17 Prototype

The Prototype Pattern belongs to the category of generation patterns. You have a certain number of products that do not differ significantly. You create a prototype of a product. When variations are requested, you clone the prototype and vary it. Example: you offer different types of pizzas. To cope with the volume of orders, you make a prototype of a Pizza Margherita. When a Hawaiian pizza is ordered, you clone the prototype, top it with ham and pineapple, and serve the finished pizza. That actually explains the pattern. In this chapter, I present more details and address the question of how to clone objects in Java.

17.1 Cloning Objects

The Prototype Pattern is not really difficult to understand – the problem is more about how to clone an object. Take a look at the example project car factory. There we have the class `Motor` with the attributes `id` (of the block), `hp.` (horsepower), and (cubic) `capacity` and the respective access methods. The class `Car` holds a reference to an engine instance, Car stores the number of seats and finally the special edition of the car. The class also implements the interface `Cloneable` and overrides the method `clone()` of `Object`.

```
public class Auto implements Cloneable {
    private Engine engine;
    private Edition edition;
    private final int numberSeats;

    Auto(Engine engine, Edition edition, int numberSeats)
        {
            this.engine = engine;
```

O. Musch, *Design Patterns with Java*, https://doi.org/10.1007/978-3-658-39829-3_17

```
            this.edition = edition;
            this.numberSeats = numberSeats;
    }
    // … abridged

    @Override
    public Object clone()
                    throws CloneNotSupportedException {
        return super.clone();
    }
}
```

The main method of the test class demonstrates the use of the project. First, an engine is created and installed in the car designed as a prototype. The prototype is to be created in the TINY edition.

```
var engine = new Engine("General Motors", 100, 1.6);
var prototype = new Car(engine, Edition.TINY, 4);
System.out.println("Prototype: " + prototype);
var newCar = (car) prototype.clone();
System.out.println("New car: " + newCar + "\n");
```

The following text is output on the console:

```
Prototype: 4-seater passenger car, Edition TINY, Engine:
    General Motors with 100 hp and 1.6 liters capacity
New car: 4-seater passenger car, edition TINY, engine:
    General Motors with 100 hp and 1.6 liters capacity
```

So you created a clone from the prototype.

17.1.1 Criticism of the Implementation

The solution works, but it is not unproblematic. Look critically at the first approach.

17.1.1.1 The Cloneable Interface

The method `clone()` of the class `Object` has the visibility `protected`. It must therefore first be made `public` by a subclass. What is the role of the interface `Cloneable`? Please comment out the interface in the project as a test:

```
public class Auto {// implements Cloneable {
    // … abridged
    @Override
    public Object clone()
                    throws CloneNotSupportedException {
        return super.clone();
    }
}
```

The project continues to compile without errors. Now call the test method again. You will see that a `CloneNotSupported` exception is thrown. The default implementation of the `clone()` method first checks whether the object to be `cloned` is of type `Cloneable`. The interface is purely a marker interface – it does not impose a method. If the object to be cloned is not of this type, the default implementation throws the exception. If the object to be cloned is of type `Cloneable`, a bitwise copy of it is created and returned. The type of the clone is the same as the type of the original.

The documentation for the `Object.clone()` method recommends that the original object and the clone have the following relationships:

- The objects have different references `prototype != clone`
- The classes are identical: `prototype.getClass() == clone.getClass()`
- The objects are equal in the sense of `clone.equals(prototype) == true`

The main method of the test class checks these conditions.

```
System.out.println("Test for reference identity (==): "
                                + (newcar == prototype));
System.out.println("Class prototype: "
                                + prototype.getClass());
System.out.println("Class new car: "
                                + newCar.getClass());
System.out.println("Test for equality (equals): "
                                + newCar.equals(prototype));
```

The console outputs:

```
Test for reference identity (==): false
Class prototype: class car
Class new car: class car
Test for equality (equals): true
```

The default implementation of the equals method in the `Object` class checks for reference equality of the objects being compared `(prototype == clone)`. However, since prototype and clone have different references, if you override `clone()`, you will also override equals`()`. If you override `equals()`, you should also override `hashCode()`. The documentation for `Object.equals()` says:

> The equals method for class Object implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values x and y, this method returns true if and only if x and y refer to the same object (x == y has the value true).
>
> Note that it is generally necessary to override the hashCode method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

