Tecniche di Progettazione: Design Patterns

GoF: Memento Prototype Visitor

Memento

Memento

- Intent
  - "Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later."
- Motivation
  - When we want to store off an object's internal state without adding any complication to the object's interface.
  - Perhaps for an undo mechanism

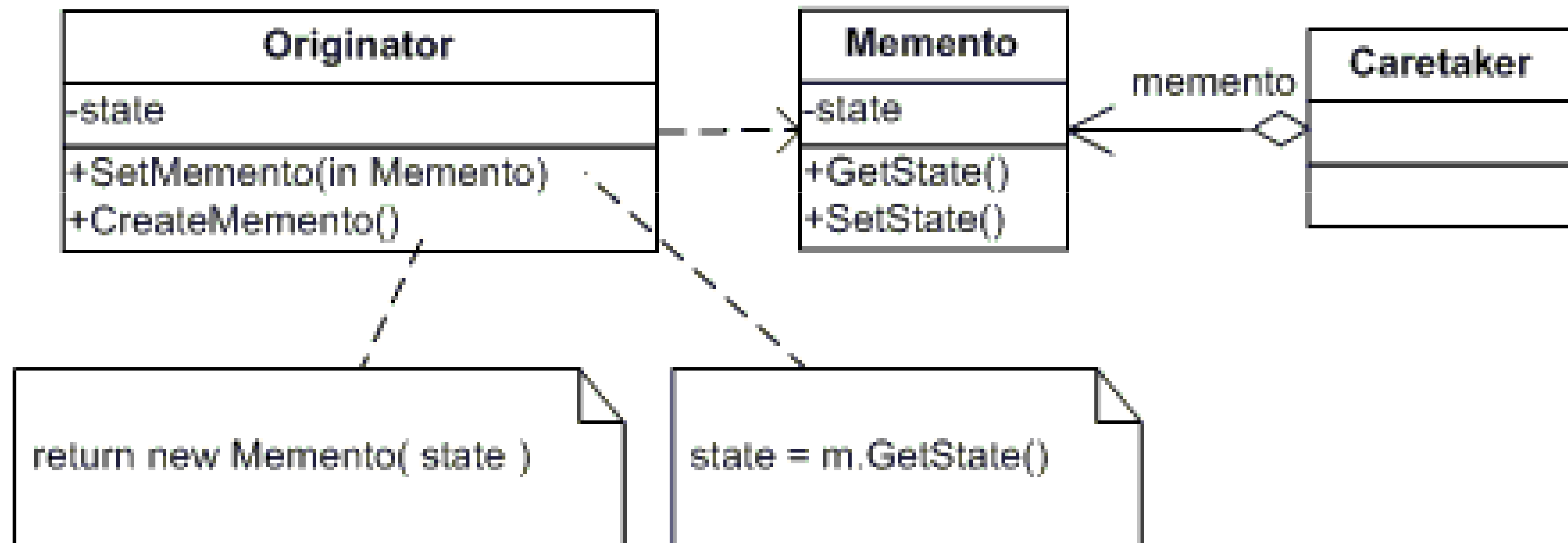
Memento pattern

- Memento:
  - a saved "snapshot" of the state of an object or objects for possible later use
  - useful for:
    - writing an Undo / Redo operation
    - ensuring consistent state in a network
    - Persistency: save / load state between executions of program

Applicability

- Use this
  - when you want to save state on a hierarchy's elements.
  - When the hierarchy's interface would be broken if implementation details were exposed.

