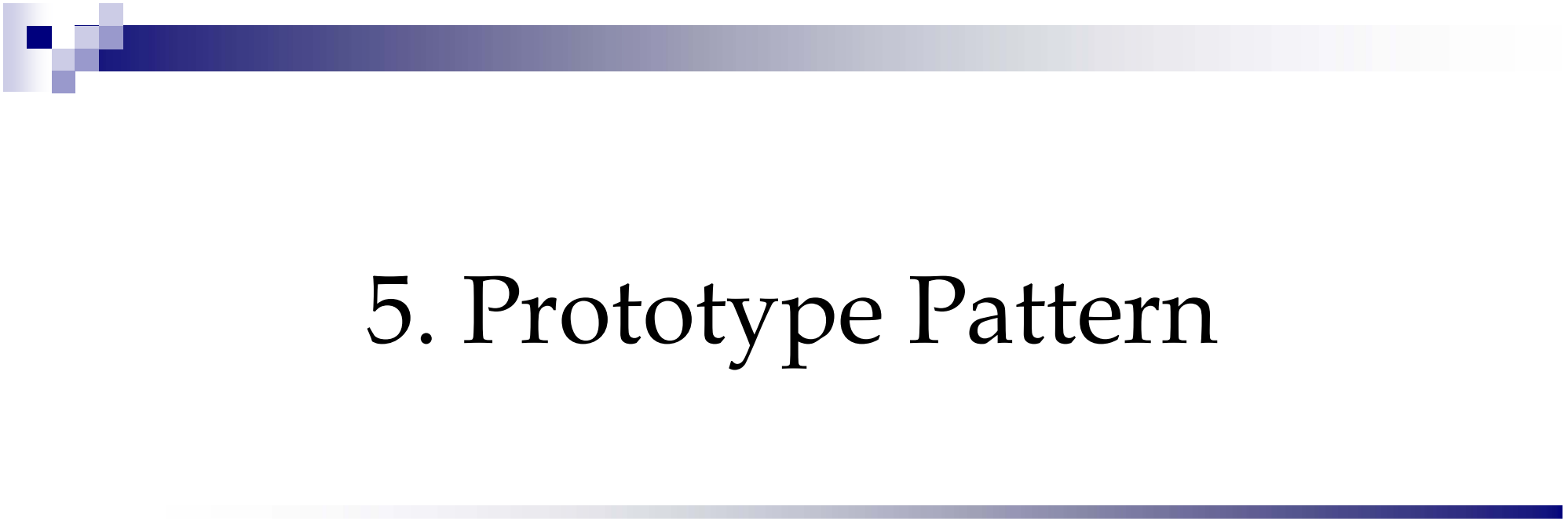
Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern University



