

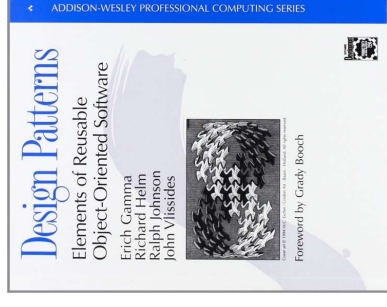
Design Patterns – Creational

UA.DETI.PDS

José Luis Oliveira

Resources

- ❖ Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.



- ❖ *Design Patterns Explained Simply*
(sourcemaking.com)

Creational patterns

Class

- ❖ Factory Method

Object

- ❖ Abstract Factory

- ❖ Builder

- ❖ Singleton

- ❖ Object Pool

- ❖ Prototype

Creational Patterns

- ❖ Problem: constructors in Java (and other OO languages) are inflexible
 - 1. Can't return a subtype of the type they belong to
 - 2. Always return a fresh new object, can't reuse
- ❖ “Factory” creational patterns present a solution to the first problem
 - Factory method, Factory object, Prototype
- ❖ “Sharing” creational patterns present a solution to the second problem
 - Singleton

Factory Method

Class

❖ Factory Method

Object

❖ Abstract Factory

❖ Builder

❖ Singleton

❖ Object Pool

❖ Prototype



Motivation

❖ Intent

- The new operator is considered harmful.
- Define a "virtual" constructor.
- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

❖ Problem

- A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

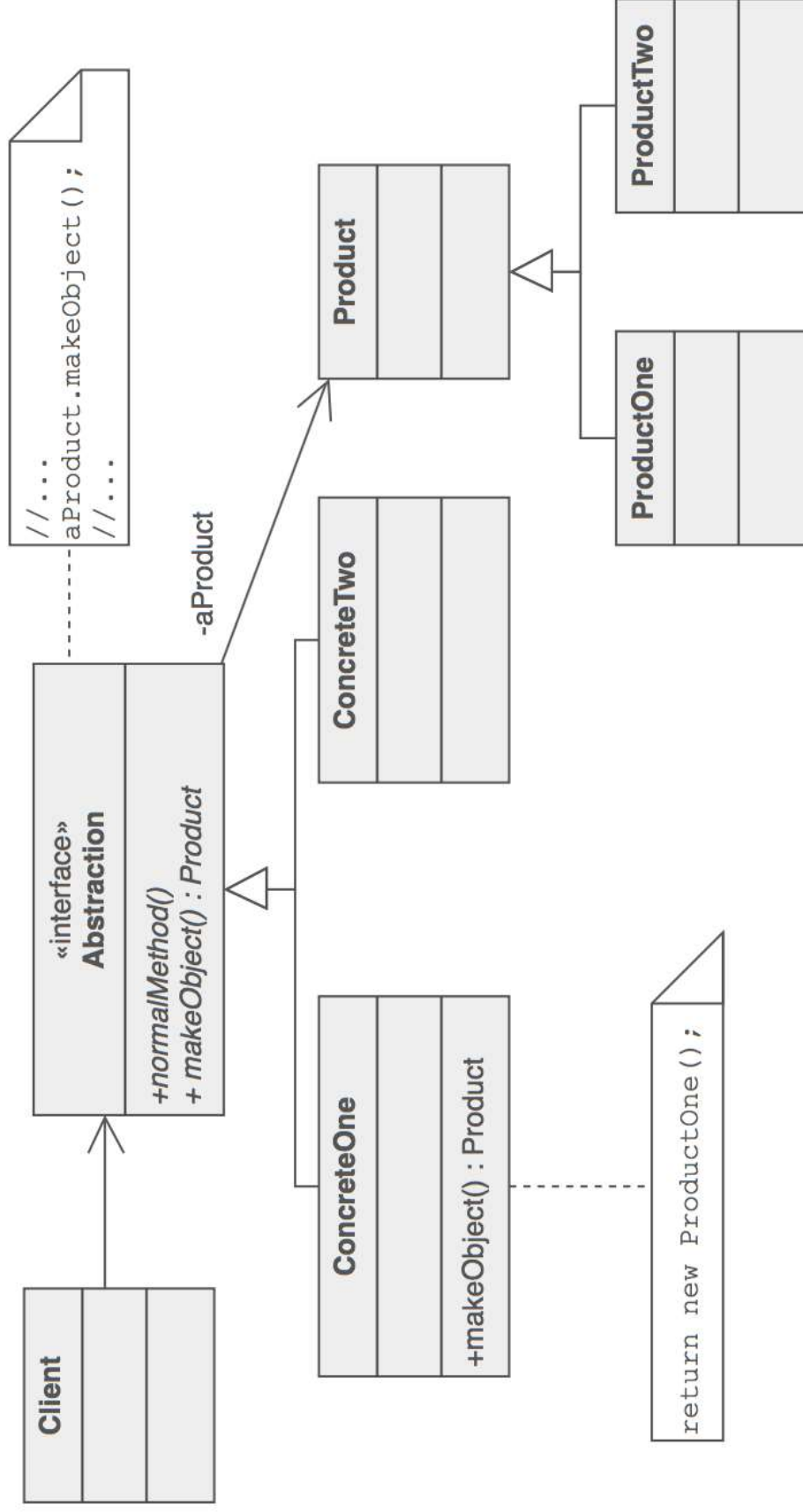
Solution (simple example)

```
public final class ComplexNumber {  
    // Static factory method returns an object of this class.  
    public static ComplexNumber valueOf(double aReal, double aImaginary) {  
        return new ComplexNumber(aReal, aImaginary);  
    }  
    // Caller cannot see this private constructor.  
    private ComplexNumber(double aReal, double aImaginary) {  
        fReal = aReal;  
        fImaginary = aImaginary;  
    }  
    //...  
    private double fReal;  
    private double fImaginary;  
}
```

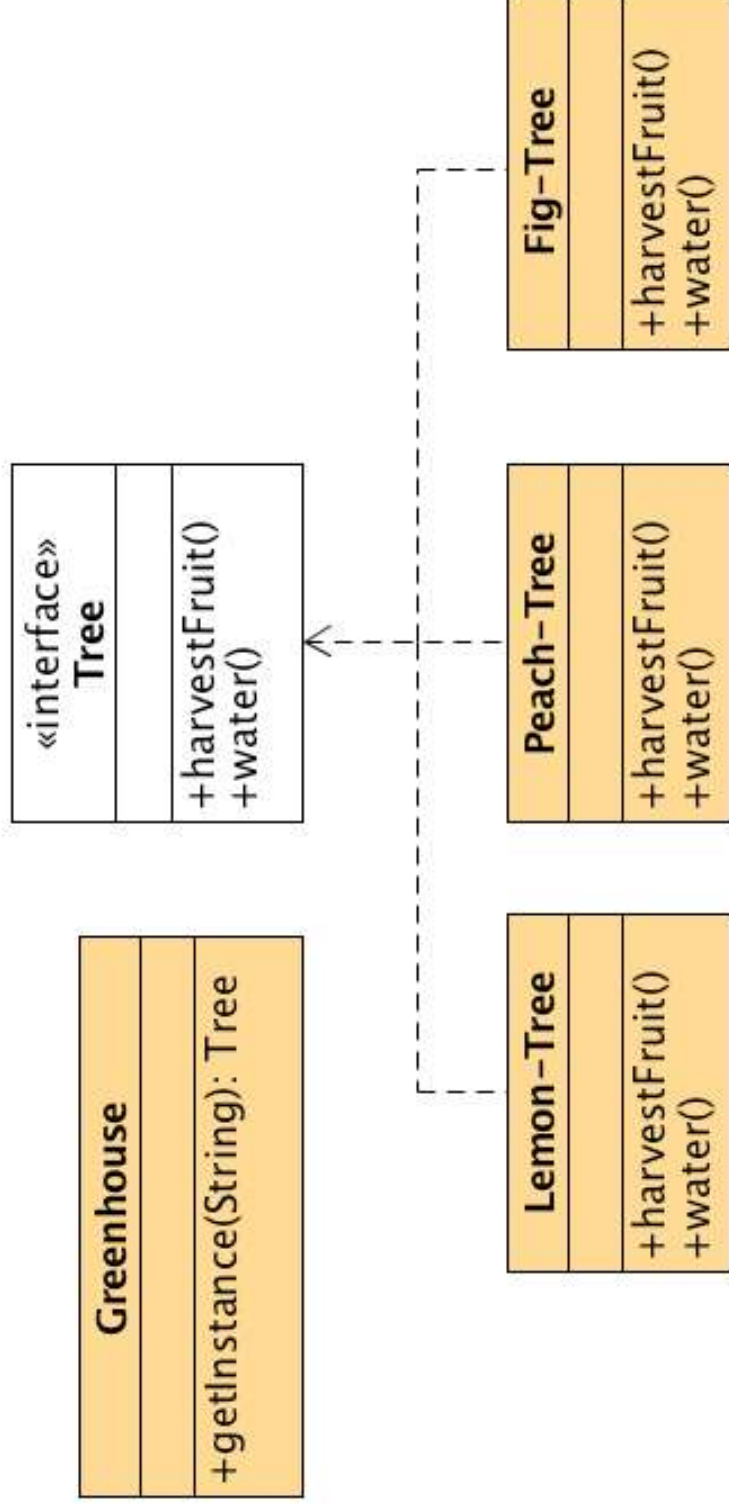
Solution

- ❖ An increasingly popular definition of factory method is **a static method of a class that returns an object of that class' type.**
 - But unlike a constructor, the actual object it returns might be an instance of a subclass.
 - Unlike a constructor, an existing object might be reused, instead of a new object created.
 - Unlike a constructor, factory methods can have different and more descriptive names
 - `Color.make_RGB_color(float red, float green, float blue)`
 - `Color.make_HSB_color(float hue, float saturation, float brightness)`