Structure



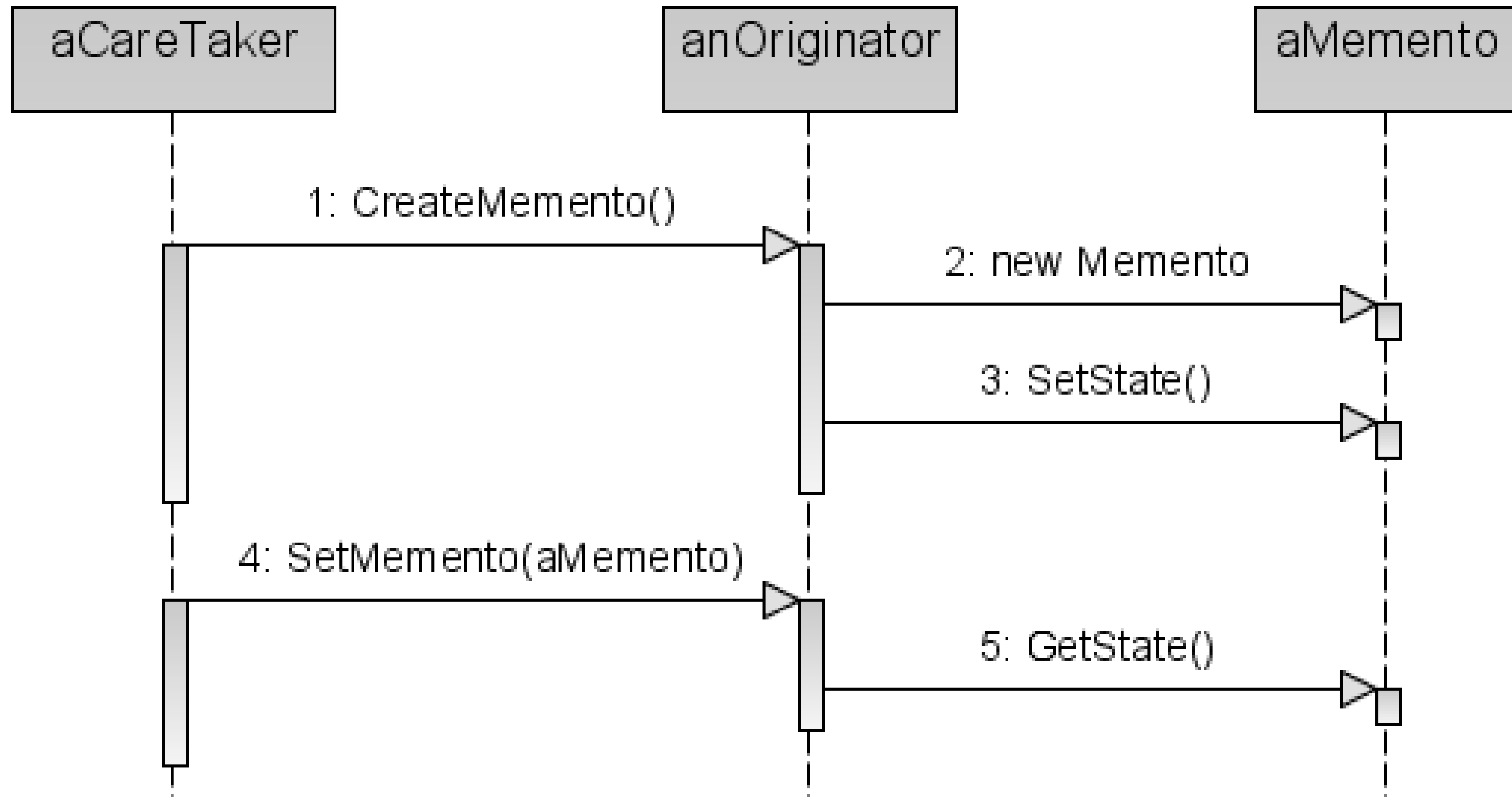
Participants

- Memento
  - stores the state of the Originator
- Originator
  - Creates the memento
  - “Uses the memento to restore its internal state”
- CareTaker
  - Keeps track of the Memento(s)
  - Never uses the Memento’s Interface to the Originator

Collaboration

- Caretaker requests a memento from an Originator.
- Originator passes back memento.
- Originator uses it to restore state.

