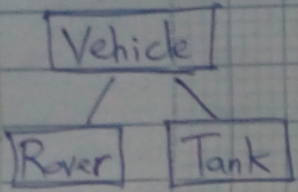
Factory



```
Factory
  makeVehicle(type)
    switch type
      case rover
        return new Rover
      case tank
        return new Tank
```

```
Vehicle rover = factory.makeVehicle('rover')
Vehicle tank = factory.makeVehicle('tank')
```

Prototype

```
PrototypeFactory
  getClone(Vehicle vehicle)
    return vehicle.makeCopy()
```

```
Vehicle
  makeCopy()
```

```
Rover
constructor
  expensive resource call
makeCopy
  return clone this
```

```
Vehicle rover1 = new Rover
Vehicle rover2 = prototypeFactory.getClone(rover1)
```

Builder

```
director = new Director(new RoverBuilder)
director.make()
[illegible]
Vehicle rover = director.get()
```

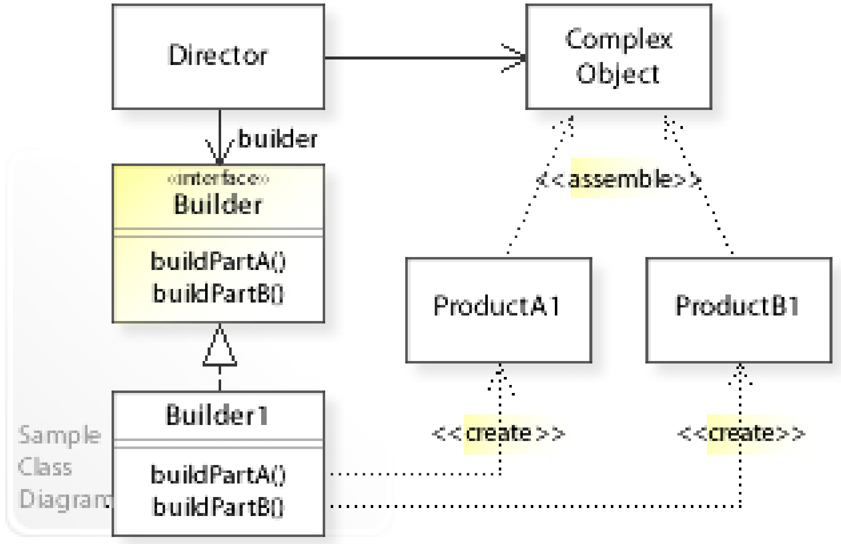
```
Director
  ref to roverBuilder

  make()
    roverBuilder.buildDoor
    roverBuilder.buildWheel

  get()
    return roverBuilder.getVehicle
```

```
VehicleBuilder
  buildDoor
  buildWheel
  getVehicle
```

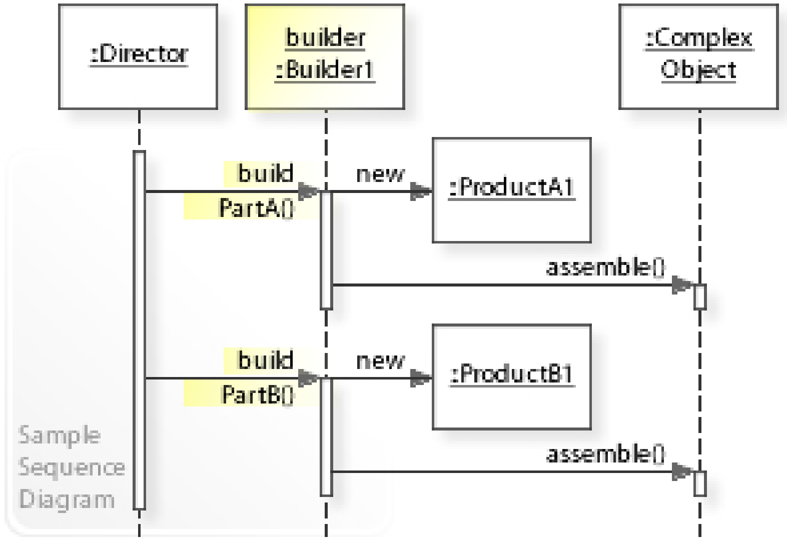
```
RoverBuilder
  ref. to rover
  buildDoor
    rover.setDoor
  buildWheel
    rover.setWheel
  getVehicle
    return rover
```

```
Vehicle
  setDoor(door)
  setWheel(wheel)
```

