Prototype Pattern

Chapter Content

- Prototype Pattern Overview
- Prototype Pattern UML Diagram
- Prototype Example
- Clone() technicalities (Java)
- Deep vs. Shallow cloning
- Prototype as an alternative to Abstract Factory
- Exercise

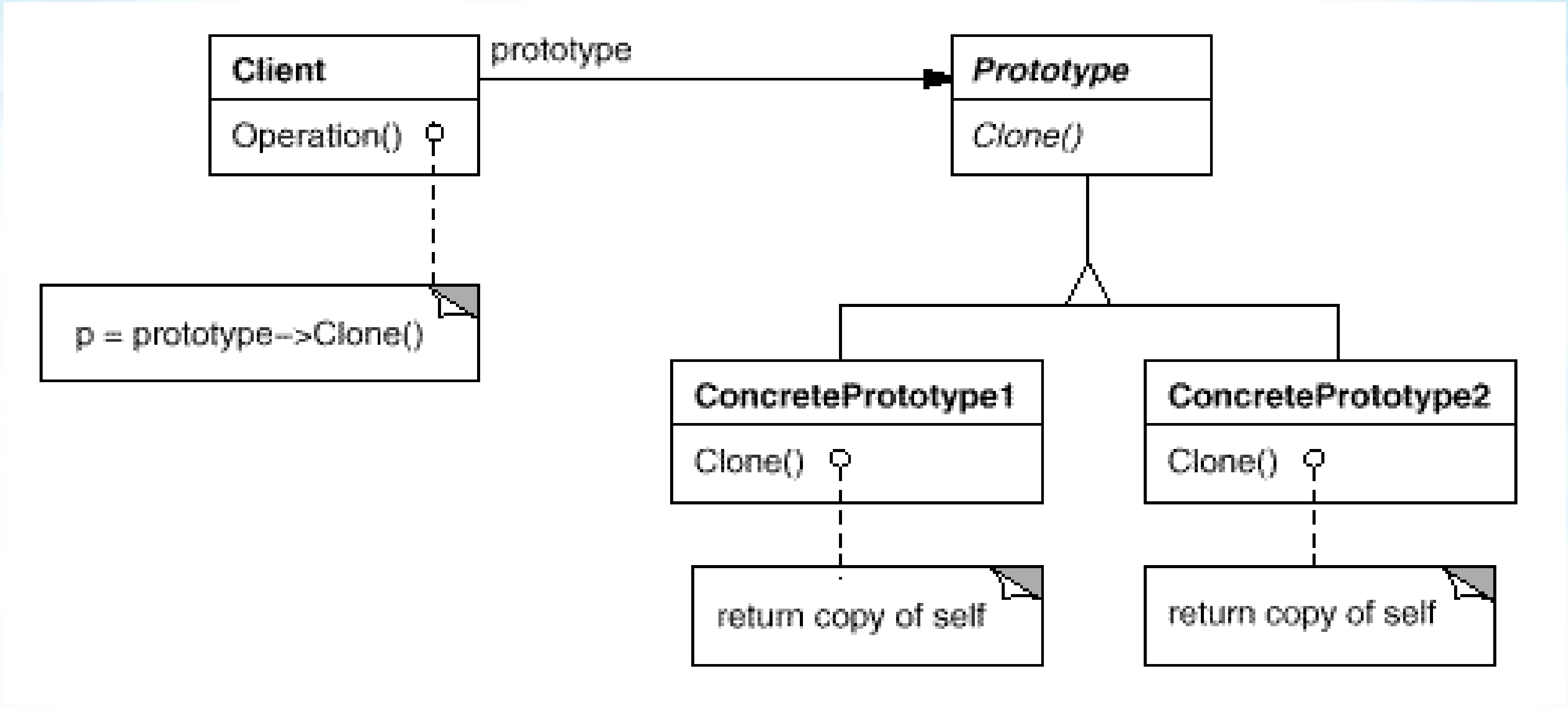
Prototype Pattern Overview

- First instance created conventionally
- Next instances will be obtained by making **copies** of the original one, modifying them if necessary.
- To obtain such copies, ask the original instance to **clone()** itself.

.)Prototype Pattern Overview (cont

- **Useful when:**
  - You need to clone an object, especially if you don’t know it’s exact sub-class (polymorphism).
  - Creating new instances is time-consuming, and it appears easier to duplicate an existing object and introduce minor changes.