Collaboration



Consequences (good)

- "Preserves Encapsulation Boundaries"
- "It simplifies Originator"

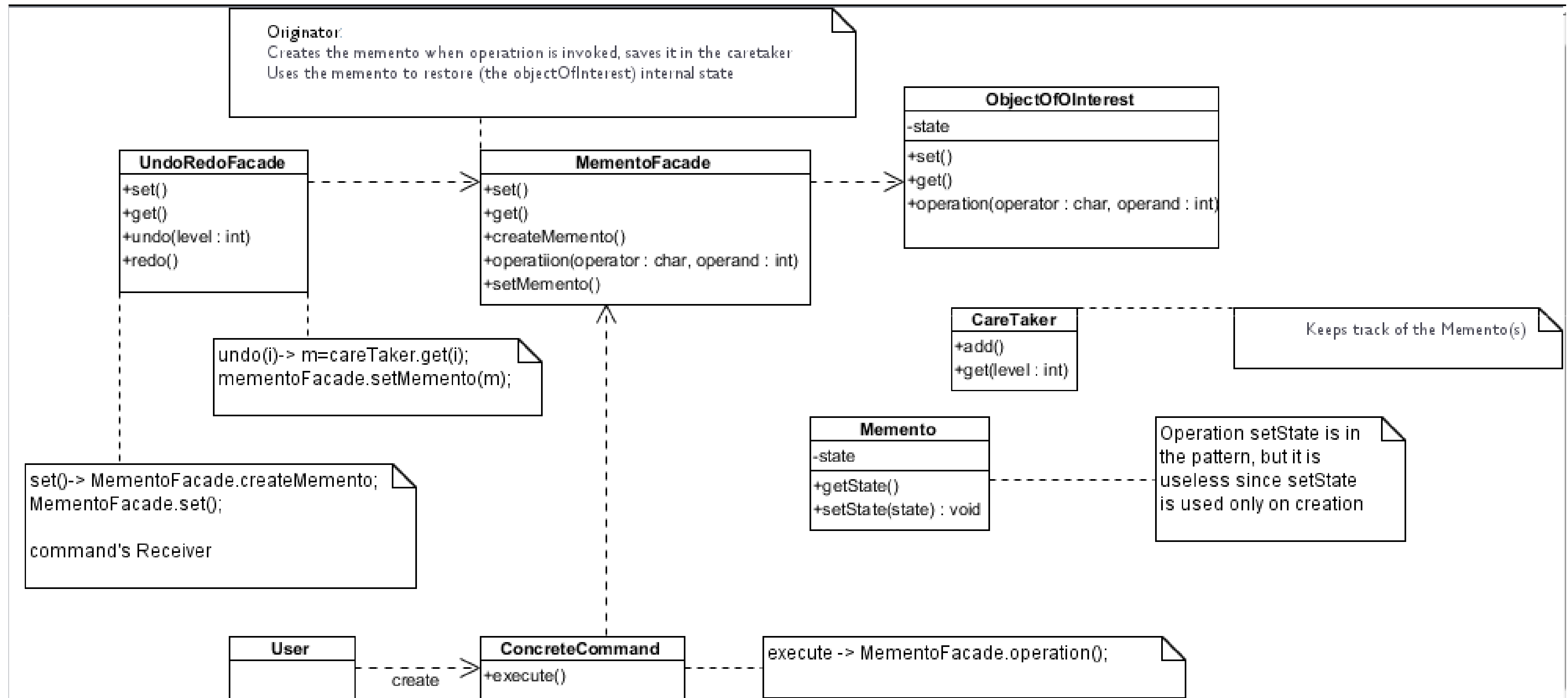
Consequences (bad)

- Might be expensive
- Difficulty defining interfaces to keep Originator encapsulated
- Hidden costs in caring for mementos
  - Caretaker could have to keep track of a lot of information for the memento

Storing Incremental Changes

- If storing state happens incrementally, then we can just record the changes of what's happened in a new memento object.
- This helps with memory difficulties.