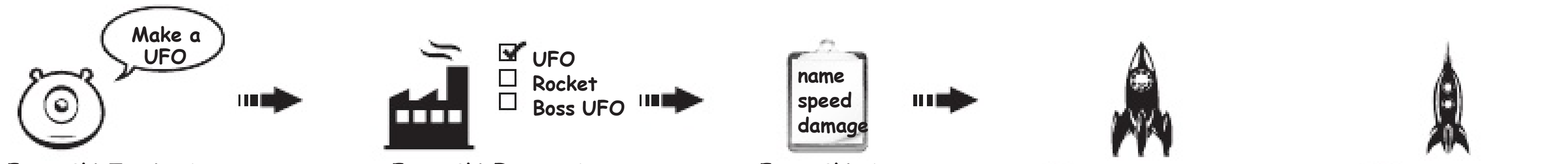
```
Rover
setDoor(door)
  this.door = door
setWheel(wheel)
  this.wheel = wheel
```



Director
Complex Object
builder
«interface»
Builder
buildPartA()
buildPartB()
<<assemble>>
ProductA1
ProductB1
Builder1
buildPartA()
buildPartB()
<<create>>
<<create>>
Sample Class Diagram



:Director
builder :Builder1
:Complex Object
build PartA()
new
:ProductA1
assemble()
build PartB()
new
:ProductB1
assemble()
Sample Sequence Diagram



EnemyShipTesting.java
<Store>

```
public static void main(String[] args){

        // Create the factory object
        EnemyShipFactory shipFactory =
            new EnemyShipFactory();

        // Enemy ship object

        EnemyShip theEnemy = null;

        Scanner userInput = new Scanner(System.in);

        System.out.print("What type of ship? (U / R / B)");

        if (userInput.hasNextLine()){

            String typeOfShip = userInput.nextLine();

            theEnemy = shipFactory.makeEnemyShip(typeOfShip);

            if(theEnemy != null){

                doStuffEnemy(theEnemy);

            } else System.out.print("Please enter U, R,
                or B next time");

        }
```

EnemyShipFactory.java
<Factory>

```
public EnemyShip makeEnemyShip(String newShipType)
{

        EnemyShip newShip = null;

        if (newShipType.equals("U")){

            return new UFOEnemyShip();

        } else

        if (newShipType.equals("R")){

            return new RocketEnemyShip();

        } else

        if (newShipType.equals("B")){

            return new BigUFOEnemyShip();

        } else return null;

    }
```

EnemyShip.java
<AbstractClass>

```
public abstract class EnemyShip
{

    private String name;
    private double speed;
    private double damage;

    public String getName()
        { return name; }
    public void setName(String newName)
        { name = newName; }

    public double getDamage()
        { return amtDamage; }
    public void setDamage(double newDamage)
        { amtDamage = newDamage; }
}
```

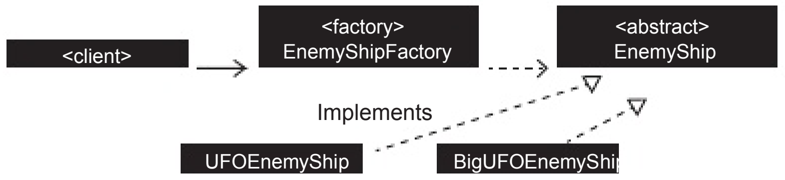
☐ **BigUFOEnemyShip**

```
public class BigUFOEnemyShip extends EnemyShip
{
    public BigUFOEnemyShip(){

        setName("Big UFO Enemy Ship");
        setDamage(40.0);
        setSpeed(10.0);

    }
}
```

