IGL s1 2021-2022

Maksym Lytvynenko

Exercice A

1. a) Comment on the instructions in listing 3:

This code demonstrates polymorphism in action. The same variable f of type Forme is used to reference different shape objects (Circle and Line) and calls their specific implementations of drawDescription().

b) Explain why the drawDescription() method of Forme can be considered a higher-order function:

The drawDescription() method in Forme is a higher-order function because:

- It uses the abstract method getDescription() which it doesn't implement itself
- It delegates the actual description generation to its subclasses

c) This question asks about parameterization and dynamic binding:

- Type parameterization: The static type is Forme
- Parameter(s): None in the method call
- Argument(s): The implicit this reference
- Dynamic binding: Occurs when getDescription() is called inside drawDescription()
    - For the first call: Cercle's getDescription() is called
    - For the second call: Ligne's getDescription() is called

2.
```
public class Ligne extends Forme {
private Point start; // Starting point
private Point end; // Ending point
public Ligne(Point start, Point end) {
        this.start = start;
         this.end = end; }
@Override
public String getDescription() {
 // Must match format: "une ligne de 5.0@2.0 a 5.0@5.0"
 return "une ligne de " + start + " a " + end; } }
```

3. For the JUnit tests organization:
We should describe how to organize tests for a framework like FramPict:

- Create test classes for each main component:
    - DessinTest for testing the Dessin class functionality
    - FormeTest for testing common Shape behaviors
    - Individual test classes for each shape type (LigneTest, CercleTest, etc.)
- Test cases should cover:
    - Basic object creation and initialization

- Shape description generation
- Drawing functionality
- Collection management (adding/removing shapes)
- Edge cases and error conditions
- Integration tests for complex drawings

Exercice B

1. Give the Java code for redefining the equals method on your Ligne class:

```
public class Ligne extends Forme {
    private Point start;
    private Point end;
    @Override
    public boolean equals(Object o) {
        // First check if it's the same object reference
        if (this == o) return true;
         // Check if null or different class
        if (o == null || getClass() != o.getClass()) return false;
        // Cast to Ligne since we know it's the same class
        Ligne other = (Ligne) o;
        // Compare the points
        return start.equals(other.start) && end.equals(other.end) ||
                start.equals(other.end) && end.equals(other.start); } }
```

2. Comment on your redefinition from the previous question, particularly using relevant terms associated with Liskov's substitution rules (sometimes called specialization rules):

For the Liskov Substitution Principle (LSP) comment: Our equals implementation respects LSP because:

- It maintains the contract of Object.equals()
- It's reflexive (x.equals(x) is true)
- It's symmetric (if x.equals(y) then y.equals(x))
- It's transitive (if x.equals(y) and y.equals(z) then x.equals(z))
- It handles null properly
- It only compares objects of the same type

3. Give the ordered list of methods executed (for each list element, give the method name and the class where it's defined) following the message: f1.drawDescription();
The sequence is:

1. drawDescription() from Forme class
2. getDescription() from Ligne class (due to polymorphism)
3. toString() from Point class (called twice, once for each point)