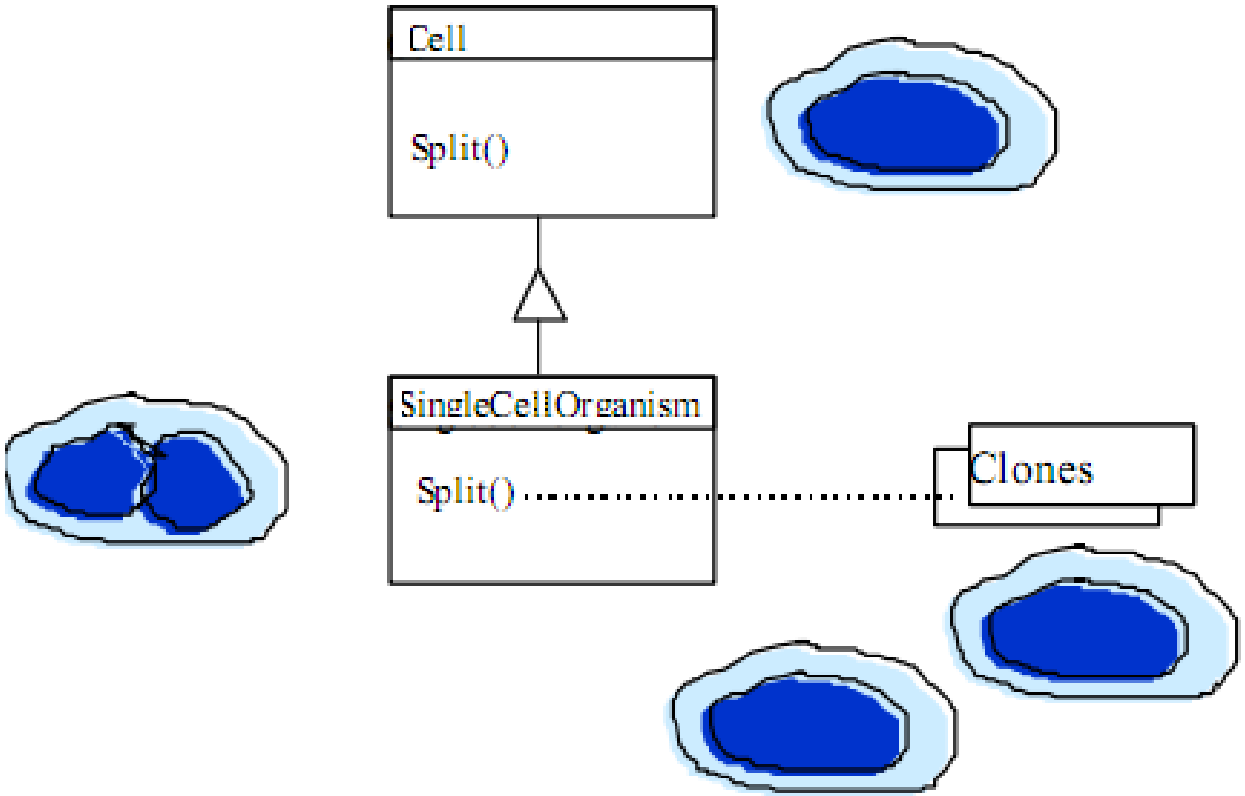
5. Prototype Pattern

Intent

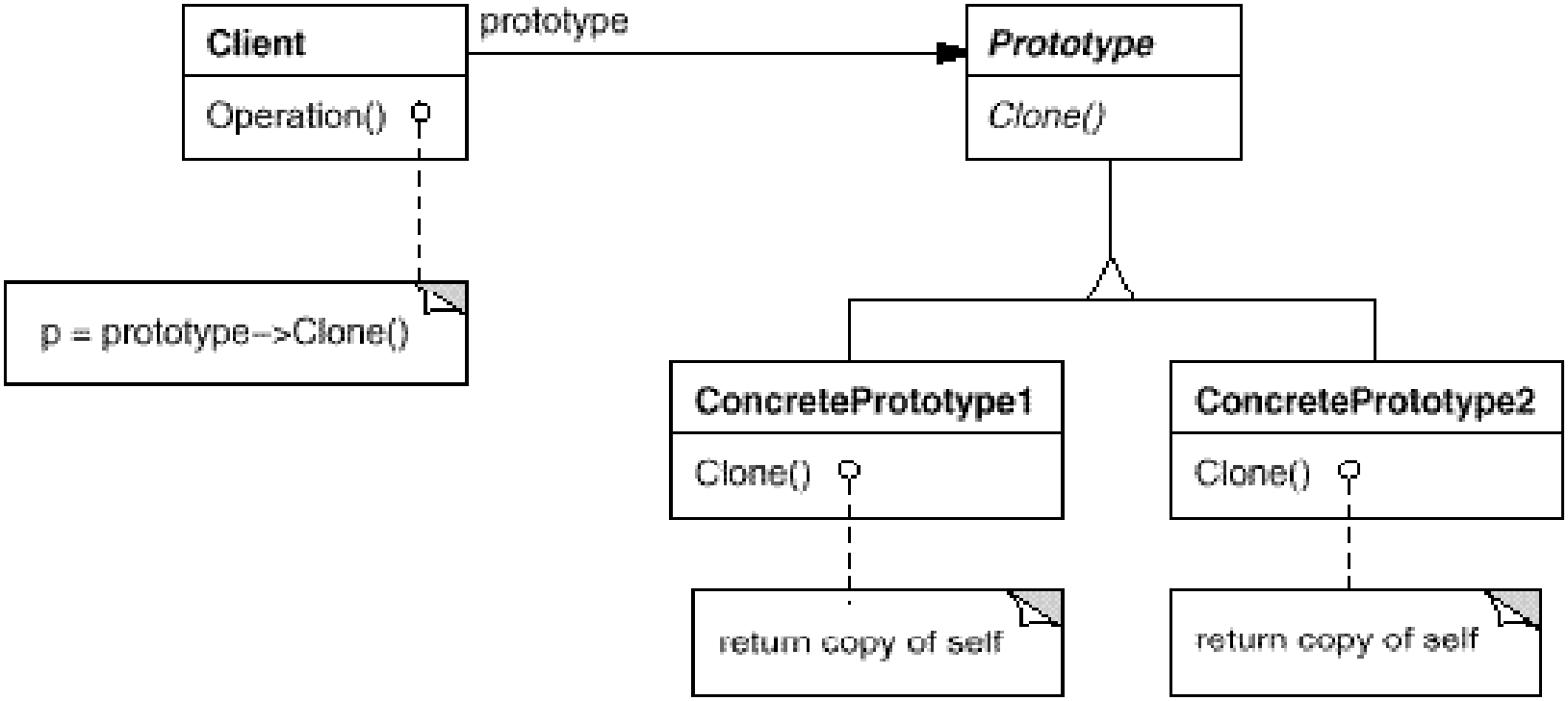
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
- 通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。
- For some objects which are :
  - Complex in internal structure;
  - Difficult to create or unable to create;
  - Complex in initial state;