17.1.1.2 The Problem of Equal References

In the last code section, the main method returns the engine of the prototype and sets a different identifier there. Again, have the data of the vehicles output to the console.

```
motor = prototype.getEngine();
motor.setId("Marshall Motors");
System.out.println("Prototype: " + prototype);
System.out.println("New car: " + newCar);
```

The console now outputs (abbreviated):

```
Prototype: (…) Engine: Marshall Motors with 100 hp (…)
New car: (…) Engine: Marshall Motors with 100 hp (…)
```

A change that you have made to the prototype is reflected in the clone. You have to critically question whether this is intentional. All employees of a company – the prototypes and the clones – have the same employer, and if he is called "Dr. Z" today and "Dr. Q" tomorrow, this affects all employees. Here, a change to a referenced object of a prototype is certainly important for all clones. However, the reverse case is also conceivable: Dolly[1] and the prototype sheep may have the same DNA. However, it must never happen that you shear Dolly and the original sheep then also stands shaven on the meadow. In this case, changes to the clone or the prototype must have no effect on the other specimen. But first let's look at why the engine is the same on both cars, even though you only made the change to one.

[1] https://de.wikipedia.org/wiki/Dolly_(sheep).

17.1.1.3 What Happens During Cloning

I delegated the cloning to the default implementation of the clone() method in the example above:

```
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

This standard implementation copies the object to be cloned bit by bit. The contents of the variables are copied unchanged. What happens to a variable with a primitive data type? At the point where you create an int variable, for example, the value of the number is stored after an assignment. The variable simply makes it easier for you to access the memory location within the object. When you assign a new value to the variable, the memory location within the object is overwritten. This assignment has no effect on the clone, which has its own memory area. If you have a variable with a complex data type, a reference to the assigned object is stored in the location of the variable. The object is stored on the heap. And that reference is copied just like the value of the primitive variable. So two objects independently hold a reference to the same object. This is not critical if you are referencing immutable objects. If the clone and prototype reference the same String object, and you change that String on either the clone or prototype, a new String object is created and referenced there. The bitwise copy procedure is called shallow copy. When you clone arrays or collections, you always get a shallow copy. The Test_Collections sample project demonstrates the shallow copy using several examples.

This may be desired as in the joint employer example. But it may also be that you do not want to have two shorn sheep standing in the meadow. You can only put one engine in one car. So it makes sense to change the depth of the copy and create a deep copy. To do this, you recursively trace all referenced objects and create copies of them.

The class `Object` defines the protected method `clone()`. Subclasses make this method public; they override the method with `super.clone()`. The default implementation of the method in `Object` requests memory for the object's runtime type. The object is then copied one bit at a time. If the object to be cloned is not of type `Cloneable`, the default implementation throws an exception. You as the programmer are responsible for overriding the `clone()` method in a meaningful way. In particular, this also means deciding whether to create deep or shallow copies. The caller of the clone method you override cannot tell whether you are copying deep or shallow. You should therefore always document your procedure in the JavaDoc so that no uncertainties arise for the user with regard to the copying behavior.

17.1.2 Cloning in Inheritance Hierarchies

It is entirely up to you how you implement the clone method. The following solutions are therefore also conceivable. The clone method of the Car class could create a new Car object with the new operator. The number of seats and the reference to the enum are copied; a new engine (no clone here!) is created.

```
@Override
public Obect clone() throws CloneNotSupportedException {
    var newEngine = new Engine(“Bishop Motors“, 80, 1.4);
    var car = new Car(newEngine, Edition.TINY, 4);
    return car;
}
```

You can achieve a similar effect with a CopyConstructor. Here, too, a new car instance is created; the prototype then passes itself as an argument to the private constructor when cloned. You can find this code in the sample project CarFactory_CopyConstructor:

```
public class Car implements Cloneable {
    private Engine engine;
    private Edition edition;
    private final int numberSeats;