Structure



Example



Example

```
interface Arvore {
    void regar();
    void colherFruta();
}

class Figueira implements Arvore {
    protected Figueira() {System.out.println("Figueira plantada.");}
    public void regar() { System.out.println("Figueira: Regar muito pouco"); }
    public void colherFruta() { System.out.println("Hum.. figos!"); }
}

class Pessegueiro implements Arvore {
    protected Pessegueiro() {System.out.println("Pessegueiro plantado.");}
    public void regar() { System.out.println("Pessegueiro: Regar normal"); }
    public void colherFruta() { System.out.println("Boa.. pessegos!"); }
}

class Limoeiro implements Arvore {
    protected Limoeiro() {System.out.println("Limoeiro plantada.");}
    public void regar() { System.out.println("Limoeiro: Regar pouco"); }
    public void colherFruta() { System.out.println("Ahh.. Caipirinha!"); }
}
```

Example

```
class Viveiro {
    public static Arvore factory(String pedido){
        if (pedido.equalsIgnoreCase("Figueira"))
            { return new Figueira(); }
        if (pedido.equalsIgnoreCase("Pessegueiro"))
            { return new Pessegueiro(); }
        if (pedido.equalsIgnoreCase("Limoeiro"))
            { return new Limoeiro(); }
        else
            throw new IllegalArgumentException(pedido + " não existente!");
    }
    //...
}

// or with Java Reflection
public static Arvore factory2(String pedido) {
    Arvore arv = null;
    try {
        arv = (Arvore) Class.forName("Factory."+pedido).newInstance();
    } catch (Exception e) {
        throw new IllegalArgumentException(pedido + " nao existente!");
    }
    return arv;
}
```

Example

```
public static void main(String[] args) {  
    Arvore pomar[] = {  
        Viveiro.factory("Figueira"),  
        Viveiro.factory("Pessegueiro"),  
        Viveiro.factory("Limoeiro")  
    };  
    for (Arvore a: pomar)  
        a.regar();  
    for (Arvore a: pomar)  
        a.colherFruta();  
}
```

```
Figueira plantada.  
Pessegueiro plantado.  
Limoeiro plantada.  
Figueira: Regar muito pouco  
Pessegueiro: Regar normal  
Limoeiro: Regar pouco  
Hum.. figos!  
Boa.. pessegos!  
Ahh.. Caipirinha!
```

Another Example

```
class Race {  
    Race createRace() {  
        Bicycle bike1 = new Bicycle();  
        Bicycle bike2 = new Bicycle(); //...  
    }  
}  
class TourDeFrance extends Race {  
    Race createRace() {  
        Bicycle bike1 = new RoadBicycle();  
        Bicycle bike2 = new RoadBicycle(); //...  
    }  
}  
class Cyclocross extends Race {  
    Race createRace() {  
        Bicycle bike1 = new MountainBicycle();  
        Bicycle bike2 = new MountainBicycle(); //...  
    }  
}
```

Problem with this code:

Code duplication!

createRace is very similar among the 3 classes.

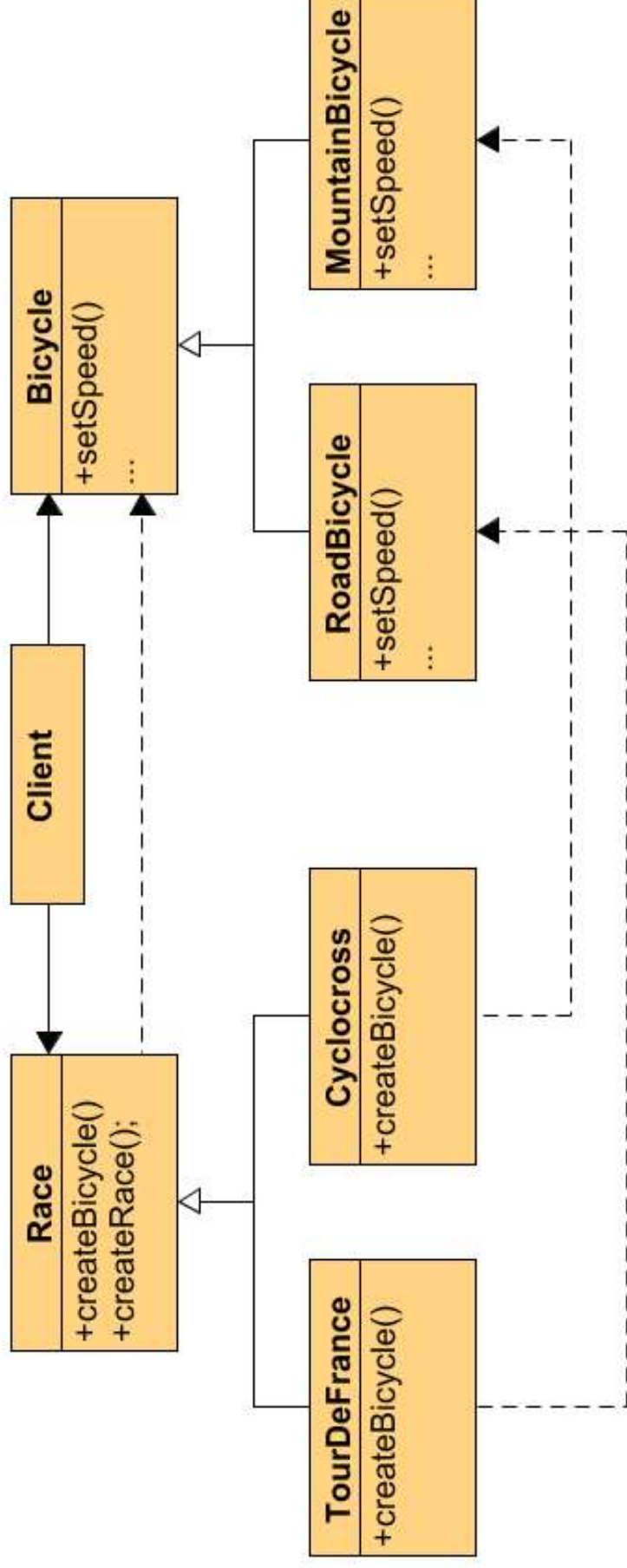
Why not have a single createRace in Race?

Using Factory Method

```
class Race {  
    Bicycle createBicycle() {  
        return new Bicycle();  
    }  
    Race createRace() {  
        Bicycle bike1 = createBicycle();  
        Bicycle bike2 = createBicycle(); //...  
    }  
}  
class TourDeFrance extends Race {  
    Bicycle createBicycle() {  
        return new RoadBicycle();  
    }  
}  
class Cyclocross extends Race {  
    Bicycle createBicycle() {  
        return new MountainBicycle();  
    }  
}
```

Parallel Hierarchies

- ❖ Can extend with new Races and Bikes with no modification (generally) to Client



Factory Methods in the JDK

- ❖ Calendar replaced Date (JDK1.0)
- ❖ DateFormat encapsulates knowledge on how to format a Date
 - Options: Just date? Just time? date+time?

```
Calendar td = Calendar.getInstance();  
Date today = td.getTime();
```

```
DateFormat df1 = DateFormat.getDateInstance();  
DateFormat df2 = DateFormat.getTimeInstance();  
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL);
```

```
System.out.println(df1.format(today)); // "9/jan/2015"  
System.out.println(df2.format(today)); // "10:01:24"  
System.out.println(df3.format(today)); // "Sexta-feira, 9 de Janeiro de 2015"
```

Check list

- ❖ If the constructor may lead to inconsistent objects, consider designing a factory method.
- ❖ Consider making all constructors private or protected.
- ❖ If you have an inheritance hierarchy that exercises polymorphism, consider adding a polymorphic creation capability by defining a static factory method in the base class.
- ❖ Consider designing an internal "object pool" that will allow objects to be reused instead of created from scratch.