  can be created by clone the prototypical instance.

Example



Structure

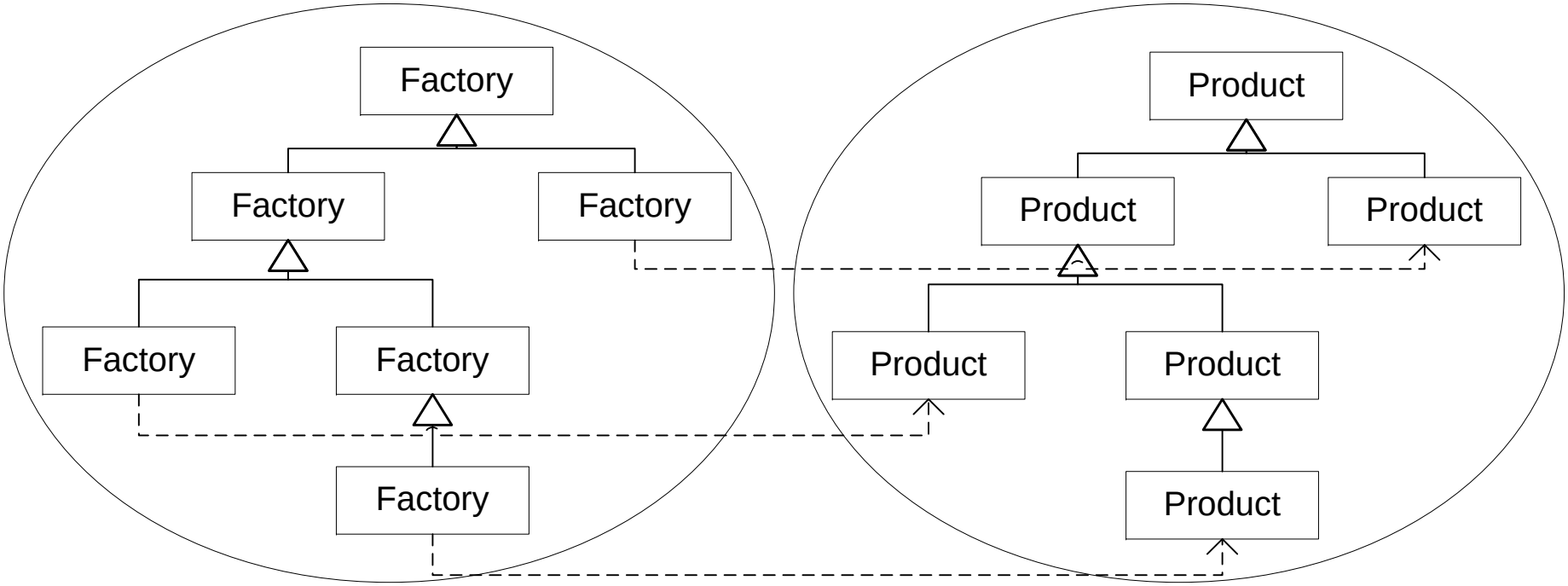

Client
Operation()
prototype
Prototype
Clone()
p = prototype->Clone()
ConcretePrototype1
Clone()
ConcretePrototype2
Clone()
return copy of self
return copy of self

Participants

- Prototype: Declares an interface for cloning itself.
- ConcretePrototype: Implements an operation for cloning itself.
- Client: Creates a new object by asking a prototype to clone itself.

Consequences

- Same consequences that Factory and Builder have;
- Adding and removing products at run-time;
- Reducing the structure of creators.
- Each subclass of prototype must implement the Clone operation, which may be difficult.