    Auto(Engine engine, Edition edition, int numberseats)
        {
        this.engine = engine;
        this.edition = edition;
        this.numberSeats = numberSeats;
    }

    private Car(Car car) {
        this.engine =
                    new Engine(“Bishop Motors“, 80, 1.4);
        this.numberSeats = auto.numberSeats;
        this.edition = auto.edition;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return new Auto(this);
    }
    // … abridged
}
```

Please look at these two approaches very skeptically they are not without problems. Assume that at some point you will have to deal with inheritance – car should be superclass for roadster, for example. How are the sub classes cloned? Roadster might try to request a clone with `super.clone()` - just as it did in the previous version. The clone method of Car would return an object of dynamic type Car, and that cannot be cast to a subtype, say Roadster; an appropriate cast would throw a ClassCastException.

It would be conceivable to query the dynamic type and call the corresponding constructor:

```
@Override
public Object clone() throws CloneNotSupportedException {
    if(car.getClass() == roadster.getClass())
        return new Roadster(this);
    else
        return new Car(this);
}
```

Each new subclass of Auto would get a new if branch – that's horror! But you have to make sure that the clone matches the type of the subclass, because a clone should satisfy the condition `prototype.getClass() == clone.getClass()`. Thus, if a class may have subclasses, so it is not final, it must implement the clone method differently. Since a superclass can never know all subclasses, it must not be its job to determine the runtime type of the object to be cloned. Only the default implementation of Object. clone() can take the runtime type into account. So a class that is not final should request the object to be returned with super.clone().

It is possible, of course, that a superclass must modify private data fields before returning the clone. In the example project CarFactory_Superclass, the Roadster class inherits from Car. In addition, you want the Roadster to get a new engine of its own when cloned. How do you achieve this goal? In this project version, I have removed the setEngine method. So a new car can only get a new engine in three ways: Either the constructor is parameterized appropriately, another car object changes the motor, or the object changes its motor itself. You won't call a constructor. So there must be another solution. Now take a look at the example project CarFactory_Superclass .

The client calls the clone method of the Roadster class, which in turn calls the clone method of the superclass. In the Roadster class you will find:

```
@Override
public Object clone() throws CloneNotSupportedException {
    return super.clone();
}
```

And in the car class then:

```
@Override
public Object clone() throws CloneNotSupportedException {
    var clone = super.clone();
    var carClone = (Car) clone;
    var newEngine = new Engine("Roadster Star", 80, 1.4);
    carClone.engine = newEngine;
    return carClone;
}
```

The clone method of Car first calls the clone method of Object. It gets back an object of dynamic type Roadster – you remember: Object.clone() returns the runtime type. This object is cast to type Car; this is allowed by Liskov's substitution principle – a subclass should be able to replace its base class. Now the Car instance has access to the private data field engine and assigns a new engine. After that, the cloned object is returned. The client receives an object of dynamic type Roadster with a brand new engine.

How is this done in practice? In practice, it makes sense for classes from which other classes are derived to override the clone method only protected and not implement the interface Cloneable. Subclasses are then free to support cloning or not.

Previously, the clone method was implemented according to the signature of the Object class: The return value is of type Object; also, the CloneNotSupportedException is propagated. Let's look at these two points in more detail.

When a method returns an object of a particular type, the client must first cast it to the expected type. Since Java 5, covariant return types are allowed; thus, the clone method is allowed to return a Car-object, as the following code snippet shows.

```
@Override
public Car clone() throws CloneNotSupportedException {
    var clone = super.clone();
    var carClone = (Car) clone;
    var newEngine = new Engine("Roadster Star", 80, 1.4);
    carClone.Engine = newEngine;
    return carClone;
}
```

However, if you make this change in the Car class, you must also do so in the Roadster subclass. There, too, the return value must be of type `Car`, not of type `Roadster`.

The other issue, the CloneNotSupportedException is passionately argued about in the forums; the prevailing opinion in the literature is not happy about this exception. On the one hand, there is a method `clone()` that is overridden by the class; on the other hand, this very method says that it may not support cloning at all. The CloneNotSupportedException is thrown when the default implementation of the `clone()` method determines that the object to be cloned is not of type `Cloneable`. However, this case is only relevant if none of the superclasses in an inheritance hierarchy implements the interface `Cloneable`.

However, this error can be detected and cured at compile time; it is therefore disproportionate to propagate a checked exception. For practical purposes, it is therefore a good idea to catch the exception within the clone method and throw an AssertionError instead:

```
public Auto clone() {
    try {
        var clone = super.clone();
        var carClone = (Car) clone;
        var newEngine =
                new Engine("Roadster Star", 80, 1.4);
        carClone.Engine = newEngine;
        return carClone;
    } catch (CloneNotSupportedException exc) {
        throw new AssertionError();
    }
}
```

In the following section, you design a larger project that builds on prototypes.

17.2 A Major Project

You draw different colored circles and connect them with lines. Both circles and lines are graphics. You can delete both circles and lines. Lines that no longer connect two graphics after deletion are also deleted. Figure 17.1 shows the program in action.

17.2.1 Discussion of the First Version

Right-click to open the context menu and choose to either draw a line or add a circle or select a graphic. You have different circles at your disposal: red, blue, green and so on. If you choose to draw a line, click either a circle or a line and drag a line to the target graphic, which can also be either a line or a circle. If you select Select a graphic, you can select either a circle or a line. Once you have made a selection, choose "Edit…" and "Delete selected" from the main menu. The selected graphic will then be deleted, with all affected connections – lines – also being deleted. If you have selected a circle, you can also move it.

You can model the highway network of Germany, the metro map of Paris or whatever with this version of the graphics editor. In further expansion stages, you could add boxes in addition to circles and turn the framework into a small UML editor. Before I show you the essential classes of the sample project GraphEditor_1, it's best to open it with NetBeans and familiarize yourself with the handling.

The program is started with the main method in the `ApplStart` class. This method creates a JFrame and adds the main menu and the drawing area, an instance of the

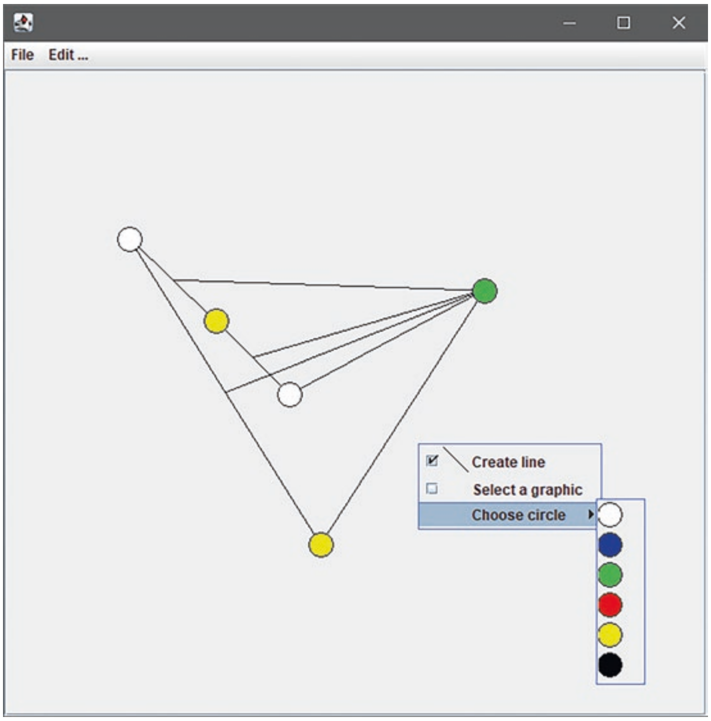

Fig. 17.1 Example program GraphEditor

PanelCanvas class, there. The PrototypeManager class creates the prototypes of the circles to be drawn. In practice, this class can grow to any size; extensive initialization routines to create the prototypes are offloaded there. The class Diagram is the data model of the editor. Here, the graphics are stored and managed in a list. The interface GraphicIF defines methods that must be implemented by all graphics (circles and lines). The interface RelationIF, which is derived from this, describes methods that only have to be implemented by lines. The Circle and Relation classes implement these interfaces and define the prescribed methods. For example, each graphic must be able to describe the rectangle enclosing it and store a specific color and its position. Perhaps most importantly, a graphic must be able to draw itself and, most importantly, clone itself. Lines always connect two other graphics, so they must also be able to store and return the start and end graphics.

The diagram is drawn on the PanelCanvas. The MouseAdapter overrides some EventHandlers. The PanelCanvas also defines some actions: The Create_Line_Action sets a flag that a line should be drawn. The Select_Action specifies that a graphic can be selected. The actions are passed to JCheckBoxMenuItem instances and hooked into the panel's context menu. The actions are combined into a ButtonGroup so that only one can be selected at a time.