Abstract Factory

Class

- ❖ Factory Method

Object

- ❖ Abstract Factory

- ❖ Builder

- ❖ Singleton

- ❖ Object Pool

- ❖ Prototype



Motivation

❖ Problem

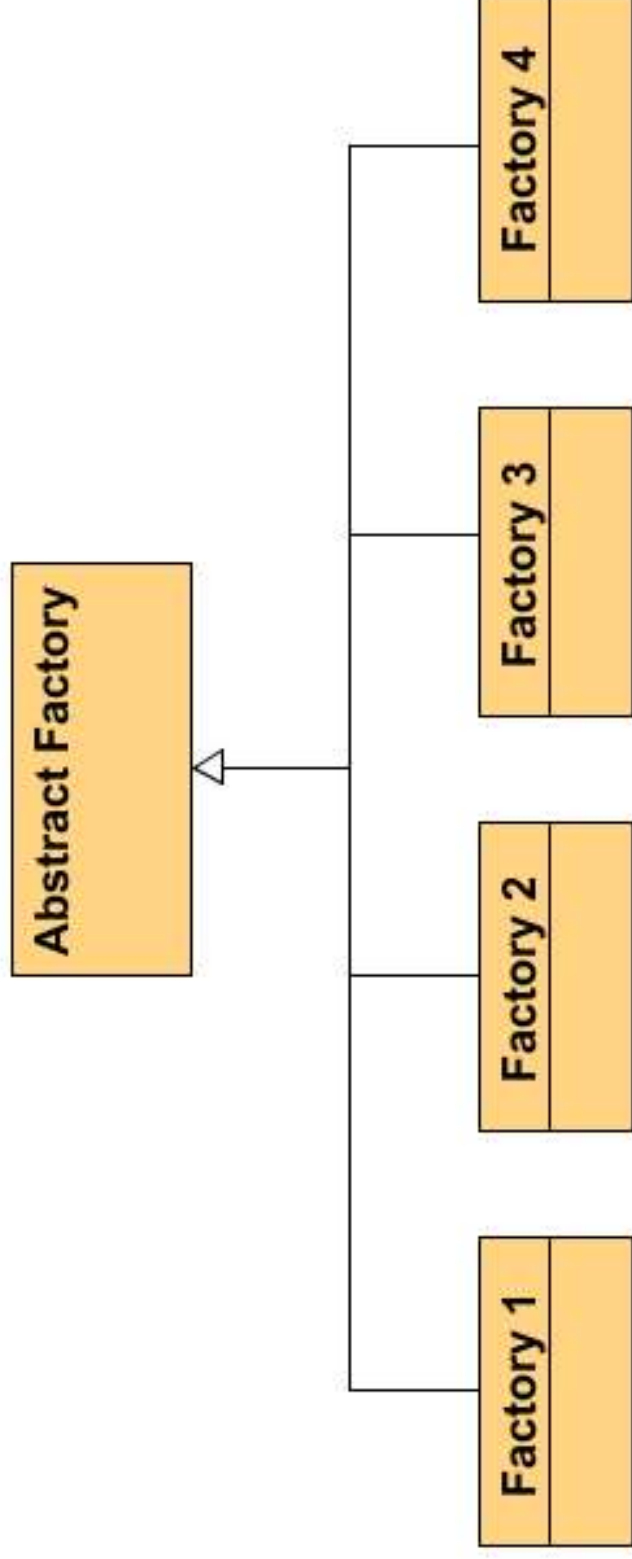
- If an application is to be portable, it needs to encapsulate platform dependencies.
- These "platforms" might include a windowing system, operating system, database, etc.

❖ Intent

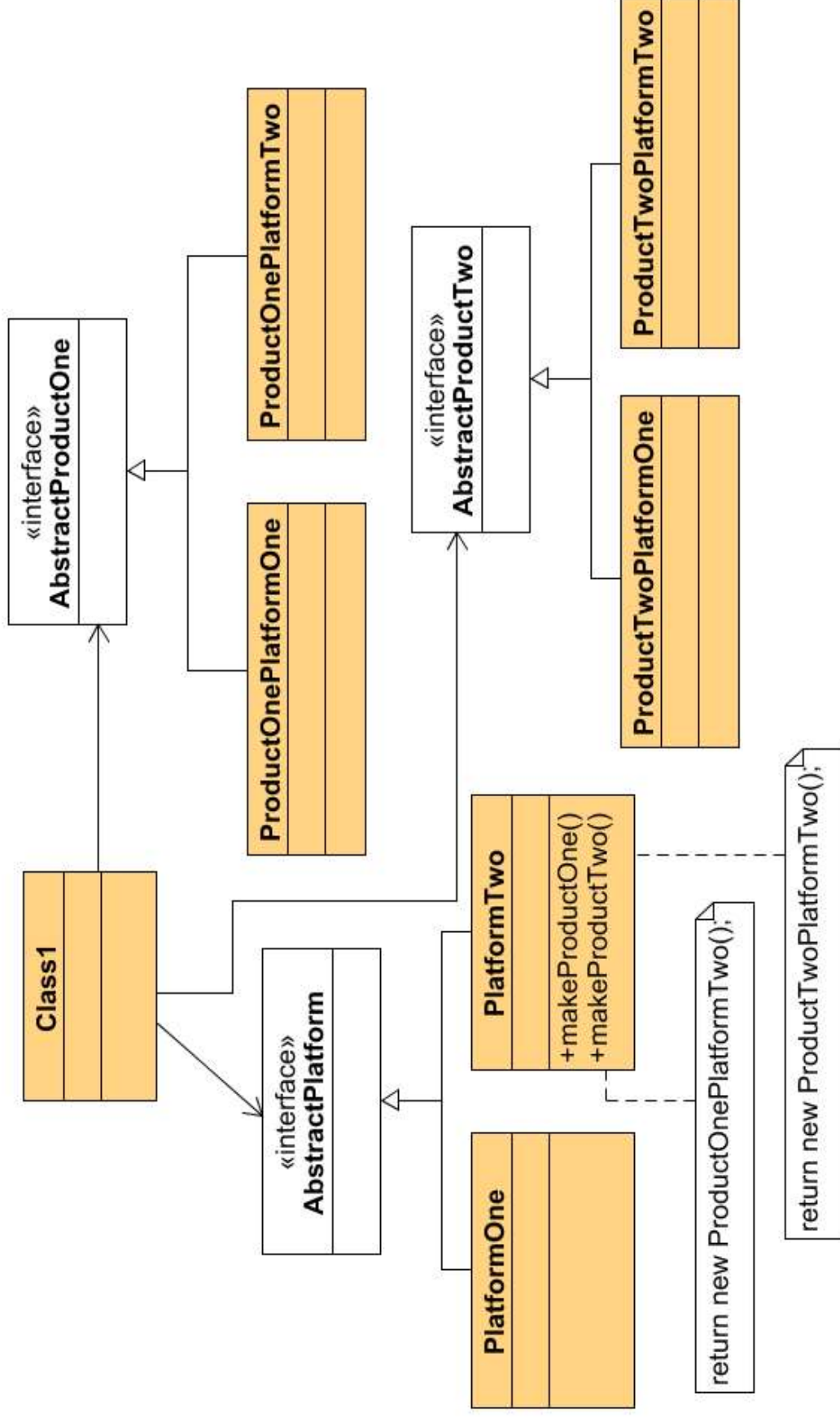
- The new operator is considered harmful.
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates many possible "platforms", and the construction of a suite of "products".

Solution

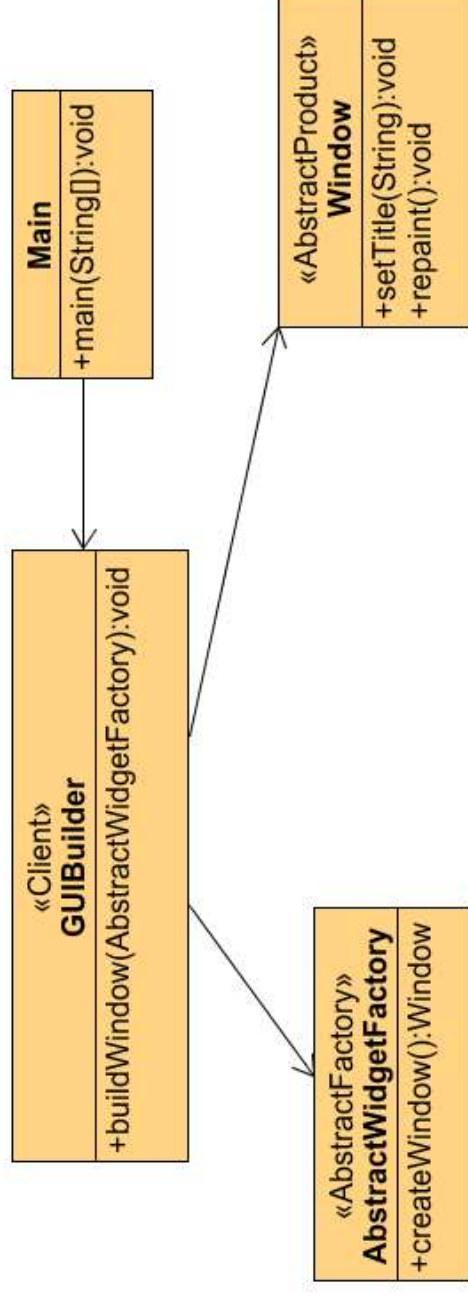
- ❖ The Abstract Factory defines a Factory Method per product