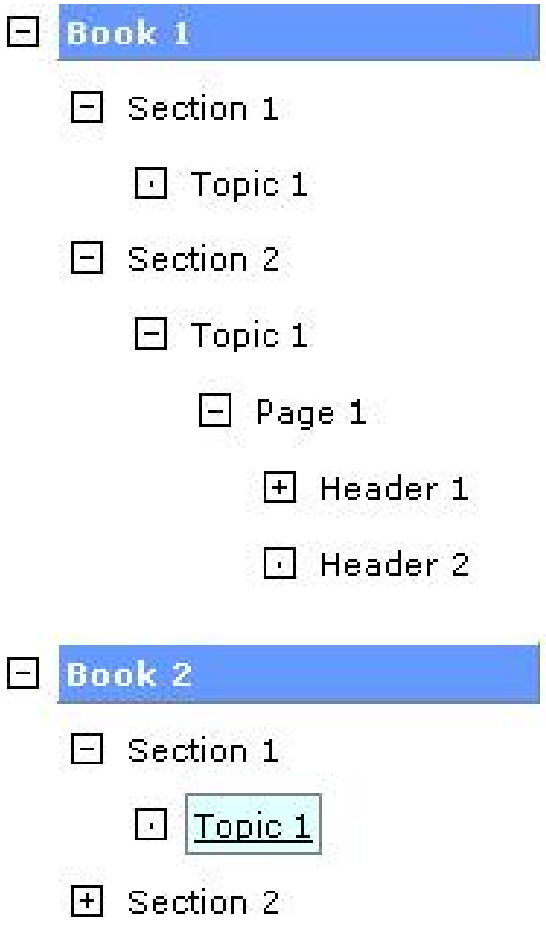

Factory
Factory
Factory
Factory
Factory
Factory
Product
Product
Product
Product
Product
Product

Applicability

- Use the prototype pattern when a system should be independent of how its products are created, composed, and represented; *AND* [in'stænʃieit]
- When the classes to instantiate are specified at run-time, for example, by dynamic loading; *OR*
- To avoid building a class hierarchy of factories that parallels class hierarchy of products; OR
- When instances of a class can have one of only a few different combinations of state. It may be more convenient to install a corresponding number of prototypes and clone them rather than instantiating the class manually, each time with the appropriate state.

Example

- Tree-viewed system menu



Extension 1: Clone method

- The class includes the clone() method for objects to make copies of themselves.
  - A copied object will be a new object instance, separate from the original.
  - The copied object may or may not contain exactly the same state (the same instance variable values) as the original. The state is controlled by the object being copied.
  - The decision as to whether the object allows itself to be cloned at all is up to the object.

Extension 2: Clone in Java

- clone() is a method in the Java programming language for object duplication.
- In Java, objects are manipulated through reference variables, and there is no operator for copying an object.
  - The assignment operator duplicates the reference, not the object.
- The clone() method provides this functionality. clone() acts like a constructor.

Extension 2: Clone in java.lang.Object

- The mechanisms provided by the Java Object class is used to make a simply copy (shallow copy) of an object including all of its state;
- By default, this capability is turned off. In order for an object to be considered cloneable, an object must implement the java.lang.Cloneable interface.
  - Flag interfaces that indicates the object wants to cooperate in being cloned. The Cloneable interface does not actually contain any methods. (Why??)
  - If the object isn't Cloneable, the clone() method throws a CloneNotSupportedException exception.

Extension 2: Clone in java.lang.Object

- The clone() method is declared as "protected"
  - By default it can be called only by an object on itself, an object in the same package, or another object of the same type or a subtype.
- By convention, classes that implement this interface should override Object.clone() with a public method. (Why not be abstract??)

```
protected native Object clone()
                 throws CloneNotSupportedException;
```

The Java Native Interface (JNI) is a programming framework that allows Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages, such as C, C++ and assembly.

Extension 3: Returned type of clone()

- The syntax for calling clone in Java is:
  ```
  Object copy = obj.clone();
  ```
  or commonly
  ```
  MyClass copy = (MyClass) obj.clone();
  ```
  which provides the typecasting needed to assign the generic Object reference returned from clone to a reference to a MyClass object.
- One disadvantage with the design of the clone() method is that the return type of clone() is Object, and needs to be explicitly cast back into the appropriate type.
- However, overriding clone() to return the appropriate type is preferable and eliminates the need for casting in the client (since J2SE 5.0).
  - Covariance (协变) of the return type refers to a situation where the return type of the overriding method is changed to a type related to (but different from) the return type of the original overridden method, following the Liskov substitution principle.