```
private final ButtonGroup group = new ButtonGroup();
// … abridged
group.add(mnCreateLine);
group.add(mnSelect);
```

In the constructor, the prototype circles are also created as menu items. First, the prototype manager queries all prototypes. Then a separate menu item is created iteratively for each prototype; when called, the prototype is passed to the data field `nextGraphic`.

```
public PanelCanvas(Diagram) {
    // … abridged

    for (final GraphicIF tempGraphic :
                    PrototypeManager.getPrototypes())
        addPrototype(tempGraphic);
        // … abridged
}
    // … abridged

private void addPrototype(final GraphicIF prototype) {
    final var drawAction = new AbstractAction() {
        @Override
        public void actionPerformed(ActionEvent event) {
            createLine = false;
            nextGraphic = prototype;
        }
    };

    var mnNewGraphic = new JCheckBoxMenuItem(drawAction);
    mnCirclePrototypes.add(mnNewGraphic);
    group.add(mnNewGraphic);
    var icon = prototype.getIcon();
    mnNewGraphic.setIconTextGap(0);
    mnNewGraphic.setIcon(icon);
}
```

If you have selected a circle and click on the drawing area, the event handler `mouse-Clicked` is called, which requests the prototype from the data field `nextGraphic`, clones it and saves the clone in the diagram.

```
@Override
public void mouseClicked(MouseEvent event) {
    mousePosition = event.getPoint();
```

```
        if (nextGraphic != null)
        try {
            var newGraphic = (GraphicIF) nextGraphic.clone();
            selected = newGraphic;
            diagram.add(newGraphic, mousePosition);
        } catch (CloneNotSupportedException ex) {
            new ErrorDialog(ex);
        }
        else
            // Search for a graphic object
            // at the mouse position and select
            selected = diagram.findGraphic(mousePosition);
        repaint();
    }
```

These are the essential points of the program. Everything else are gimmicks that facilitate the handling of the program. Please analyze the code of the program independently.

17.2.2 The Second Version – Deep Copy

In the second version (sample project GraphEditor_2) it should be possible to clone the whole diagram. Two canvas instances are placed on the JFrame. If you click on Clone Diagram in the File menu item, the diagram is copied from the left side to the right side. You can now modify both diagrams independently. Figure 17.2 shows you the interface of the new version.

The menu item is defined in the `ApplStart` class and hooked into the menu. It queries for the diagram at the left panel and inserts it at the right panel. The `PanelCanvas` class defines the `getDiagramAsClone()` method in this version. First, the `ByteArrayOutputStream` `baos` is created and passed to the `ObjectOutputStream` `oos`. Into the baos the diagram is serialized. Then you create the `ByteArrayInputStream` `bais`. The data of the baos is passed to this. The bais is passed to the `ObjectInputStream` `ois`, which deserializes the serialized diagram. The resulting object is cast to a diagram and returned. Serialization and deserialization cause the objects, including the objects they reference, to be independent of each other.