Exercise: re-engineer the command-memento example



Holte Pratic

Prototype

Prototype Pattern

- A creational pattern
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Applicability

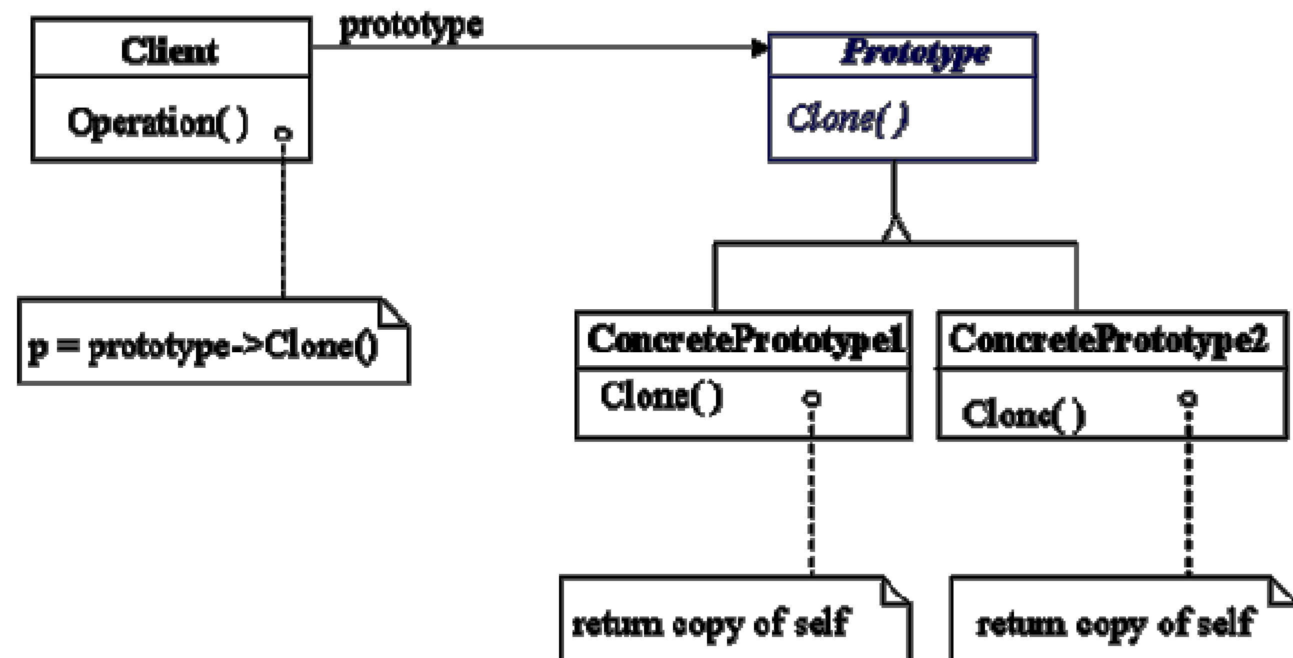
- when a system should be independent of how its products are created, composed, and represented and any of the following is the case:
  - when the classes to instantiate are *specified at run-time*
  - avoid building a class *hierarchy of factories that parallels the class hierarchy of products*
  - when instances of a class can have one of only a few different combinations of state.
    - It may be more convenient to have the proper number of prototypes and clone them rather than instantiate the class manually each time with appropriate state.

Structure & Participants

Prototype(Graphic)
-declares an interface for cloning itself.

ConcretePrototype (Staff, WholeNote, HalfNote)
-implements an operation for cloning itself.

Client(GraphicalTool)
- creates a new object by asking a prototype to clone itself.



Problem

Tool
Manipulate( )

Graphic

RotateTool
Manipulate( )

GraphicTool
Manipulate( )