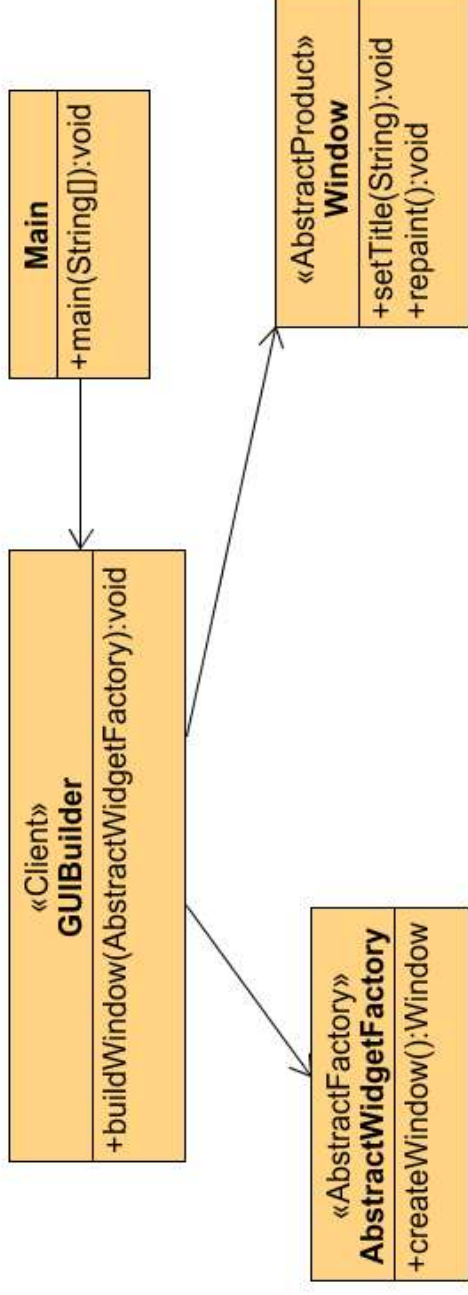
Structure



Example

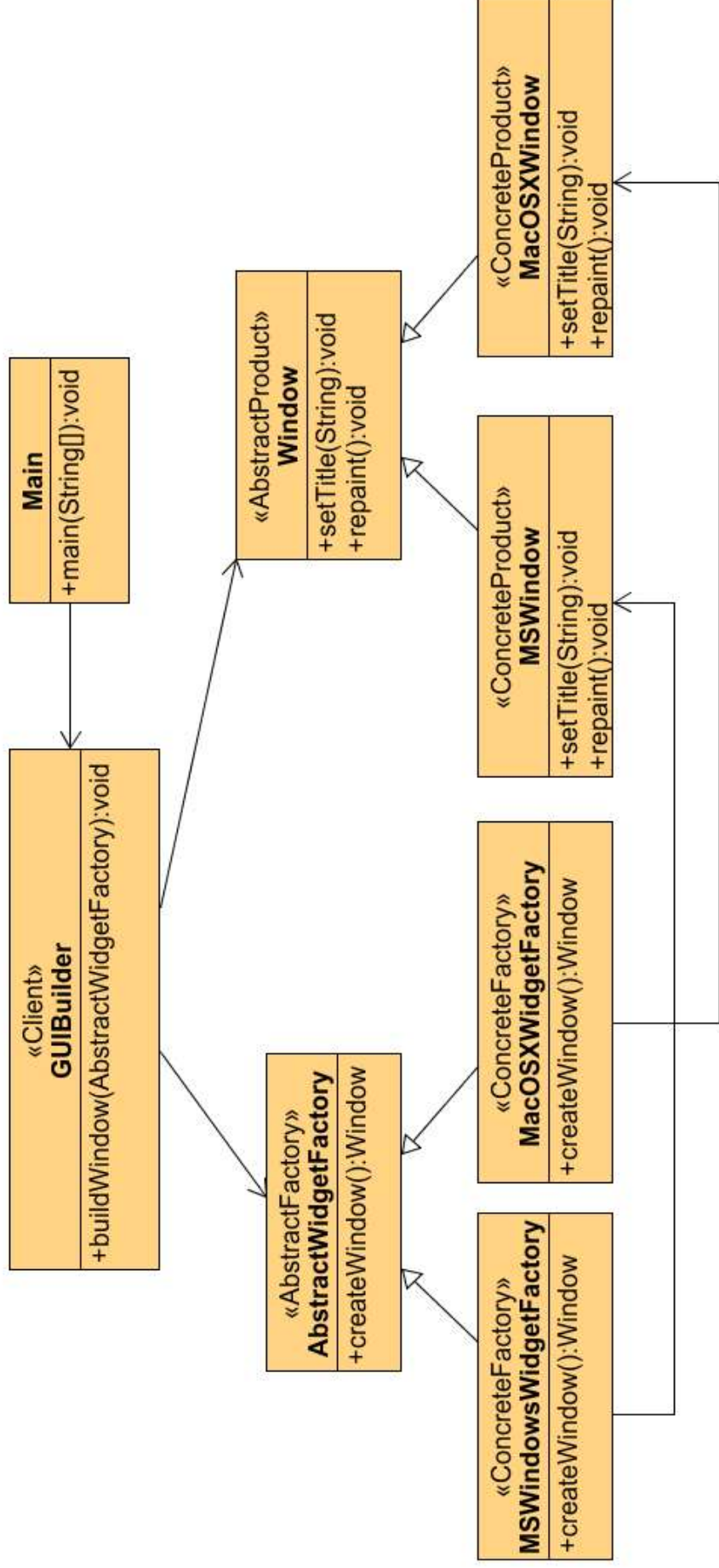


Example



```
public class GUIBuilder {  
    public void buildWindow(AbstractWidgetFactory widgetFactory) {  
        Window window = widgetFactory.createWindow();  
        window.setTitle("New Window");  
    }  
}
```


Example

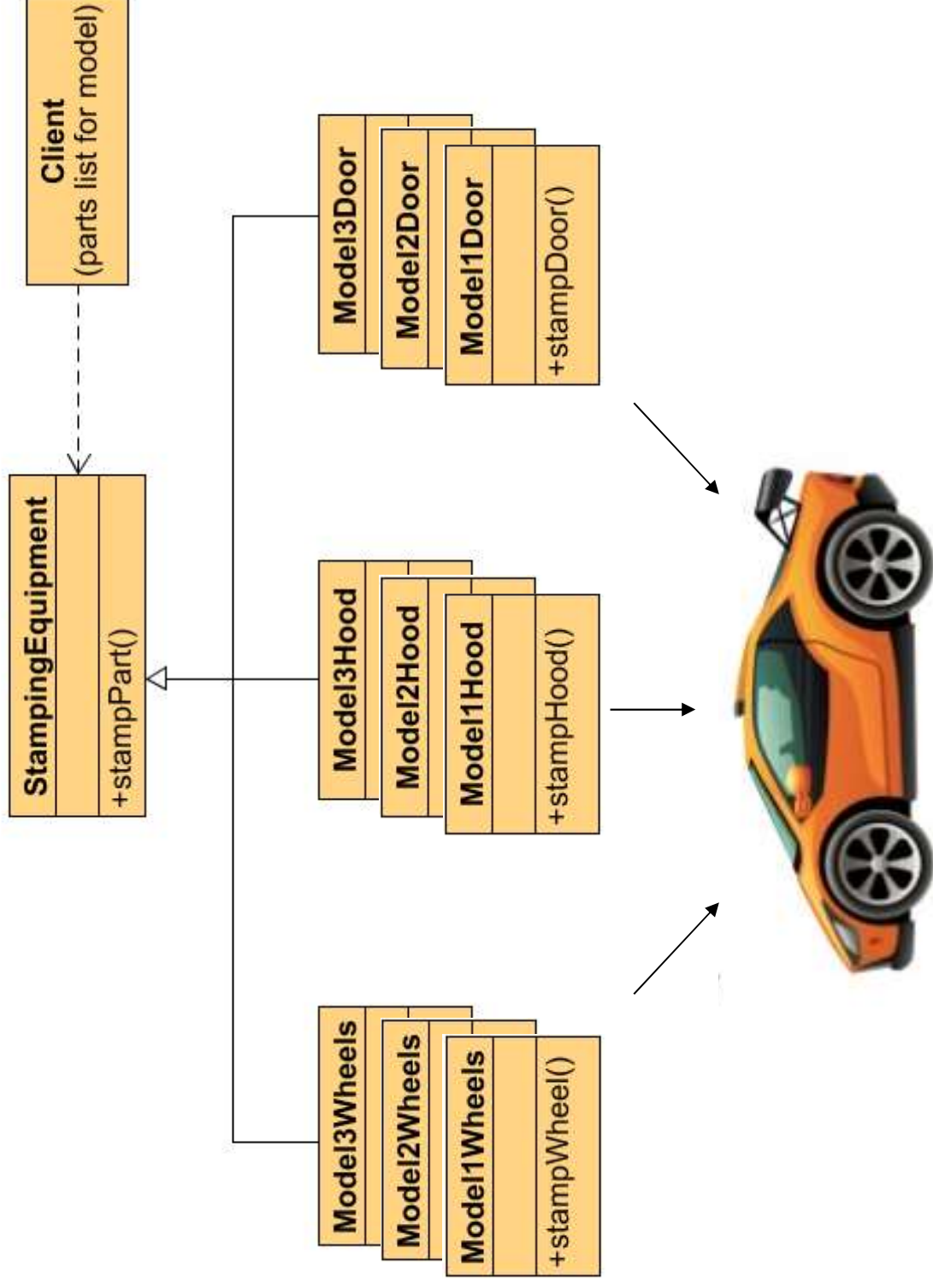


Example

```
public class MainTest {
    public static void main(String[] args) {
        GUIBuilder builder = new GUIBuilder();
        if (Platform.currentPlatform() == "MACOSX")
            builder.buildWindow(new MacOSXWidgetFactory());
        else if (Platform.currentPlatform() == "WIN")
            builder.buildWindow(new MsWindowsWidgetFactory());
        else //...
    }
}

public class GUIBuilder {
    public void buildWindow(AbstractWidgetFactory widgetFactory) {
        Window window = widgetFactory.createWindow();
        window.setTitle("New Window");
    }
}
```

Another example



Check list

- ❖ Decide if "platform independence" and creation services are the current source of pain.
- ❖ Map out a matrix of "platforms" versus "products".
- ❖ Define a factory interface that consists of a factory method per product.
- ❖ Define a factory derived class for each platform that encapsulates all references to the new operator.
- ❖ The client should retire all references to new, and use the factory methods to create the product objects.