Extension 4: Conditions of cloning

- For every object *x*, Object.clone() method satisfies：
  - x.clone().getClass() == x.getClass();
  - *x*.clone() != *x* (reference equity);
  - x.clone().equals(x) (if equals method is defined as valued equity)

Extension 5: clone() in interface

- Cannot access the clone() method on an abstract type (interface but not abstract class).

```
List list = new ArrayList();
list.clone(); //?????
```

- The only way to use the clone() method is if you know the actual class of an object; which is contrary to the abstraction principle of using the most generic type possible (DIP) .

Extension 6: clone() and the Singleton pattern

- Override the clone() method in a Singleton (Not in Java)

```
public Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

Extension 7: clone() and inherited hierarchy

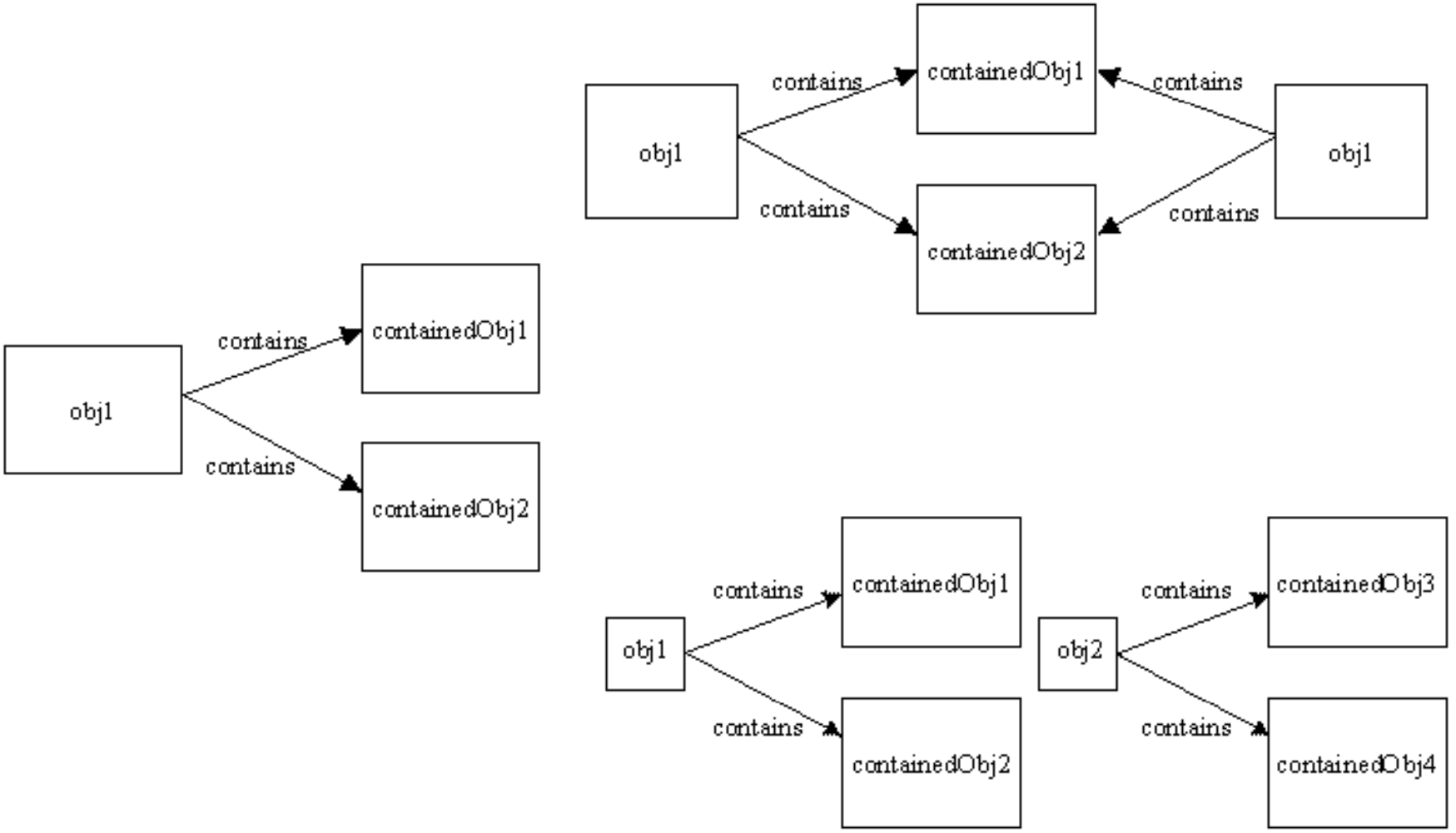
- Every type reference that needs to support the clone function, The type reference itself or one of its parents must
  - have a publicly accessible clone() method
  - Implements Cloneable.