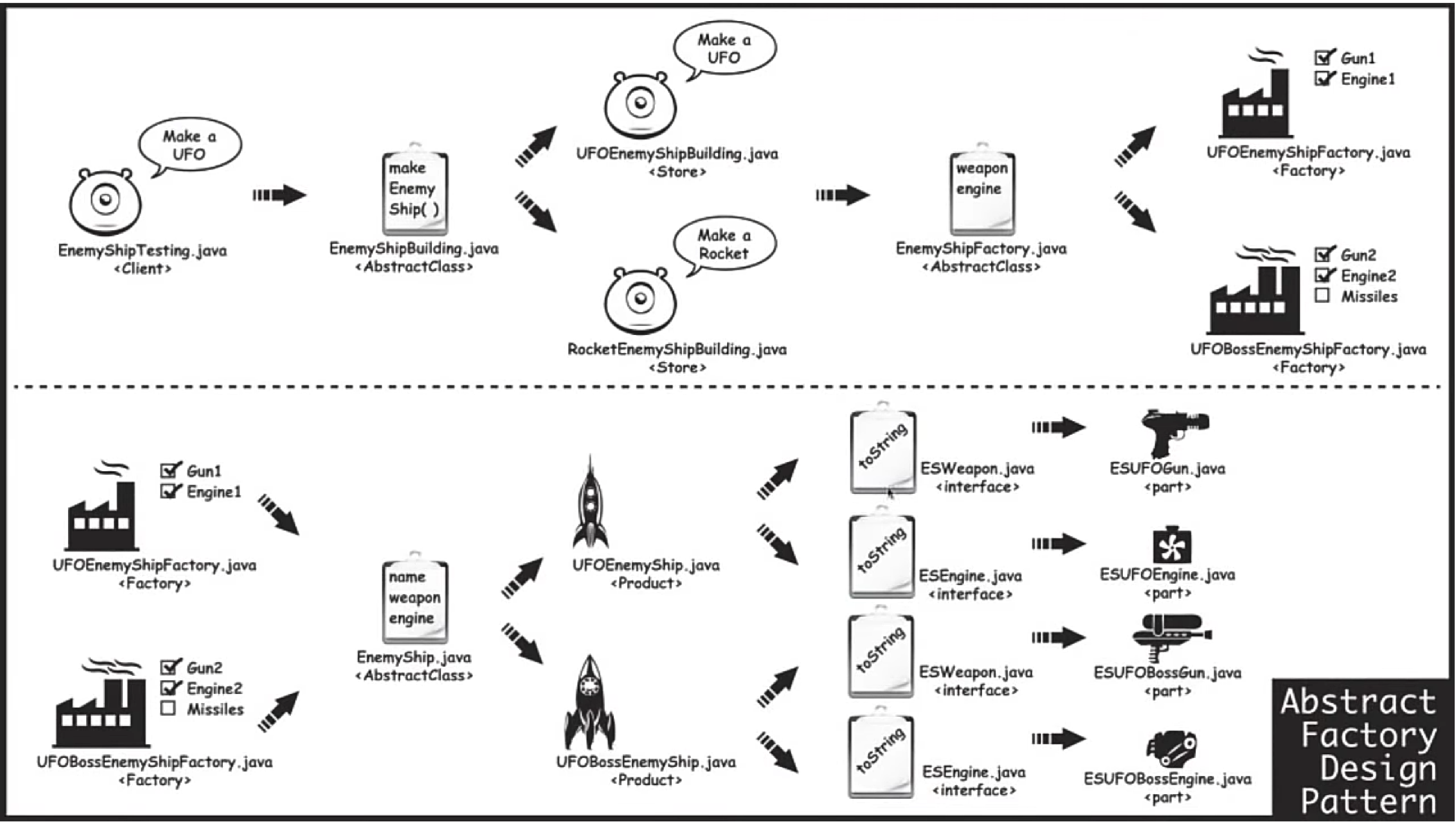
✓ **UFOEnemyShip**

```
public class UFOEnemyShip extends EnemyShip
{
    public UFOEnemyShip(){

        setName("UFO Enemy Ship");
        setDamage(20.0);
        setDamage(20.0);

    }
}
```

The **Factory Pattern** allows you to create objects without specifying the exact class of object that will be created.



Factory Design Pattern


Make a UFO
EnemyShipTesting.java <Client>
make Enemy Ship( )
EnemyShipBuilding.java <AbstractClass>
Make a UFO
UFOEnemyShipBuilding.java <Store>
Make a Rocket
RocketEnemyShipBuilding.java <Store>
weapon engine
EnemyShipFactory.java <AbstractClass>
☑ Gun1
☑ Engine1
UFOEnemyShipFactory.java <Factory>
☑ Gun2
☑ Engine2
☐ Missiles
UFOBossEnemyShipFactory.java <Factory>
☑ Gun1
☑ Engine1
UFOEnemyShipFactory.java <Factory>
☑ Gun2
☑ Engine2
☐ Missiles
UFOBossEnemyShipFactory.java <Factory>
name weapon engine
EnemyShip.java <AbstractClass>
UFOEnemyShip.java <Product>
UFOBossEnemyShip.java <Product>
toString
ESWeapon.java <interface>
toString
ESEngine.java <interface>
toString
ESWeapon.java <interface>
toString
ESEngine.java <interface>
ESUFOGun.java <part>
ESUFOEngine.java <part>
ESUFOBossGun.java <part>
ESUFOBossEngine.java <part>
Abstract Factory Design Pattern

Factory design pattern

Factory design pattern is one of the **Creational design pattern.** Factory design pattern is used when we have a super class with multiple sub-classes and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class.