Staff

MusicalNote

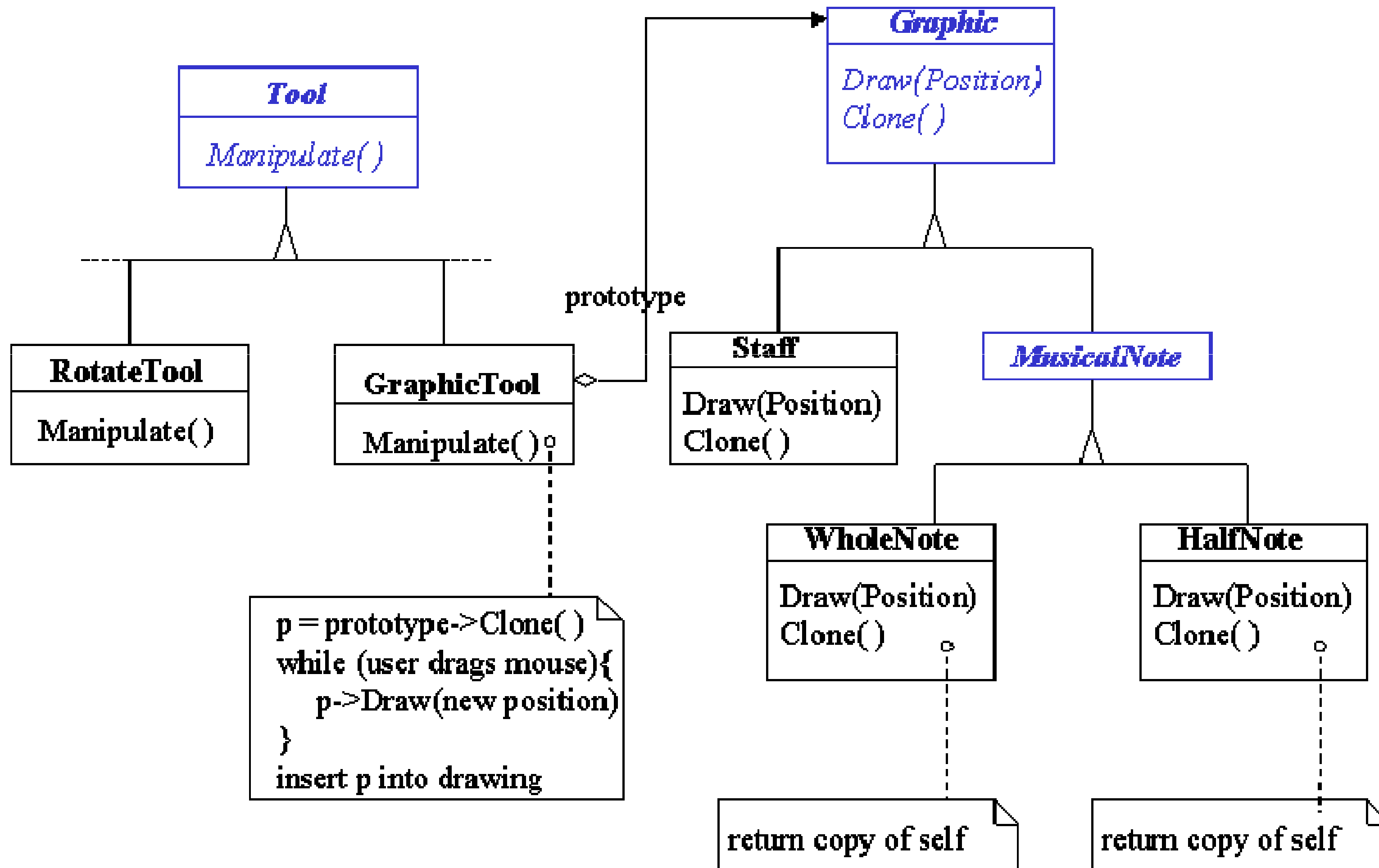
WholeNote

HalfNote

StaffGraphicTool
Manipulate( )

WholeNoteGraphicTool
Manipulate( )

Prototype solution



Benefits of Prototype Pattern

- Hides the complexities of making new instances from the client.
- Provides the option for the client to generate objects whose type is not known.
- In some circumstances, copying an object can be more efficient than creating a new object.

Implementation of Prototype Pattern

- It is built on the method .clone()

java.lang Class Object
protected Object **clone**() throws CloneNotSupportedException

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object x, the expression:

x.clone() != x

will be true, and that the expression:

x.clone().getClass() == x.getClass()

will be true, but these are not absolute requirements. While it is typically the case that:

x.clone().equals(x)

will be true, this is not an absolute requirement.