```
interface I extends Cloneable{
}

abstract class X implements I{
    public X clone() throws CloneNotSupportedException {
        return (X)super.clone();
    }
}

abstract class Y extends X {
}

class Z extends Y {
}
```

```
I i = new Z();
X x = new Z();
Y y = new Z();
Z z = new Z();

X x0 = i.clone();
X x1 = x.clone();
X x2 = y.clone();
X x3 = z.clone();

Y y0 = i.clone();
Y y1 = x.clone();
Y y2 = y.clone();
Y y3 = z.clone();

Z z0 = i.clone();
Z z1 = x.clone();
Z z2 = y.clone();
Z z3 = z.clone();
System.out.println(z.getClass()==z.clone().getClass());
System.out.println("OK");
```

Extension 8: Shallow copy (clone) and Deep copy (clone)

- Shallow copy: A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the references are copied. (Object.clone())
- Deep copy: It is a complete duplicate copy of an object. A deep copy generates a copy not only of the primitive values of the original object, but copies of all sub-objects as well, all the way to the bottom, it is field-for-field copy .
- If you need a true, complete copy of the original object, then you will need to implement a full deep copy for the object.


obj1
contains
containedObj1
contains
containedObj2
obj1
contains
containedObj1
contains
obj1
contains
containedObj2
obj1
contains
containedObj1
contains
containedObj2
obj2
contains
containedObj3
contains
containedObj4

Extension 8: Deep copy

- If the object refers to other complex objects, which in turn refer to others, then this task can be daunting indeed.
  - With complex object graphs deep copying can become problematic, with recursive references. Once one object is cloneable, others tend to follow until the entire graph attempts to implement Cloneable. Sooner or later you run into a class that you can't make Cloneable.
- Traditionally, each class in the object must be individually implement the Cloneable interface and override its clone() method, in order to make a deep copy of itself as well as its contained objects.

```
class A implements Cloneable{
    private B b = new B();
    private List<C> cList = new ArrayList<C>();

    public A clone() throws CloneNotSupportedException {
        A a = (A)super.clone();
        a.b = this.b.clone();
        //a.cList = this.cList.clone();
        List<C> clonedList = new ArrayList<C>();
        for (C object : this.cList) {
            clonedList.add(object.clone());
        }
        a.cList = clonedList;
        return a;
    }
}
class B implements Cloneable{
    public B clone() throws CloneNotSupportedException {
        return (B)super.clone();
    }
}
class C implements Cloneable{
    public C clone() throws CloneNotSupportedException {
        return (C)super.clone();
    }
}
```

Extension 9: clone() and final fields

- Generally, deep clone() is incompatible with final fields.
  - clone() is essentially a default constructor (one that has no arguments), it is impossible to assign a final field within a clone() method;
  - A compiler error is the result.;
  - Changing the final field to immutable field.
- The only solution is to remove the final modifier from the field, giving up all the benefits it conferred.
- For this reason, many programmers prefer to clone the objects by Serialization.

Extension 10: Deep cloning through Serialization

- Serialization: Saving the current state of an object to a stream,
- Deserialization: Restoring an equivalent object from that stream.
  - The stream functions as a container for the object. Its contents include a partial representation of the object's internal structure, including variable types, names, and values.
  - The container may be transient (RAM-based) or persistent (disk-based). A transient container may be used to prepare an object for transmission from one computer to another.

Extension 10: Deep cloning through Serialization

- Serialization is used for reconstruct the object immediately.
  - Ensure that all classes in the object's graph (all fields) are serializable (implements Serializable).
  - Create input stream and output stream.
  - Use the input stream and output streams to create object input stream and object output stream.
  - Pass the object that you want to copy to the object output stream.
  - Read the new object from the object input stream and cast it back to the class of the object you sent.