```
    public Diagram getDiagramAsClone() {
        Diagram clone = null;
        try {
            this.nextGraphic = null;
            this.selected = null;
            this.createLine = false;
            ObjectOutputStream oos;
```

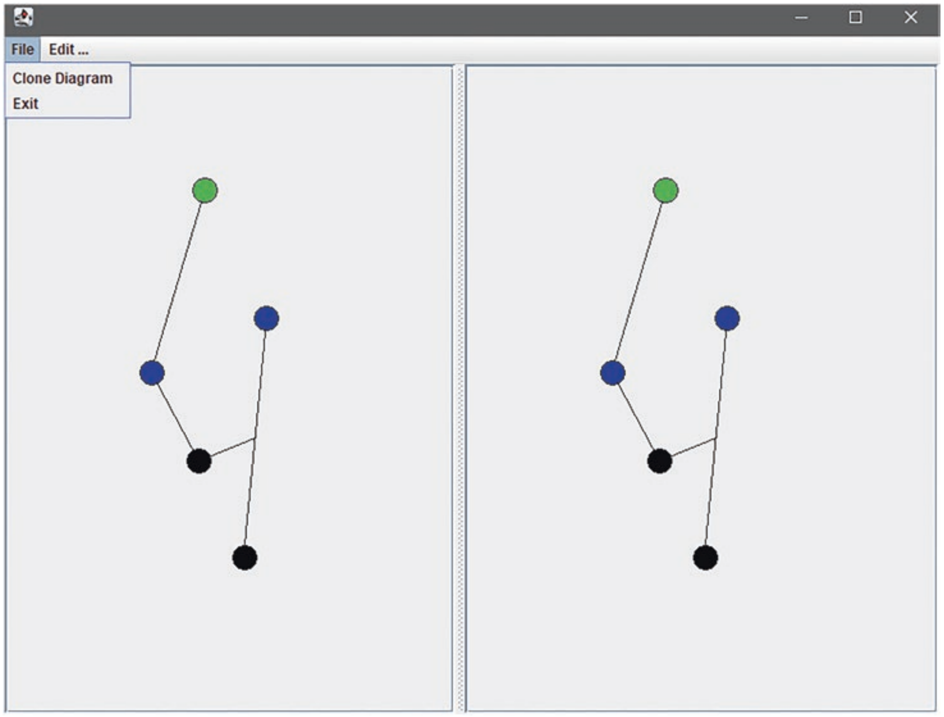

Fig. 17.2 Cloned diagram

```
            ByteArrayInputStream bais;
            ObjectInputStream ois;
            try ( var baos = new ByteArrayOutputStream()) {
                oos = new ObjectOutputStream(baos);
                oos.writeObject(diagram);
               bais =
                new ByteArrayInputStream(baos.toByteArray());
                ois = new ObjectInputStream(bais);
                clone = (diagram) ois.readObject();
            }
            oos.close();
            bais.close();
            ois.close();
        } catch (IOException ex) {
            new ErrorDialog(ex);
        } finally {
            return clone;
        }
    }
```

In the latest version of the program you will design your own prototypes.

17.2.3 Defining Your Own Prototypes

A special feature of the Prototype Pattern is that you can develop and add your own prototypes at runtime. The example project GraphEditor_3 is based on the first project version. So you already know most of the functionality. There is a new menu item New Prototype in the Edit menu. When you select this menu item, a color selection dialog appears. Select a color and click Ok. A circle with the desired color is now available as a prototype in the context menu. In the class `ApplStart` the action `newPrototypeAction` is defined, which calls the method `createPrototype()` on the drawing area. Within the method a JColorChooser is called. With the return value a new circle is created and added as prototype to the context menu as well as to the prototype manager.