Builder

Class

❖ Factory Method

Object

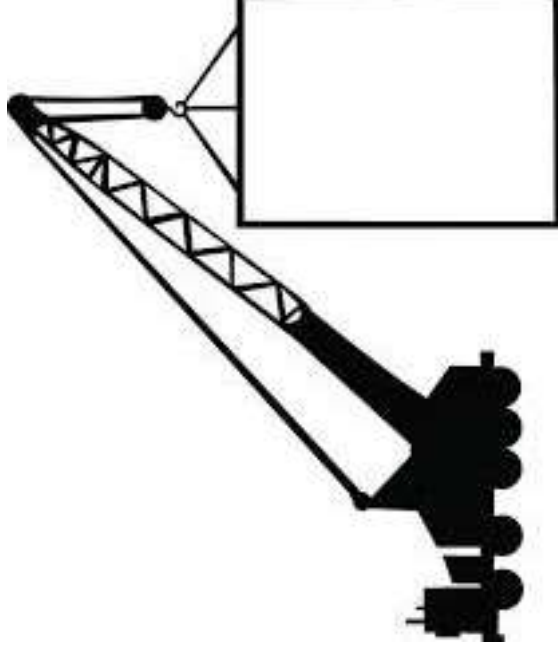
❖ Abstract Factory

❖ **Builder**

❖ Singleton

❖ Object Pool

❖ Prototype



Motivation

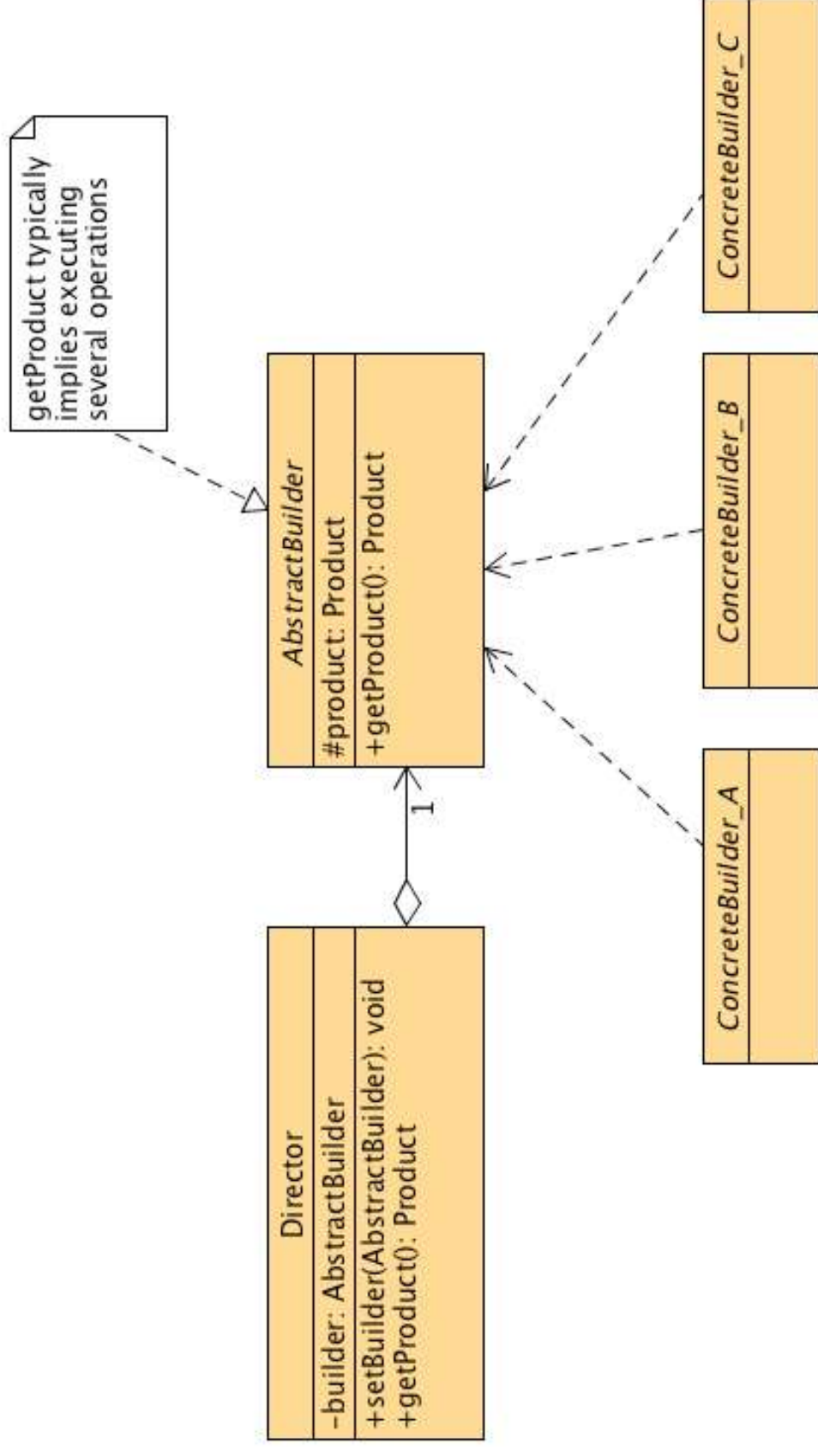
❖ Intent

- 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.

❖ Problem

- An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

Structure



Example (1)

```
class Pizza { /* "Product" */
```

```
    private String dough;
```

```
    private String sauce;
```

```
    private String topping;
```

```
    public void setDough(String dough) { this.dough = dough; }
```

```
    public void setSauce(String sauce) { this.sauce = sauce; }
```

```
    public void setTopping(String topping) { this.topping = topping; }
```

```
    public String toString() { /* .. */ }
```

```
}
```

```
abstract class PizzaBuilder { /* "Abstract Builder" */
```

```
    protected Pizza pizza = new Pizza();
```

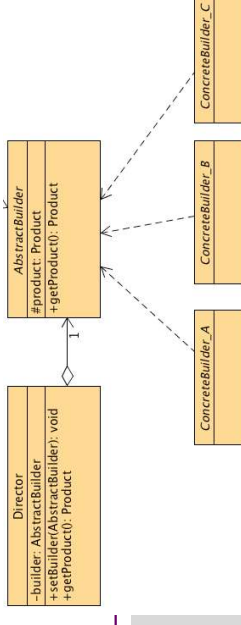
```
    public Pizza getPizza() { return pizza; }
```

```
    public abstract void buildDough();
```

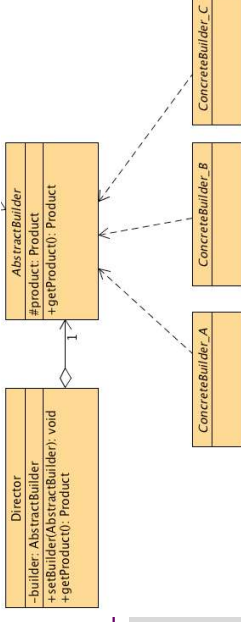
```
    public abstract void buildSauce();
```

```
    public abstract void buildTopping();
```

```
}
```



Example (2)



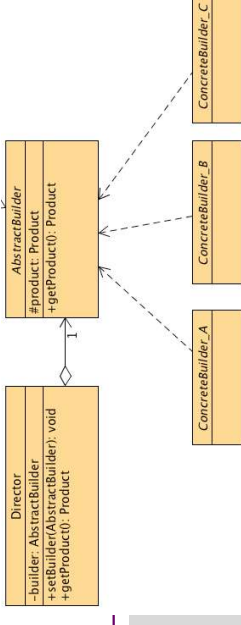
```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("cross"); }
    public void buildSauce() { pizza.setSauce("mild"); }
    public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("pan baked"); }
    public void buildSauce() { pizza.setSauce("hot"); }
    public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}
```

Example (3)

```
class Waiter { /* "Director" */
    private PizzaBuilder pizzaBuilder;

    public void setPizzaBuilder(PizzaBuilder pb) {
        pizzaBuilder = pb;
    }
    public void constructPizza() {
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
    public Pizza getPizza() {
        return pizzaBuilder.getPizza();
    }
}
```



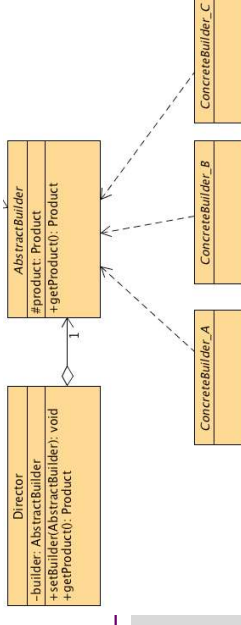
Example (4)

```
/* A customer ordering a pizza. */
class BuilderExample {
    public static void main(String[] args) {
        Waiter waiter = new Waiter();

        waiter.setPizzaBuilder(new HawaiianPizzaBuilder());
        waiter.constructPizza();
        Pizza pizza = waiter.getPizza();
        System.out.println(pizza);

        waiter.setPizzaBuilder(new SpicyPizzaBuilder());
        waiter.constructPizza();
        pizza = waiter.getPizza();
        System.out.println(pizza);
    }
}
```

```
Pizza [dough=cross, sauce=mild, topping=ham+pineapple]
Pizza [dough=pan baked, sauce=hot, topping=pepperoni+salam]
```



Check list

- ❖ Decide if a common input and many possible representations (or outputs) is the problem at hand.
- ❖ Encapsulate the parsing of the common input in a Reader class (*the Director*).
- ❖ Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.
- ❖ Define a Builder derived class for each target representation.
- ❖ The client creates a Reader object and a Builder object, and registers the latter with the former.
- ❖ The client asks the Reader to "construct".
- ❖ The client asks the Builder to return the result.

Another example – slightly different

- ❖ Consider a builder when faced with many constructors
- ❖ Use a builder inner class

Another example

```
public class NutritionFacts {
    private final int servingsSize;           // (mL) required
    private final int servings;              // (per container)
    private final int calories;              // optional
    private final int fat;                   // (g) optional
    private final int sodium;               // (mg) optional
    private final int carbohydrate;         // (g) optional

    public NutritionFacts(int servingsSize, int servings,
                          int calories, int fat, int sodium,
                          int carbohydrate) {
        this.servingsSize = servingsSize;
        this.servings = servings;
        this.calories = calories;
        this.fat = fat;
        this.sodium = sodium;
        this.carbohydrate = carbohydrate;
    }
}
```

What's wrong?

Example – more constructors

```
public NutritionFacts(int servingSize, int servings) {
    this(servingSize, servings, 0);
}

public NutritionFacts(int servingSize, int servings,
    int calories) {
    this(servingSize, servings, calories, 0);
}

public NutritionFacts(int servingSize, int servings,
    int calories, int fat) {
    this(servingSize, servings, calories, fat, 0);
}

public NutritionFacts(int servingSize, int servings,
    int calories, int fat, int sodium) {
    this(servingSize, servings, calories, fat, sodium, 0);
}
```

Still
wrong?