```java
public abstract class SerialCloneable implements Cloneable, Serializable {

    private static final long serialVersionUID = SerialCloneable.class.hashCode();

    public Object clone() {
        try {
            ByteArrayOutputStream bout = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(bout);
            out.writeObject(this);
            out.close();

            ByteArrayInputStream bin = new ByteArrayInputStream(bout
                    .toByteArray());
            ObjectInputStream in = new ObjectInputStream(bin);
            Object ret = in.readObject();
            in.close();
            return ret;
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
```

Extension 10: Problems of Serialization

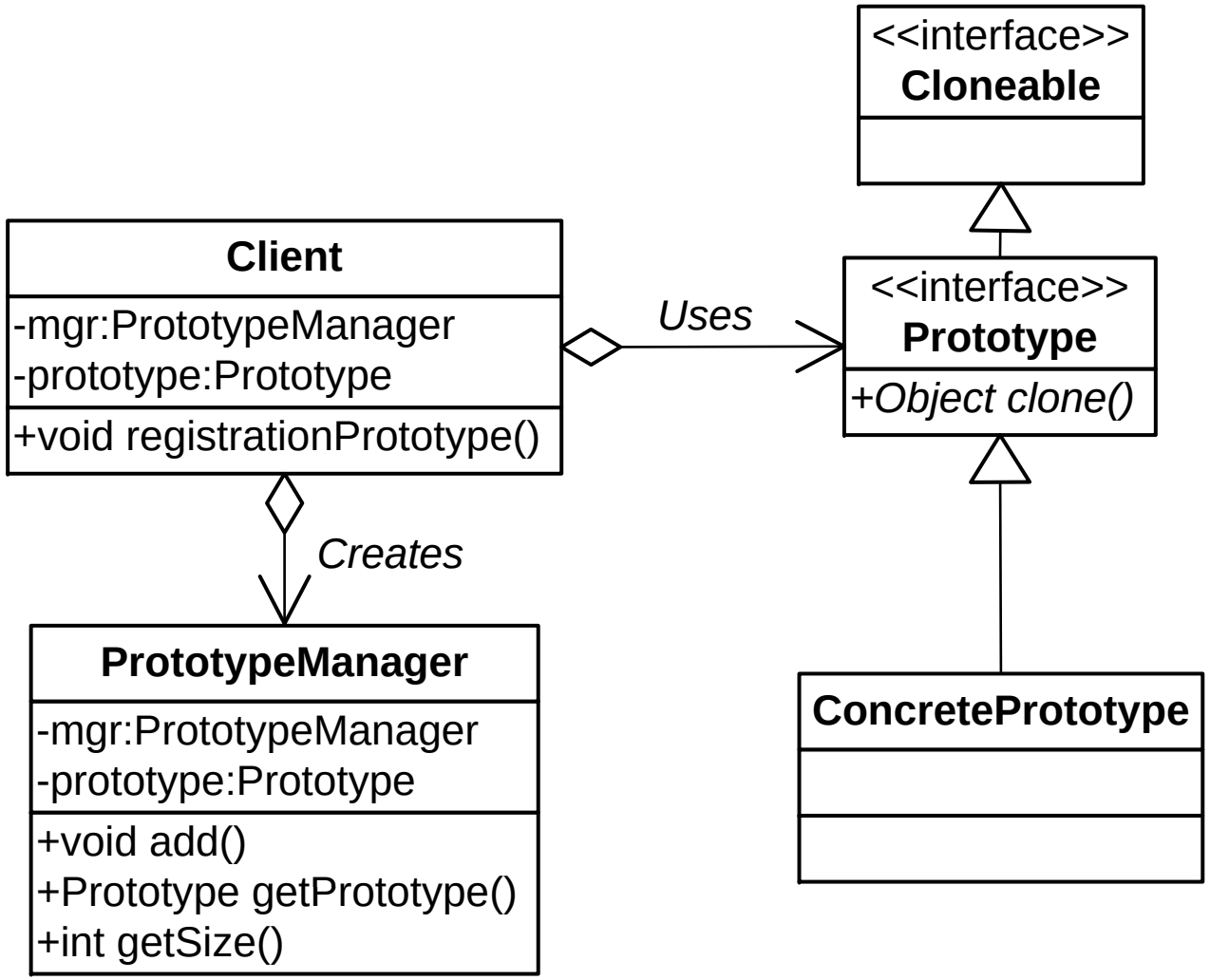
- Serialization is hugely expensive. It could easily be a hundred times more expensive than the clone() method.
- Not all objects are serializable.
- Making a class serializable is tricky and not all classes can be relied on to get it right.

Extension 10: Conditions of Serialization

- Some classes are not suitable for serialization:
  - Related to the native code or unmanaged resources;
  - The internal state of an instance is relay on the runtime context or Java virtual machine, such as Thread, InputStream，PrintJob，Connection;
  - May bring the potential security problems;
  - Singletons;
  - The state of an instance is changed frequently;
  - An utilized class which mainly contains static methods;

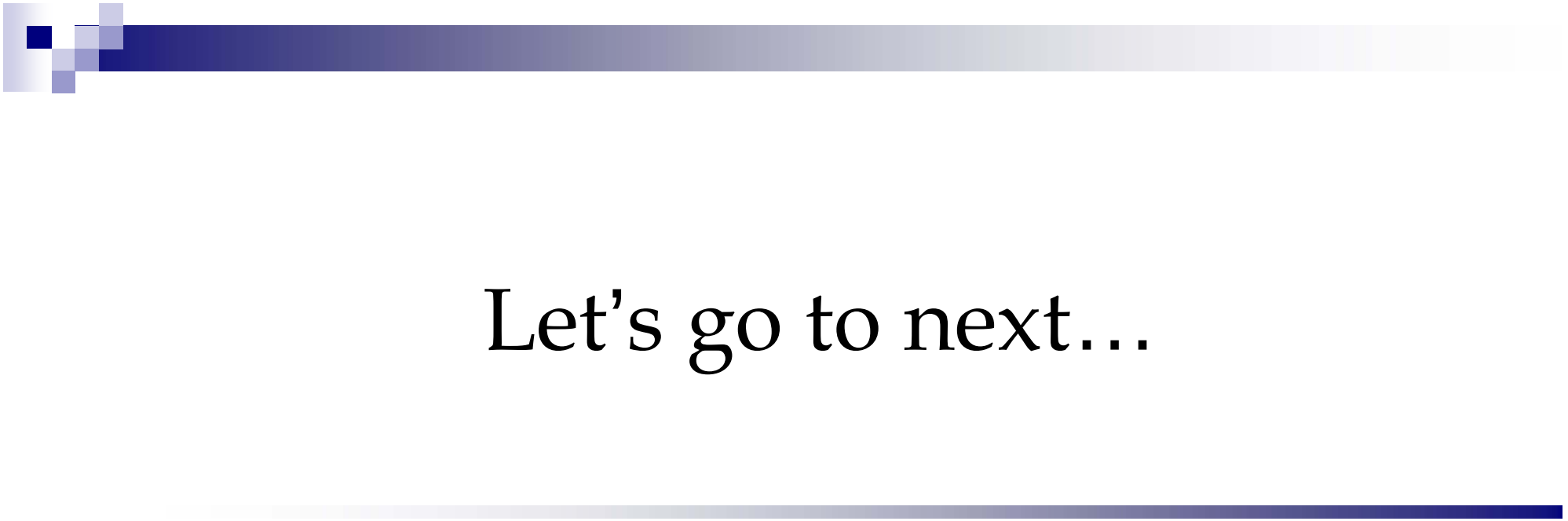
Variation 1: Using a prototype manager

- When the number of prototypes (NOT instances) in a system isn't fixed, keep a registry of available prototypes.