By convention, the returned object should be obtained by calling super.clone. If a class and all of its superclasses (except Object) obey this convention, it will be the case that x.clone().getClass() == x.getClass().

java.lang Class Object
protected Object **clone**() throws CloneNotSupportedException

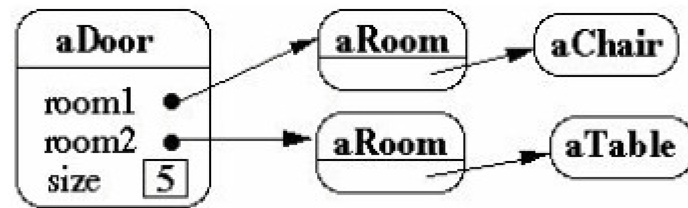
- By convention, the object returned by this method should be independent of this object (which is being cloned).
- To achieve this independence, it may be necessary to modify one or more fields of the object returned by super.clone before returning it.
  - Typically, this means copying any mutable objects that comprise the internal "deep structure" of the object being cloned and replacing the references to these objects with references to the copies.
  - If a class contains only primitive fields or references to immutable objects, then it is usually the case that no fields in the object returned by super.clone need to be modified.

Clone and deep/shallow copy

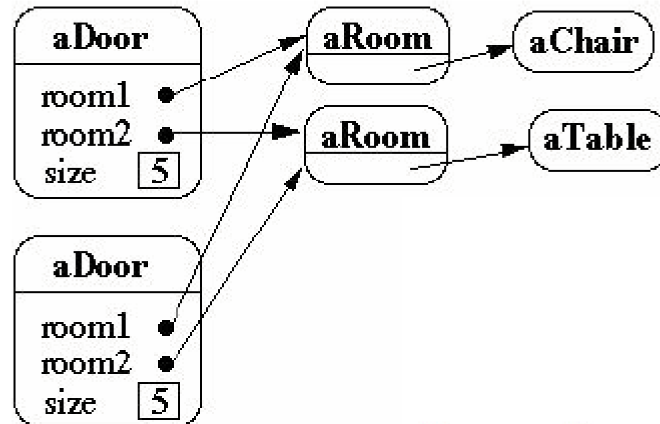
- Clone can be implemented either as a deep copy or a shallow copy:
  - In a deep copy, all objects are duplicated,
  - In a shallow copy, only the top-level objects are duplicated and the lower levels contain references.

Deep vs shallow copy

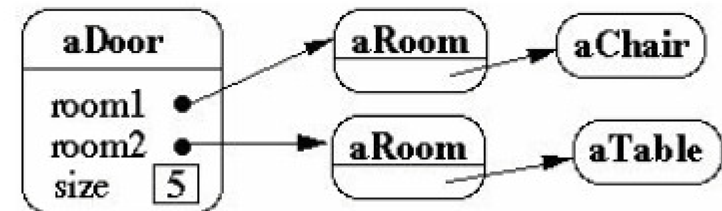
Original

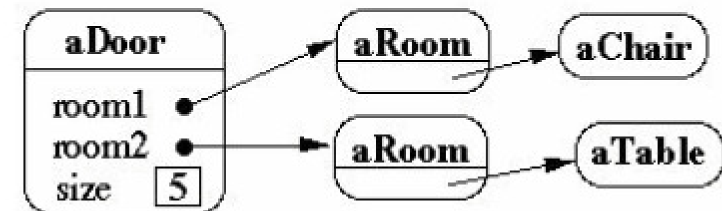


Shallow Copy



Deep Copy





java.lang
Class Object

- protected Object clone()
  - this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment
  - the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.

java.lang
Interface Cloneable

- A class implements the Cloneable interface to indicate to the Object.clone() method that it is legal for that method to make a field-for-field copy of instances of that class.
  - Invoking Object's clone method on an instance that does not implement the Cloneable interface results in the exception CloneNotSupportedException being thrown.
- By convention, classes that implement this interface should override Object.clone (which is protected) with a public method.
- Note that this interface does not contain the clone method.
  - Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface.