Super class

Super class in factory design pattern can be an interface, abstract class or a normal java class. In our example, we have abstract super class with overridden **toString()** method for testing purpose.

```
package com.ashok.designpatterns.model;

public abstract class MyComputer {

   public abstract String getRAM();
   public abstract String getHDD();
   public abstract String getCPU();

   @Override
   public String toString(){
      return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();
   }
}
```

Sub class

Let's say we have two sub classes MyPersonalComputer and MyServer with below implementation.

```
package com.ashok.designpatterns.model;

public class MyPersonalComputer extends MyComputer {

   private String ram;
   private String hdd;
   private String cpu;

   public MyPersonalComputer(String ram, String hdd, String cpu){
      this.ram = ram;
      this.hdd = hdd;
      this.cpu = cpu;
   }
   @Override
   public String getRAM() {
      return this.ram;
   }

   @Override
   public String getHDD() {
      return this.hdd;
   }

   @Override
   public String getCPU() {
      return this.cpu;
   }
}
```

Notice that both the classes are extending MyComputer super class.

```
package com.ashok.designpatterns.model;

public class MyServer extends MyComputer {
   private String ram;
   private String hdd;
   private String cpu;

   public MyServer(String ram, String hdd, String cpu){
      this.ram=ram;
      this.hdd=hdd;
      this.cpu=cpu;
   }
   @Override
   public String getRAM() {
      return this.ram;
   }

   @Override
   public String getHDD() {
      return this.hdd;
   }

   @Override
   public String getCPU() {
      return this.cpu;
   }
}
```

Finally the factory class is

```
     package com.ashok.designpatterns.factory;

     import com.ashok.designpatterns.model.MyComputer;
     import com.ashok.designpatterns.model.MyPersonalComputer;
     import com.ashok.designpatterns.model.MyServer;

     public class MyComputerFactory {

        public static MyComputer getComputer(String type, String ram, String hdd, String cpu){
           if("MyPersonalComputer".equalsIgnoreCase(type))
              return new MyPersonalComputer(ram, hdd, cpu);
           else if("MyServer".equalsIgnoreCase(type))
              return new MyServer(ram, hdd, cpu);
           return null;
        }

     }
```

Note

1. We can keep Factory class Singleton or we can keep the method that returns the subclass as static.

2. Notice that based on the input parameter, different subclass is created and returned. **getComputer** is the factory method.

Now, simple test client program that uses above factory design pattern implementation is as follows,

```
     package com.ashok.designpatterns.test;

     import com.ashok.designpatterns.factory.PCFactory;
     import com.ashok.designpatterns.factory.MyComputerFactory;
     import com.ashok.designpatterns.model.MyComputer;

     public class MyTestFactory {
        public static void main(String[] args) {
           Computer pc = ComputerFactory.getComputer("PersonalComputer","4 GB","500 GB","2.6 GHz")
           Computer server = MyComputerFactory.getComputer("MyServer","16 GB","1 TB","2.9 GHz");
           System.out.println("Factory PC Configuration is : " +pc);
           System.out.println("Factory Server Configuration is : " +server);
        }
     }

     Output
     Factory PC Configuration is : RAM= 4 GB, HDD=500 GB, CPU=2.6 GHz
     Factory Server Configuration is : RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

Prototype Design Pattern

Prototype Design Pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. This design pattern provides a mechanism to **copy the original object to a new object and then modify it according to our needs.** Prototype design pattern uses java cloning to copy the object.

Example

Suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it's not a good idea to create the Object using new keyword and load all the data again from database. The better approach would be to clone the existing object into a new object and then do the data manipulation.

```
   package com.ashok.designpatterns.prototype;

   import java.util.ArrayList;
   import java.util.List;

   public class Employees implements Cloneable {

      private List<String> empList;

      public Employees() {
         empList = new ArrayList<String>();
      }

      public Employees(List<String> list) {
         this.empList = list;
      }

      public void loadData() {
         // read all employees from database and put into the list
         empList.add("Ashok");
         empList.add("Vinod");
         empList.add("Dillesh");
         empList.add("Latha");
      }

      public List<String> getEmpList() {
         return empList;
      }

      @Override
      public Object clone() throws CloneNotSupportedException {
         List<String> temp = new ArrayList<String>();
         for (String s : this.getEmpList()) {
            temp.add(s);
         }
         return new Employees(temp);
      }
   }
```

Now the prototype design pattern example test program that will show the benefit of prototype pattern.

```
package com.ashok.designpatterns.prototype;

import java.util.List;

import com.ashok.designpatterns.prototype.Employees;

public class MyPrototypePatternTest {

   public static void main(String[] args) throws CloneNotSupportedException {
      Employees emps = new Employees();
      emps.loadData();

      Employees emps1 = (Employees) emps.clone();
      Employees emps2 = (Employees) emps.clone();
      List<String> list = empsNew.getEmpList();
      list.add("Janaki");
      List<String> list1 = empsNew1.getEmpList();
      list1.remove("Dillesh");

      System.out.println("emps List  : " + emps.getEmpList());
      System.out.println("emps1 List : " + list);
      System.out.println("emps2 List : " + list1);
   }
}

Output
emps List  : [Ashok, Vinod, Dillesh, Latha]
emps1 List : [Ashok, Vinod, Dillesh, Latha, Janaki]
emps2 List : [Ashok, Vinod, Latha]
```

If the object cloning was not provided, we will have to make database call to fetch the employee list every time. Then do the manipulations that would have been resource and time consuming.