Prototype Pattern UML Diagram



Prototype is requested to make duplicates of itself, by calling '*clone()*'. Important due to polymorphism – client doesn't necessarily know which concrete sub-class it is holding (ConcretePrototype1 / ConcretePrototype2).

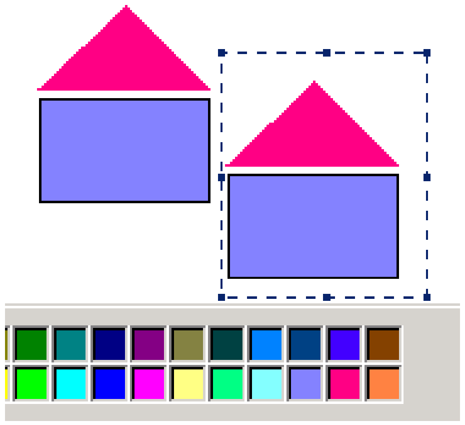
Prototype Example - Board

- **An AI program playing a board game**
  - Given current board state, try different possible moves on different duplicates of the board (Assuming we have multi-processors, we prefer cloning over un-doing steps)

```
public Board makeMove(Board board){
     for ( ... /*for each possible move*/ ) {
          Board bCopy = (Board)board.clone();
          // make move on copy & evaluate result
     }
     // return result after best move
}
```

Example - graphical editors

- Use clone when user wishes to copy-paste different shapes.
- Here, polymorphism is important: clone any shape, regardless of what sub-type it is.



Example - efficiency

- When it's costly to create the 1st instance, but later cloning is cheaper:

```
// A class that is expensive to instantiate:

public class Company implements Cloneable {
   private ArrayList employees;  // List of Employee Objects

   public Company()       {…}   // Load DB data into vector (costly)

   public Object clone()  {…}       // Copy vector data

}


Usage:

Company current = new Company();                          // Costly 1st object

Company nextYearsPrediction = (Company) current.clone();  // cheap

... // modify employee list & salaries according to predictions
```

Clone() technicalities (Java)

- To allow your class to be cloned:
  - Declare it ***implements Cloneable*** . Java will then automatically provide a clone() method.
- **However:**
  - The default clone() is **protected even if your class implements Cloneable**. You may wish to make it public.
  - The default clone is somewhat **slower** than regular allocation (since it’s native).

Technicalities (cont.)

- And:
  - The default *clone()* performs shallow copy. Override at need !

Deep vs. Shallow cloning

InterBit Training & Consulting Ltd.

Employee – shallow copy is enough:

```
public class Employee extends Object implements Cloneable {
     private String name;
     private long id;
     private float salary;

     // Override to make it public:
     public Object clone(){
          try{
               return super.clone();
          } catch(CloneNotSupportedException ex){ … }
     }
     // ...
}
```

Why is it not necessary to clone a String member ?

Deep vs. Shallow cloning (cont.)

```
public class Company implements Cloneable {

   private ArrayList<Employee> employees;
   private String    address;
   private float     assets;

   public Object clone() {  // assuming we need deep cloning
       try{
            Company copy=(Company)super.clone(); // shallow
             copy.employees = (ArrayList)employees.clone(); // shallow
             copy.employees = new ArrayList<Employee>(); // deep
            for(Employee employee : this.employees)  {
                copy.employees.add(employee.clone());
            }
            return copy;
      } catch(CloneNotSupportedException ex){ … }
  }
```

ArrayList & Employee must be Cloneable with a public clone method !

Deep vs. Shallow cloning (cont.)

- **Does it make sense?**: if Employee also points to its company, how would we clone them?

```
class Company implements Cloneable{
    protected float sumSalaries;
    private Employee [] employees;
    public Object clone() {   ???   }
}
class Employee implements Cloneable{
    private Company company;
    private float salary;
    public void setSalary(float salary) {
       company.sumSalaries += (salary – this.salary);
       this.salary = salary;
    }
    public Object clone() {  ???  }
```

Among other things, consider: Aggregation or Association ?

Alternative to Abstract Factory

Compare with the AbstractFactory

```
public class GuiFactory {
     private Button button;
     private ComboBox combo;
     public GuiFactory(Button button, ComboBox combo){
          this.button = button;
          this.combo = combo;
     }
     public Button createButton(){
          return (Button)button.clone();
     }
     public ComboBox createComboBox(){
          return (ComboBox)combo.clone();
     }
}

Usage:
GuiFactory winFactory=new GuiFactory( new WinButton(), new WinCombo());
Button bt = winFactory.createButton();
```