Point

```
class Point implements Cloneable{
    private int x;
    private int y;

    @Override
    public Point clone() {
        return (Point)super.clone();
    }
}
```

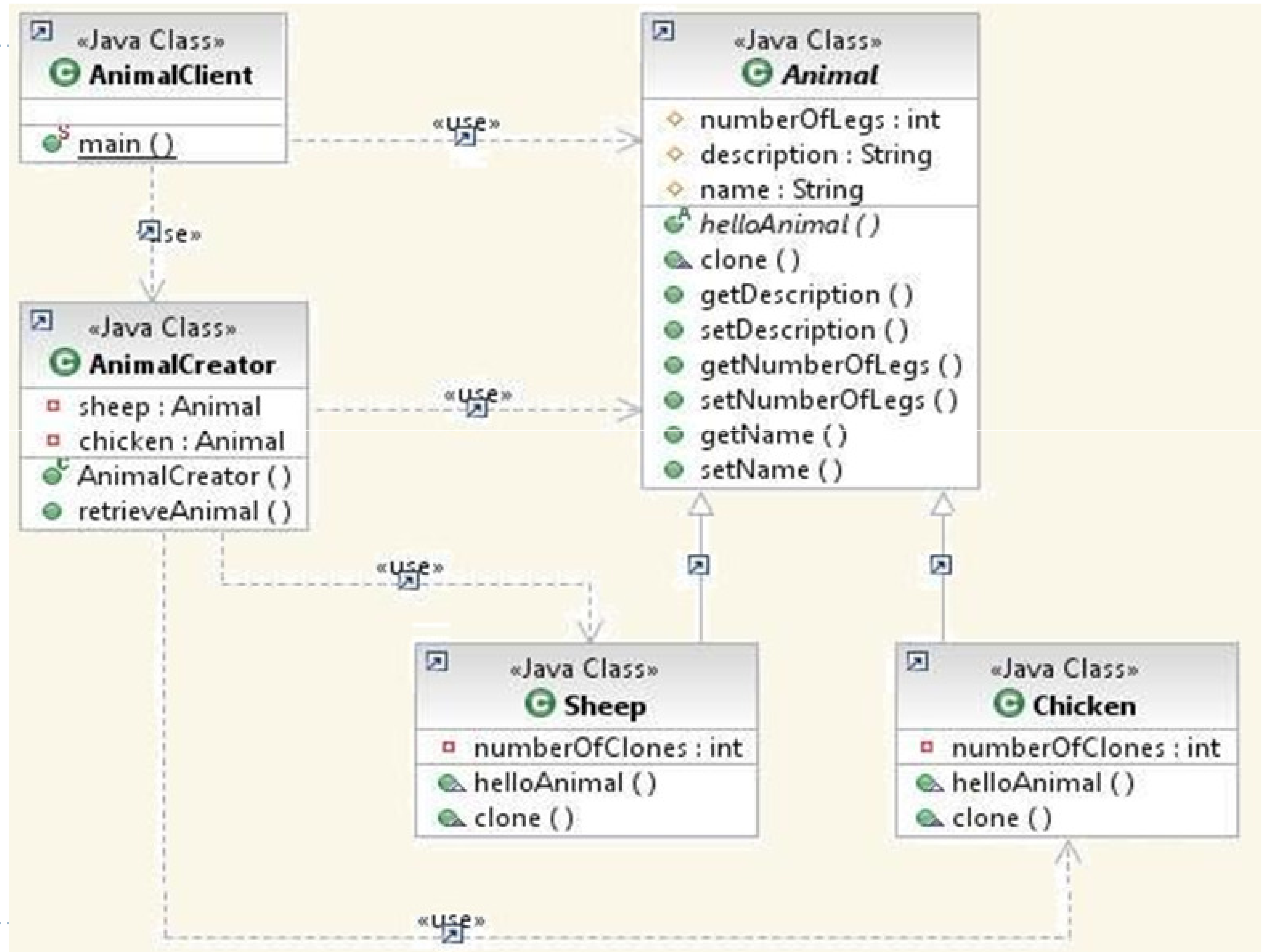
Line: shallow copy

```
class Line implements Cloneable {
    private Point start;
    private Point end;
    public Line() {
        //Careful: This will not happen for the cloned object
        SomeGlobalRegistry.register(this);
    }
    @Override
    public Line clone() {
        return (Line)super.clone();
    }
}
```