Example – with Builder (1)

```
public class NutritionFacts { // Builder Pattern
    private final int servingsSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingsSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;

        //...
```


Example – with Builder (2)

```
public Builder(int servingSize, int servings) {
    this.servingSize = servingSize; this.servings = servings;
}

public Builder calories(int val) {
    calories = val;
    return this;
}

public Builder fat(int val) {
    fat = val;
    return this;
}

public Builder carbohydrate(int val) {
    carbohydrate = val;
    return this;
}

public Builder sodium(int val) {
    sodium = val;
    return this;
} //...
```

Example – with Builder (3)

```
public NutritionFacts build() {  
    return new NutritionFacts(this);  
}  
    // end of class Builder  
  
private NutritionFacts(Builder builder) {  
    servingSize = builder.servingSize;  
    servings = builder.servings;  
    calories = builder.calories;  
    fat = builder.fat;  
    sodium = builder.sodium;  
    carbohydrate = builder.carbohydrate;  
}  
}
```

We can now use this static inner class as follows:

```
NutritionFacts sodaDrink = new NutritionFacts.Builder(240, 8).  
    calories(100).sodium(35).carbohydrate(27).build();
```

Builders in the JDK

- ❖ All implementations of `java.lang.Appendable` are good examples of use of Builder pattern in Java.

```
public static void main(String[] args) {  
    String data = new StringBuilder("Exemplo de builder_")  
        .append(1)  
        .append(true)  
        .append("_para_fechar")  
        .toString();  
    System.out.println(data);  
}
```

Exemplo de `builder_1true_para_fechar`

Singleton

Class

❖ Factory Method

Object

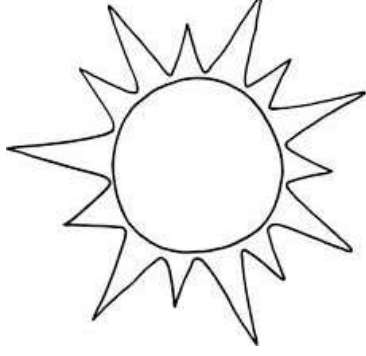
❖ Abstract Factory

❖ Builder

❖ Singleton

❖ Object Pool

❖ Prototype



Motivation

❖ Intent

- Ensure a class has only one instance, and provide a global point of access to it.
- Encapsulated "just-in-time initialization" or "initialization on first use".

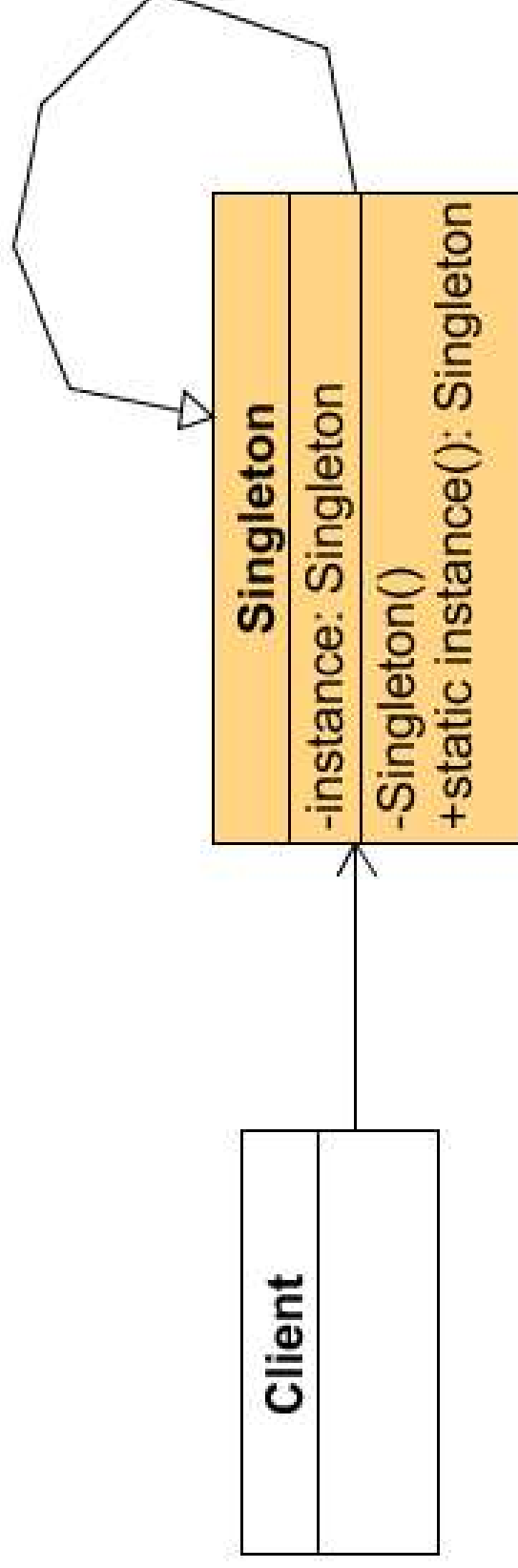
❖ Problem

- Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.

Solution

- ❖ Define the constructor as private (or protected))
 - `private Singleton(String name)`
- ❖ Define a private static reference to the single class object
 - `static private Singleton instance`
- ❖ Define a accessor method to that instance
 - `static public Singleton getInstance ()`
 - Customers can access only the singleton object through this method

Structure



Example

```
class Singleton {  
    private String name;  
    static private Singleton instance = new Singleton("Ermita");  
    private Singleton(String name) {  
        this.name = name;  
    }  
    static public Singleton getInstance() {  
        return instance;  
    }  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```


Example – lazy initialization

```
class LazySingleton {  
    private String name;  
  
    static private LazySingleton instance=null;  
  
    private LazySingleton(String name) {  
        this.name = name;  
    }  
    static public synchronized LazySingleton getInstance() {  
        if (instance == null) {  
            instance = new LazySingleton("Ermita");  
        }  
        return instance;  
    }  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

Check list

- ❖ Define a private static attribute in the "single instance" class.
- ❖ Define a public static accessor function in the class.
- ❖ Do "lazy initialization" (creation on first use) in the accessor function.
- ❖ Define all constructors to be protected or private.
- ❖ Clients may only use the accessor function to manipulate the Singleton.

Object Pool

Class

❖ Factory Method

Object

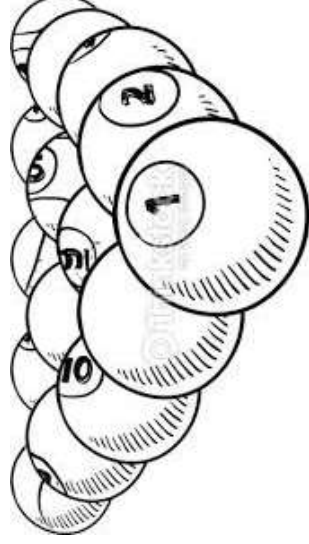
❖ Abstract Factory

❖ Builder

❖ Singleton

❖ **Object Pool**

❖ Prototype



Motivation

❖ Intent

- Object pooling can offer a significant performance boost; it is most effective in situations where:
 - the cost of initializing a class instance is high,
 - the rate of instantiation of a class is high, and
 - the number of instantiations in use at any one time is low.

❖ Problem

- Object are used to manage the object caching. A client with access to a Object pool can avoid creating a new Object by simply asking the pool for one that has already been instantiated instead.
- It is desirable to keep all Reusable objects that are not currently in use in the same object pool so that they can be managed by one coherent policy.

Solution

(1) redShoes = Shelf.acquireShoes();

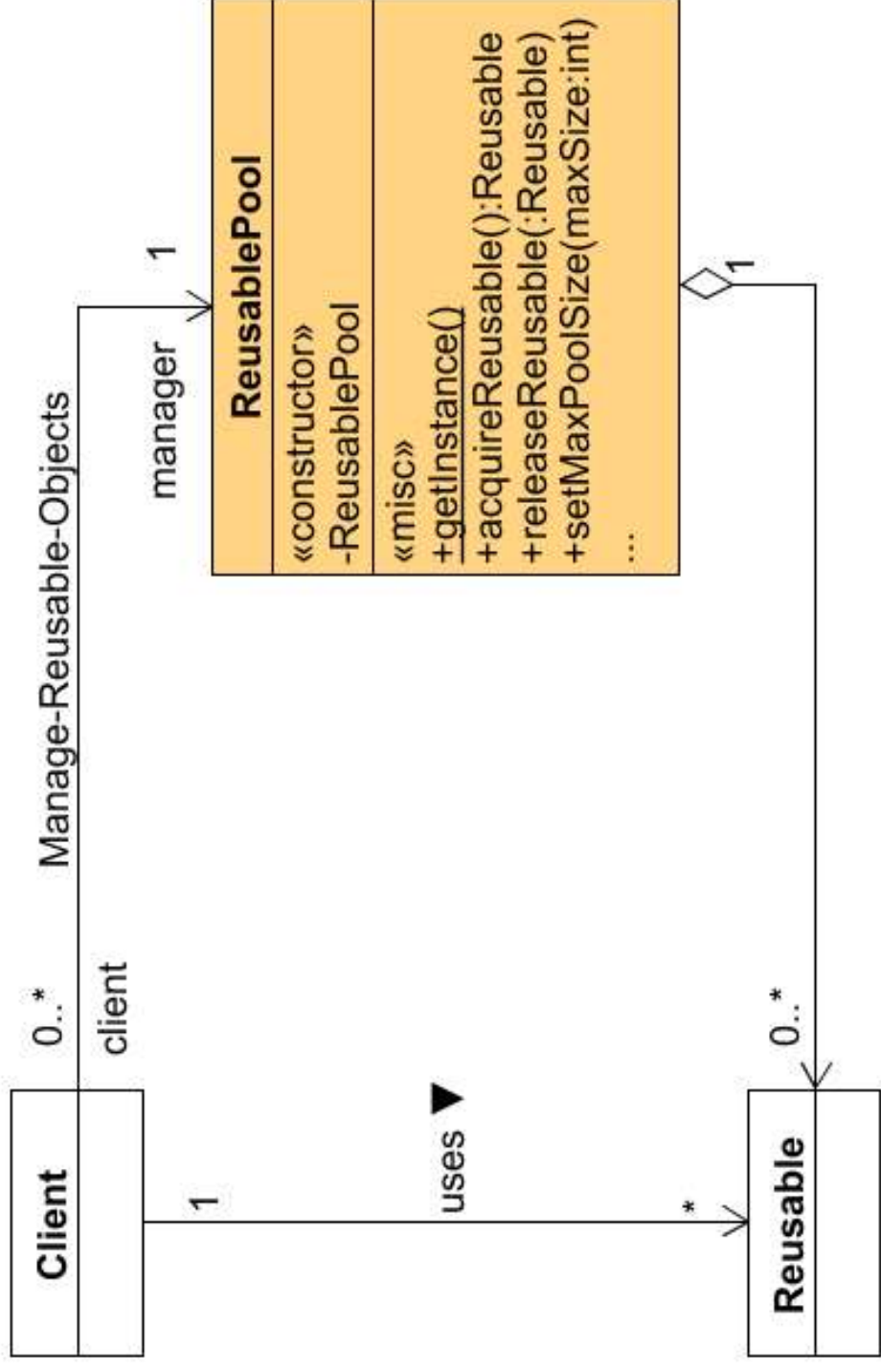
(2) client.wear(redShoes);