```
public void createPrototype() {
    var newColor =
            JColorChooser.showDialog(PanelCanvas.this,
            "New Circle", Color.cyan);
    if (newColor != null) {
        GraphicIF newPrototype = new Circle(newColor);
        addPrototype(newPrototype);
        PrototypeManager.add(newPrototype);
    }
}
```

17.3 Prototype – The UML Diagram

From the example project GraphEditor_3 you can see in Fig. 17.3 first the UML diagram of the packages graphics and prototype, and then in Fig. 17.4 the UML diagram of the package delegate and the actual application.

17.4 Summary

Go through the chapter again in key words:

- The prototype pattern hides object creation from the client.
- You want to have different objects (products) at runtime.
- A prototype of each product is created, which is cloned when a new object is requested.
- The prototypes are managed by the Prototype Manager.

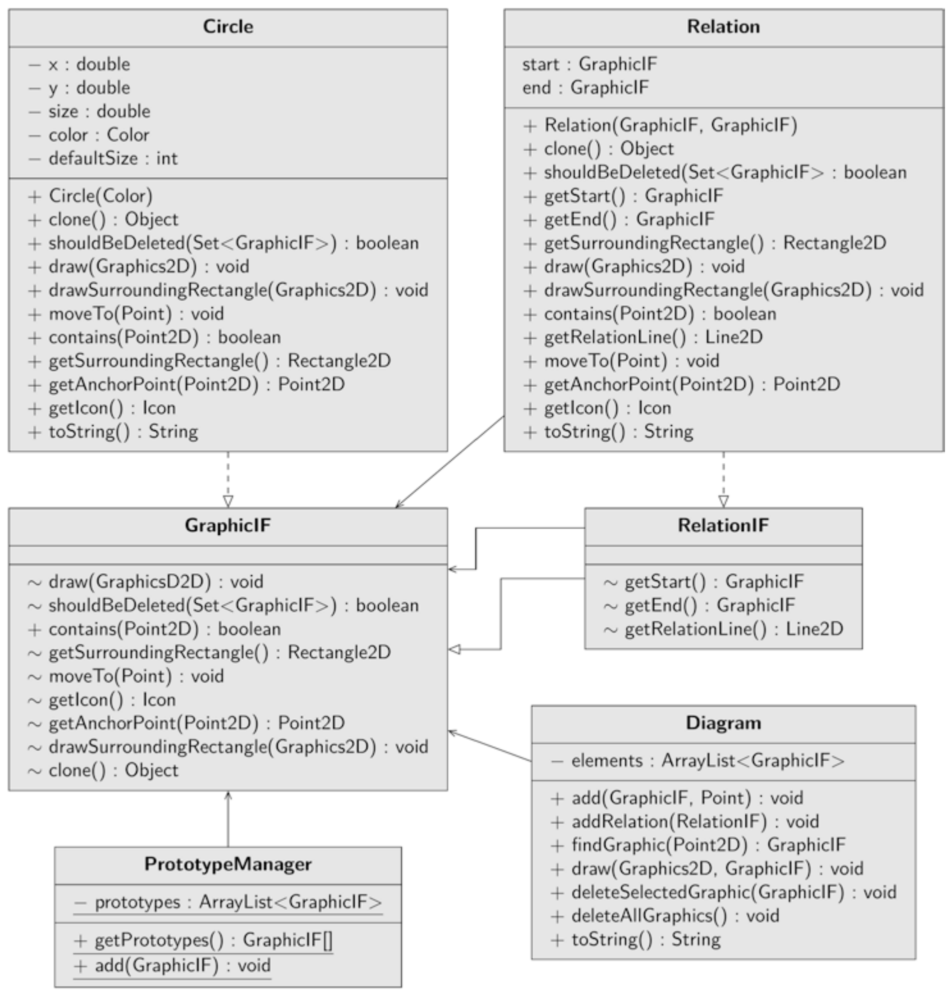
Circle
– x : double
– y : double
– size : double
– color : Color
– defaultSize : int
+ Circle(Color)
+ clone() : Object
+ shouldBeDeleted(Set<GraphicIF>) : boolean
+ draw(Graphics2D) : void
+ drawSurroundingRectangle(Graphics2D) : void
+ moveTo(Point) : void
+ contains(Point2D) : boolean
+ getSurroundingRectangle() : Rectangle2D
+ getAnchorPoint(Point2D) : Point2D
+ getIcon() : Icon
+ toString() : String
Relation
start : GraphicIF
end : GraphicIF
+ Relation(GraphicIF, GraphicIF)
+ clone() : Object
+ shouldBeDeleted(Set<GraphicIF> : boolean
+ getStart() : GraphicIF
+ getEnd() : GraphicIF
+ getSurroundingRectangle() : Rectangle2D
+ draw(Graphics2D) : void
+ drawSurroundingRectangle(Graphics2D) : void
+ contains(Point2D) : boolean
+ getRelationLine() : Line2D
+ moveTo(Point) : void
+ getAnchorPoint(Point2D) : Point2D