Line: deep copy.

```
@Override
public Line clone() {
    Line line = (Line)super.clone();
    //since Point is cloneable. Otherwise we will
    //have to instantiate and populate it's fields manually
    line.start = this.start.clone();
    line.end = this.end.clone;
    return line;
  }
```

Ex. Animal farm



Prototype Pattern Example code

```
public abstract class Animal implements Cloneable {
        protected int numberOfLegs = 0;
        protected String description = "";
        protected String name = "";

        public abstract String helloAnimal();

        public Animal clone() {
                Animal clonedAnimal = null;
                clonedAnimal = (Animal) super.clone();
                clonedAnimal.setName(name);
                return clonedAnimal;
        } // method clone

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
} // class Animal
```

Prototype Pattern Example code

```
public class Chicken extends Animal {
    private int numberOfClones = 0;

    public String helloAnimal() {
        StringBuffer chickenTalk = new StringBuffer();
        chickenTalk.append("Cluck cluck World. I am ");
        chickenTalk.append(name);
        return chickenTalk.toString();
     } // helloAnimal

     public Chicken clone() {
        Chicken clonedChicken = (Chicken) super.clone();
        String chickenName = clonedChicken.getName();
        numberOfClones++;
        clonedChicken.setName(chickenName + numberOfClones);
        return clonedChicken;
     } // method clone
}
```

Prototype Pattern Example code

```
public class Sheep extends Animal {
    private int numberOfClones = 0;

    public String helloAnimal() {
        StringBuffer sheepTalk = new StringBuffer();
        sheepTalk.append("Meeeeeee World. I am ");
        sheepTalk.append(name);
        return sheepTalk.toString();
    } // helloAnimal

    public Sheep clone() {
        Sheep clonedSheep = (Sheep) super.clone();
        String sheepName = clonedSheep.getName();
        numberOfClones++;
        clonedSheep.setName(sheepName + numberOfClones);
        return clonedSheep;
     } // method clone
}
```

Prototype Pattern Example code

```
public class AnimalCreator {
        private Animal sheep = new Sheep();
        private Animal chicken = new Chicken();

        public AnimalCreator() {
                sheep.setName("Sheep");
                chicken.setName("Chicken");
        } // no-arg constructor

        public Animal retrieveAnimal(String kindOfAnimal) {
                if ("Chicken".equals(kindOfAnimal)) {
                        return (Animal) chicken.clone();
                }
                else if ("Sheep".equals(kindOfAnimal)) {
                        return (Animal) sheep.clone();
                } // if
                return null;
        } // method retrieveAnimal
} // class AnimalCreator
```

Prototype Pattern Example code

```
public class AnimalClient {
    public static void main(String[] args) {
        AnimalCreator animalCreator = new AnimalCreator();
        Animal[] animalFarm = new Animal[8];

        animalFarm[0] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[1] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[2] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[3] = animalCreator.retrieveAnimal("Chicken");
        animalFarm[4] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[5] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[6] = animalCreator.retrieveAnimal("Sheep");
        animalFarm[7] = animalCreator.retrieveAnimal("Sheep");

        for (int i= 0; i<=7; i++) {
            System.out.println(animalFarm[i].helloAnimal());
        } // for
    } // main method
} // class AnimalClient
```

```
Cluck cluck World. I am Chicken1.
Cluck cluck World. I am Chicken2.
Cluck cluck World. I am Chicken3.
Cluck cluck World. I am Chicken4.
Meeeeeee World. I am Sheep1.
Meeeeeee World. I am Sheep2.
Meeeeeee World. I am Sheep3.
Meeeeeee World. I am Sheep4.
```