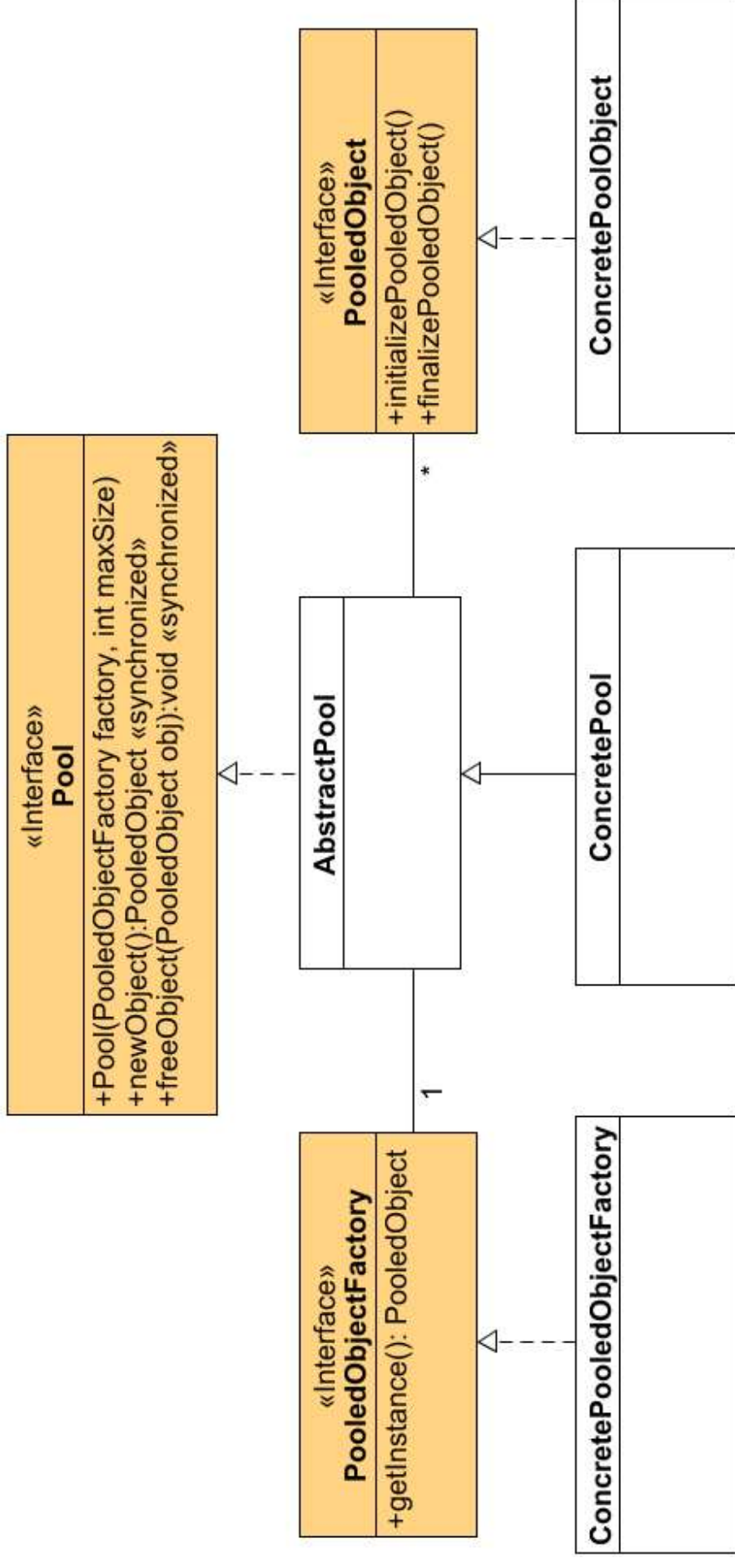
(4) Shelf.releaseShoes(redShoes);

(3) client.play();

Structure



A more complete Structure



Example - PooledObject

```
/**
 * Interface that has to be implemented by an object that can be
 * stored in an object pool through the Pool class.
 * http://www.devahead.com
 */
public interface PooledObject
```

```
{
    /**
```

```
 * Initialization method. Called when an object is retrieved
 * from the object pool or has just been created.
 */
```

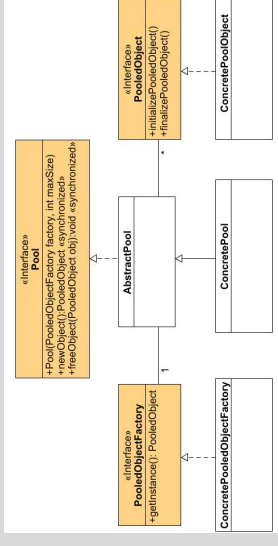
```
    public void initializePooledObject();
```

```
    /**
```

```
 * Finalization method. Called when an object is stored in
 * the object pool to mark it as free.
 */
```

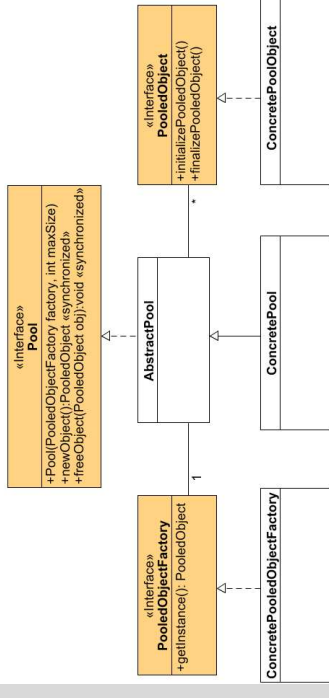
```
    public void finalizePooledObject();
```

```
}
```



Example - PooledObjectFactory

```
/**
 * Interface that has to be implemented by every class that allows
 * the creation of objects for an object pool through the
 * Pool class.
 */
public interface PooledObjectFactory
{
    /**
     * Creates a new object for the object pool.
     *
     * @return new object instance for the object pool
     */
    public PooledObject getInstance();
}
```

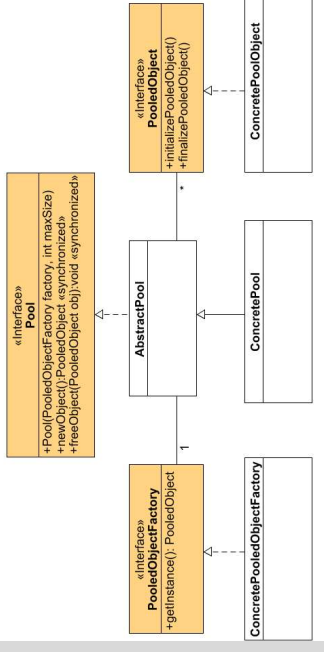


Example - AbstractPool

```
public class AbstractPool implements Pool
{
    protected final int MAX_FREE_OBJECT_INDEX;

    protected PooledObjectFactory factory;
    protected PooledObject[] freeObjects;
    protected int freeObjectIndex = -1;
```

```
/**
 * @param factory the object pool factory instance
 * @param maxSize the maximum number of instances stored in the pool
 */
public AbstractPool(PooledObjectFactory factory, int maxSize)
{
    this.factory = factory;
    this.freeObjects = new PooledObject[maxSize];
    MAX_FREE_OBJECT_INDEX = maxSize - 1;
}
```



Example – *AbstractPool.newObject*

```
/**
 * Creates a new object or returns a free object from the pool.
 * @return a PooledObject instance already initialized
 */
public synchronized PooledObject newObject() {
    PooledObject obj = null;

    if (freeObjectIndex == -1) {
        // There are no free objects so I just
        // create a new object that is not in the pool.
        obj = factory.getInstance();
    } else {
        // Get an object from the pool
        obj = freeObjects[freeObjectIndex];
        freeObjectIndex--;
    }
    obj.initializePooledObject();
    return obj;
}
```

Example - *AbstractPool.freeObject*

```
/**
 * Stores an object instance in the pool to make it available for a subsequent
 * call to newObject() (the object is considered free).
 * @param obj the object to store in the pool and that will be finalized
 */
public synchronized void freeObject(PooledObject obj)
{
    if (obj != null) {
        // Finalize the object
        obj.finalizePooledObject();
        // put an object in the pool only if there is still room for it
        if (freeObjectIndex < MAX_FREE_OBJECT_INDEX) {
            freeObjectIndex++;
            // Put the object in the pool
            freeObjects[freeObjectIndex] = obj;
        }
    }
}
```

Check list

- ❖ Create the *Pool* class with a collection of *PooledObjects*
- ❖ Create *acquire* and *release* methods in *Pool* class

Important remarks

- The creation and destruction of short lived objects (i.e. memory allocation and GC) is more efficient in modern JVMs.
- Object Pool must only be used for special objects whose creation is relatively costly, like DB / network connections, threads etc.

