represented.
- Need to avoid building a class hierarchy of factories that parallels the class hierarchy of products.
- The classes to instantiate are specified at run-time



## *Prototype* Example 1




The mitotic division of a cell, resulting in two identical cells, is an example of a prototype that plays an active role in copying itself and thus, demonstrates the Prototype pattern. When a cell splits, two cells of identical genotype result, i.e. the cell clones itself.



## *Prototype* Example 1 (Continued ...)

- The cell corresponds to the Prototype, as it has an “interface” for cloning itself.
- A specific instance of a cell corresponds to the ConcretePrototype.
- The DNA or genetic blue print corresponds to the Client, as it creates a new cell by instructing a cell to divide and clone itself.



## *Prototype* Example 2

In this example, we make new spoons and forks by cloning the objects which have been set as prototypes[a].

```
public class PrototypeFactory {
    AbstractSpoon prototypeSpoon;
    AbstractFork prototypeFork;

    public PrototypeFactory(AbstractSpoon spoon, AbstractFork fork) {
       prototypeSpoon = spoon;
       prototypeFork = fork;
    }

    public AbstractSpoon makeSpoon()
      {return (AbstractSpoon)prototypeSpoon.clone();}
    public AbstractFork makeFork()
      {return (AbstractFork)prototypeFork.clone();}
}
```

[a] Here, we only give the spoon class hierarchy. The corresponding classes for fork can be defined analogously.



## *Prototype* Example 2 (Continued ...)

```
public abstract class AbstractSpoon implements Cloneable {
   String spoonName;

   public void setSpoonName(String spoonName)
     {this.spoonName = spoonName;}

   public String getSpoonName() {return this.spoonName;}

   public Object clone() {
       Object object = null;
       try {
           object = super.clone();
       } catch (CloneNotSupportedException exception) {
           System.err.println("AbstractSpoon is not Cloneable");
       }
       return object;
   }
}
```



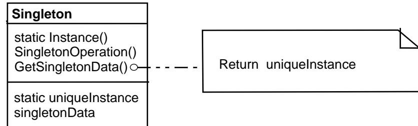
## *Prototype* Example 2 (Continued ...)

```
public class SoupSpoon extends AbstractSpoon {
   public SoupSpoon() {setSpoonName("Soup Spoon");}
}

public class SaladSpoon extends AbstractSpoon {
   public SaladSpoon() {setSpoonName("Salad Spoon");}
}

public class TestPrototype {
   public static void main(String[] args)
   {
       PrototypeFactory prototypeFactory =
          new PrototypeFactory(new SoupSpoon(), new SaladFork());
       AbstractSpoon spoon = prototypeFactory.makeSpoon();
       AbstractFork fork = prototypeFactory.makeFork();
   }
}
```

## *Singleton* Review

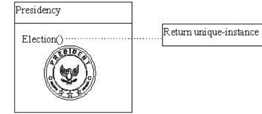



**When?**

- Need exactly one instance of a class and a well-known access point.
- Need to have the sole instance extensible by subclassing.



## *Singleton* Example




The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



## *Singleton* Example (Continued ...)

- The Office of the Presidency of the United States corresponds to the Singleton.
- The office has an instance operator (the title of "President of the United States") which provides access to the person in the office. At any time, at most one unique instance of the president exists.



## References

▷ *CS407's Textbook*

▷ *Non-Software Examples of Software Design Patterns*, Object Magazine, Vol. 7, No. 5, July 1997, pp. 52-57. Also available at: `www.agcs.com/supportv2/techpapers/patterns/papers/`

▷ *Design Patterns in Java: Reference and Example Site.* Available at `www.fluffycat.com/java/patterns.html`

▷ `www.cs.clemson.edu/~malloy/courses/patterns/patterns.html`

▷ `www.cs.uh.edu/~svenkat/ooad/`

▷ `www.ugolandini.net`