- Clients won't manage prototypes themselves but will store and retrieve them from the registry.(Factory Method + Prototype + Registry)
  - A prototype manager is an associative store that returns the prototype matching a given key.
  - A prototype manager has operations for registering a prototype under a key and for unregistering it.
  - A client will ask the registry for a prototype before cloning it.
  - Clients can change or even browse through the registry at run-time. This lets clients extend and take inventory on the system without writing code.


<<interface>>
Cloneable
Client
-mgr:PrototypeManager
-prototype:Prototype
+void registrationPrototype()
Uses
<<interface>>
Prototype
+Object clone()
Creates
PrototypeManager
-mgr:PrototypeManager
-prototype:Prototype
+void add()
+Prototype getPrototype()
+int getSize()
ConcretePrototype

Variation 2: Initializing clones

- Some time the new cloned instance is required to have different states from the prototype.
  - Initialize or reset some or all of its internal state to values of their choosing.
- Generally, you can't pass these values in the clone operation, because their number will vary between classes of prototypes.
- Passing parameters in the clone operation precludes a uniform cloning interface.
  - If prototype classes already define operations for (re)setting states. Clients may use these operations immediately after cloning.
  - If not, then you may have to introduce an initialize operation that takes initialization parameters as arguments and sets the clone's internal state accordingly.



Let’s go to next…