Prototype

Class

- ❖ Factory Method

Object

- ❖ Abstract Factory

- ❖ Builder

- ❖ Singleton

- ❖ Object Pool

- ❖ **Prototype**



Motivation

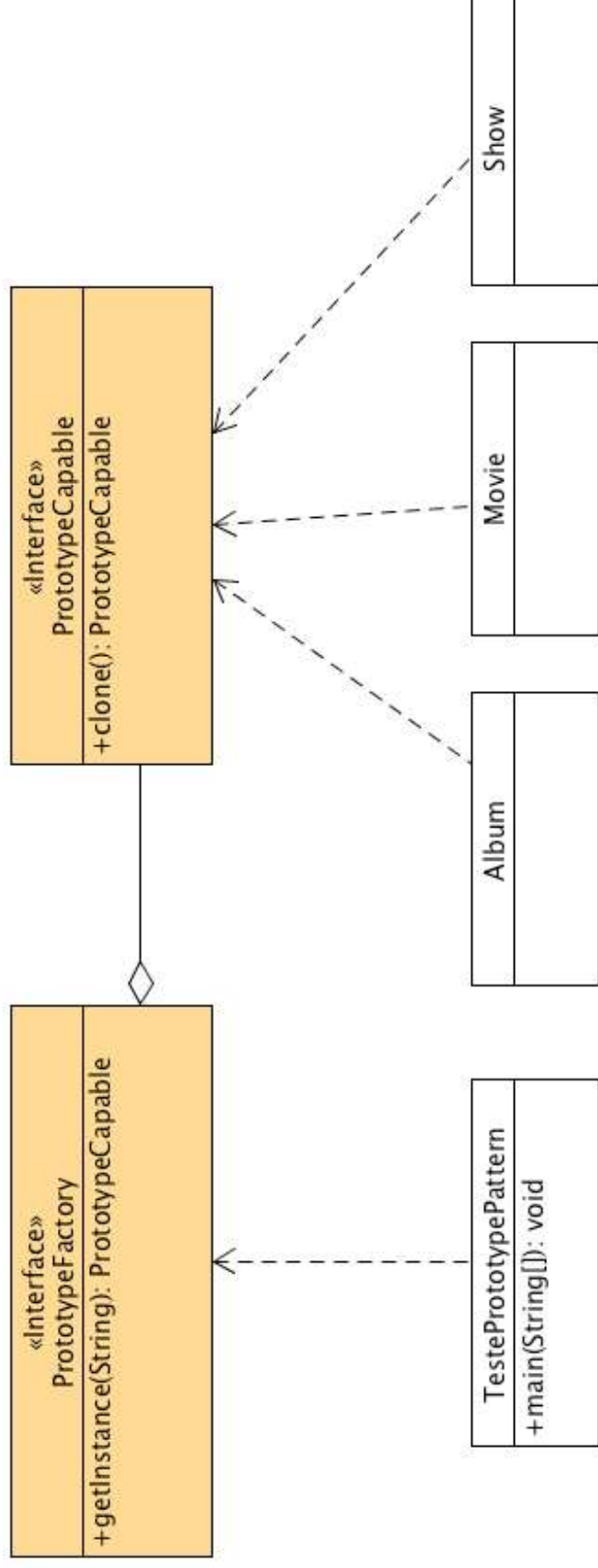
❖ Intent

- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- Co-opt one instance of a class for use as a breeder of all future instances.
- The new operator considered harmful.

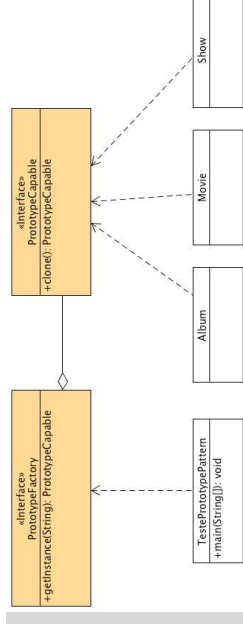
❖ Problem

- Application "hard wires" the class of object to create, in each "new" expression.

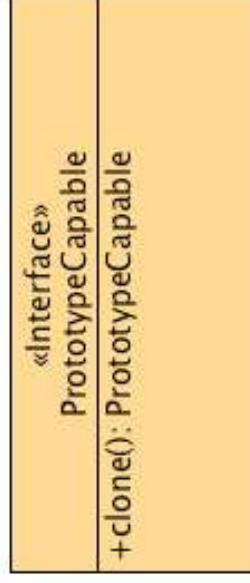
Structure



Example – the contract

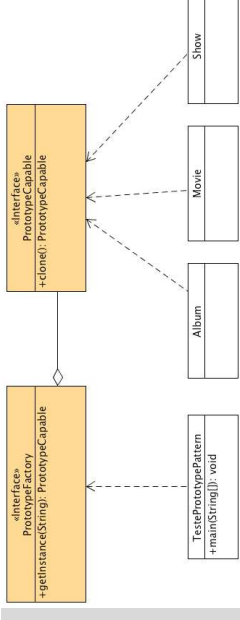


```
public interface PrototypeCapable extends Cloneable
{
    public PrototypeCapable clone() throws CloneNotSupportedException;
}
```



Example – the model

```
public class Album implements PrototypeCapable
{
    private String name = null;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public Album clone() throws CloneNotSupportedException {
        System.out.println("Cloning Album object..");
        return (Album) super.clone();
    }
    @Override
    public String toString() {
        return "Album";
    }
}
```



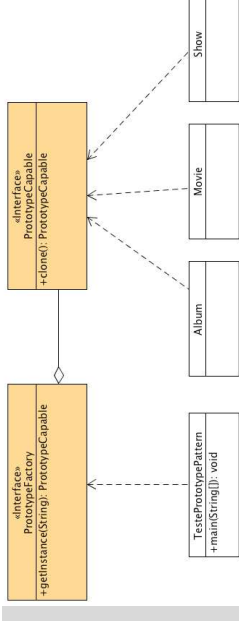
the same for Movie, Show, ..

Example – the factory

```
public class PrototypeFactory {  
    public static enum ModelType {  
        MOVIE, ALBUM, SHOW;  
    }  
}
```

```
private static Map<ModelType, PrototypeCapable> prototypes =  
    new HashMap<>();  
static {  
    prototypes.put(ModelType.MOVIE, new Movie());  
    prototypes.put(ModelType.ALBUM, new Album());  
    prototypes.put(ModelType.SHOW, new Show());  
}
```

```
public static PrototypeCapable getInstance(ModelType s)  
    throws CloneNotSupportedException {  
    return (prototypes.get(s)).clone();  
}
```



Example – the client

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

```
            PrototypeCapable proto;
```

```
            proto = PrototypeFactory.getInstance(ModelType.MOVIE);
```

```
            System.out.println(proto);
```

```
            proto = PrototypeFactory.getInstance(ModelType.ALBUM);
```

```
            System.out.println(albumPrototype);
```

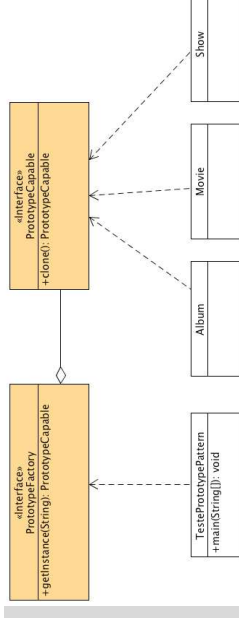
```
            proto = PrototypeFactory.getInstance(ModelType.SHOW);
```

```
            System.out.println(proto);
```

```
        } catch (CloneNotSupportedException e)
```

```
        {  
            e.printStackTrace();
```

```
        }  
    }  
}
```



```
Cloning Movie object..  
Movie  
Cloning Album object..  
Album  
Cloning Show object..  
Show
```

Check list

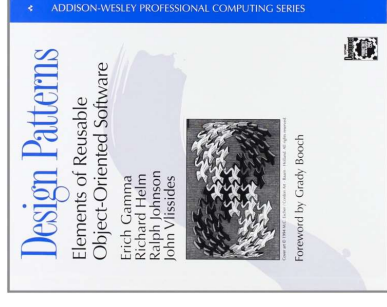
- ❖ Add a `clone()` method to the existing "product" hierarchy.
- ❖ Design a "registry" that maintains a cache of prototypical objects. The registry could be encapsulated in a new Factory class, or in the base class of the "product" hierarchy.
- ❖ Design a factory method that: may (or may not) accept arguments, finds the correct prototype object, calls `clone()` on that object, and returns the result.
- ❖ The client replaces all references to the new operator with calls to the factory method.

Creational patterns – Summary

- ❖ Abstract Factory
 - Creates an instance of several families of classes
- ❖ Builder
 - Separates object construction from its representation
- ❖ Factory Method
 - Creates an instance of several derived classes
- ❖ Singleton
 - A class of which only a single instance can exist
- ❖ Object Pool
 - Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
- ❖ Prototype
 - A fully initialized instance to be copied or cloned

Resources

- ❖ Design Patterns – Elements of Reusable Object-Oriented Software; Gamma, et. al.



- ❖ *Design Patterns Explained Simply*
(sourcemaking.com)