+ getIcon() : Icon
+ toString() : String
GraphicIF
~ draw(GraphicsD2D) : void
~ shouldBeDeleted(Set<GraphicIF>) : boolean
+ contains(Point2D) : boolean
~ getSurroundingRectangle() : Rectangle2D
~ moveTo(Point) : void
~ getIcon() : Icon
~ getAnchorPoint(Point2D) : Point2D
~ drawSurroundingRectangle(Graphics2D) : void
~ clone() : Object
RelationIF
~ getStart() : GraphicIF
~ getEnd() : GraphicIF
~ getRelationLine() : Line2D
Diagram
– elements : ArrayList<GraphicIF>
+ add(GraphicIF, Point) : void
+ addRelation(RelationIF) : void
+ findGraphic(Point2D) : GraphicIF
+ draw(Graphics2D, GraphicIF) : void
+ deleteSelectedGraphic(GraphicIF) : void
+ deleteAllGraphics() : void
+ toString() : String
PrototypeManager
– prototypes : ArrayList<GraphicIF>
+ getPrototypes() : GraphicIF[]
+ add(GraphicIF) : void

Fig. 17.3 UML diagram of the prototype pattern (sample project GraphEditor_3, packages graphics and prototype)

- Each prototype must override the `clone()` method when using the cloneable interface.
- The method `clone()` is defined in `Object` with the visibility `protected`.
- The default implementation of this method results in a flat copy of the object; referenced objects are not cloned.
- Collections also clone as a flat copy by default.
- Superclasses should override `clone()` at least in a `protected` manner to allow subclasses to be implemented in a meaningful way.

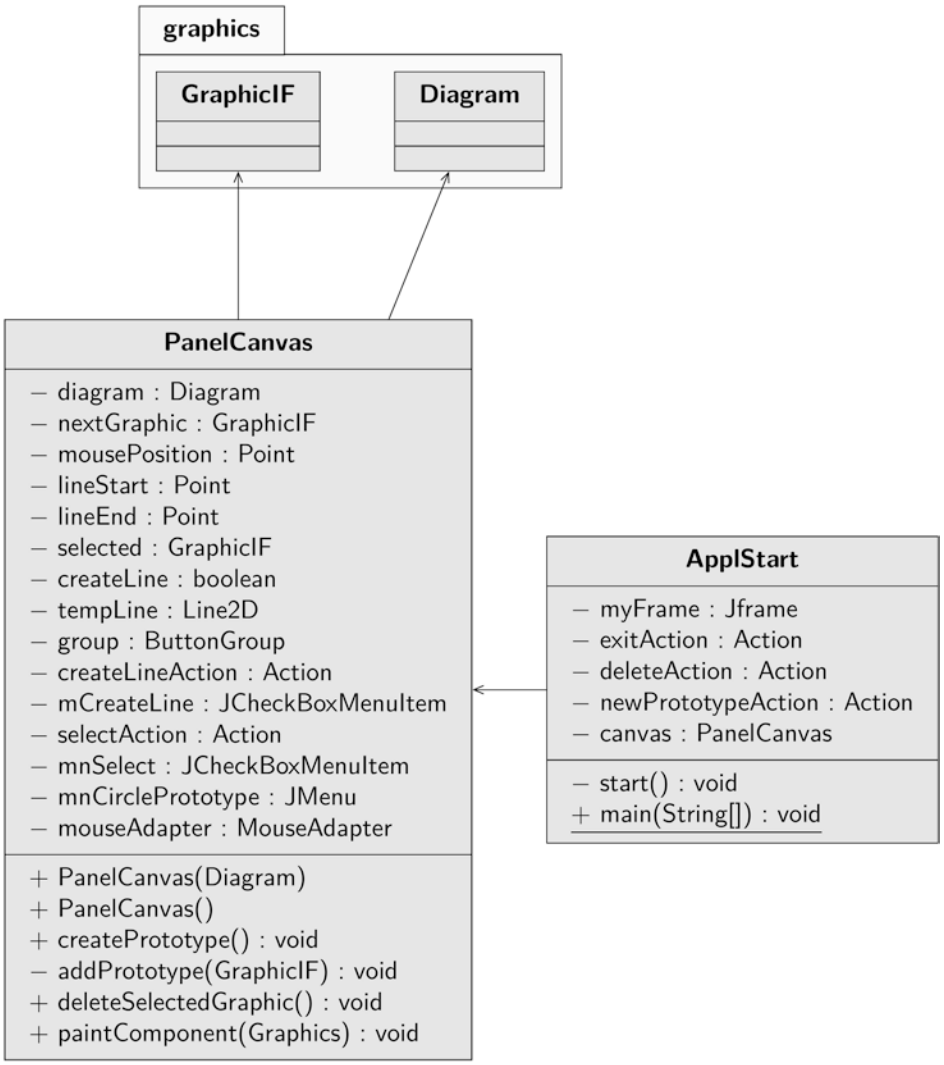

Fig. 17.4 UML diagram of the Prototype Pattern (example project GraphEditor_3, package delegate and application)

17.5 Description of Purpose

The Gang of Four describes the purpose of the "Prototype" pattern as follows:

> Determine the types of objects to create by using a prototypical copy, and create new objects by copying that prototype.