4. Is the instruction: 'Vector<Forme> v = new Vector<Cercle>();' accepted by the Java compiler? Comment.

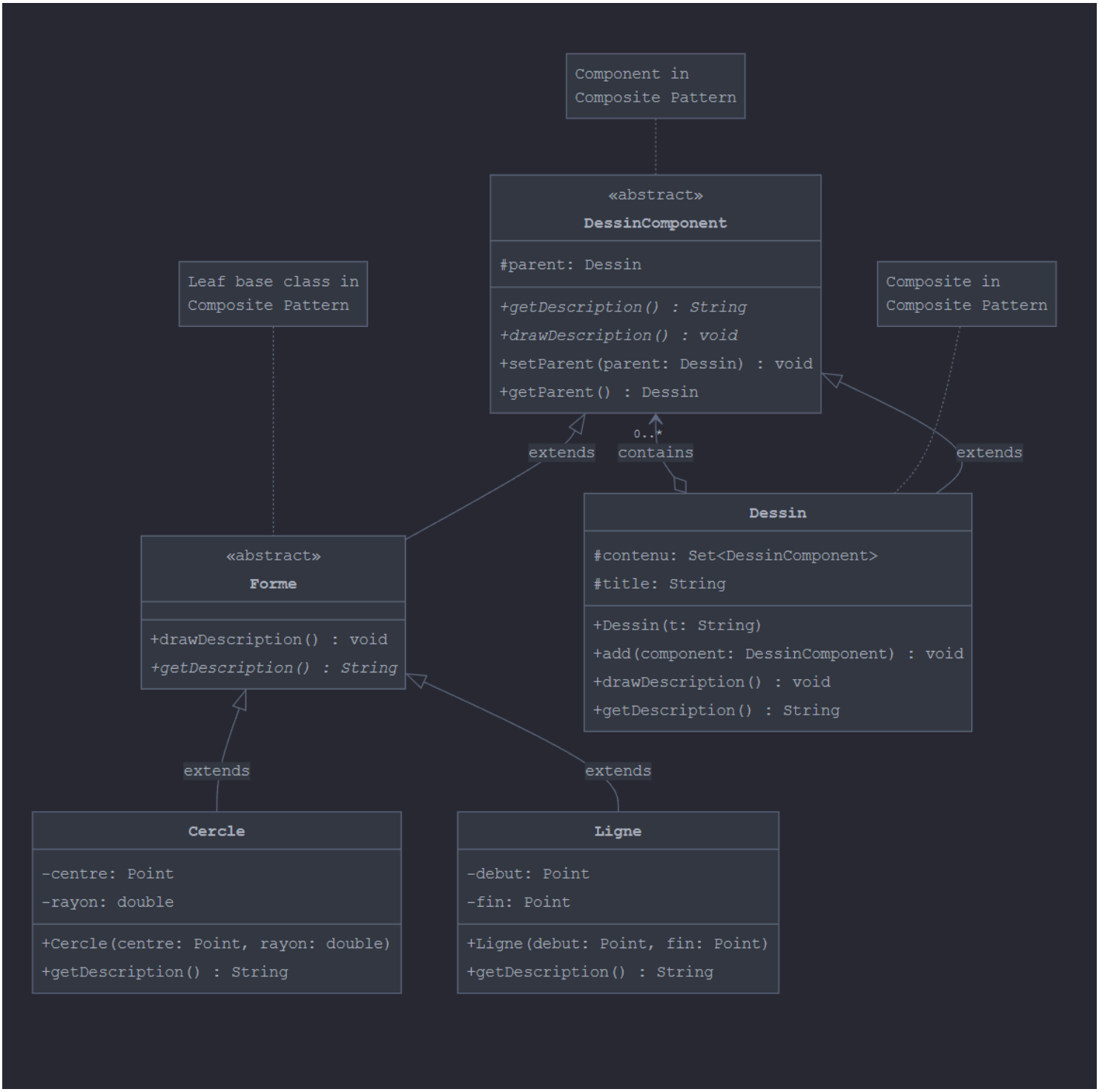
This is NOT accepted by the Java compiler because of type safety. Even though Cercle is a subtype of Forme, Vector<Cercle> is not a subtype of Vector<Forme>. This prevents situations where you could add a Ligne to what's supposed to be a collection of only Cercle objects.

5. As a second solution, we imagine factoring an equals method on the Forme class; do you see a solution that could work for our problem? If not, why? If yes, give your idea with a phrase or code.

```
public abstract class Forme {
        // Each shape defines what makes it unique
        protected abstract Object[] getEqualityComponents();
        @Override
        public final boolean equals(Object o) {
                if (this == o) return true;
                if (o == null || getClass() != o.getClass()) return false;
                Forme other = (Forme) o;
                return Arrays.equals(getEqualityComponents(),
                                other.getEqualityComponents()); } }
```

Exercice C

1. Modify the base architecture of frampict to allow a drawing to be composed of another drawing. This should allow executing the instructions in listing 4.



```
// First, we need a common interface/class for both shapes and drawings
public abstract class DessinComponent {
        public abstract String getDescription();
        public abstract void drawDescription(); }
// Now Forme becomes a child of DessinComponent
public abstract class Forme extends DessinComponent {
        @Override
        public void drawDescription() { System.out.println(this.getDescription()); }
        public abstract String getDescription(); }
 // Dessin also becomes a child of DessinComponent
public class Dessin extends DessinComponent {
        protected Set<DessinComponent> contenu; // Note: now can contain any
        DessinComponent protected String title;
        public Dessin(String t) { contenu = new HashSet<DessinComponent>(); title = t; }
        public void add(DessinComponent component) { contenu.add(component); }
        @Override
        public void drawDescription() { System.out.println(this.getDescription() + " obtenu par :");
        for (DessinComponent component : contenu) {
                if (component instanceof Dessin) {
                        System.out.print("- "); // Add indentation for nested drawings }
                        component.drawDescription(); } }
        @Override
        public String getDescription() { return "Un dessin de " + title; } }
```

2. Improve, if needed, this new architecture, or explain why the previous one is sufficient, to be able to define a new method dessinsEnglobants() that returns, for a given shape, the collection of drawings to which it belongs. For the 'circle with center: 7.0@7.0' in listing 4, the result would be a collection containing d1 and d2.

```
public abstract class DessinComponent {
         // Add reference to parent drawing
        protected Dessin parent;
        public void setParent(Dessin parent) { this.parent = parent; }
        public Dessin getParent() { return parent; }
         // Get all containing drawings (including nested ones)
        public Collection<Dessin> dessinsEnglobants() {
                List<Dessin> containers = new ArrayList<>();
                Dessin current = this.parent;
                while (current != null) {
                        containers.add(current);
                        current = current.getParent(); }
                return containers; }
        // ... previous methods ... }
```

```
public class Dessin extends DessinComponent {
@Override
public void add(DessinComponent component) {
        contenu.add(component); component.setParent(this);// Set parent reference }
// ... other methods remain the same ... }
```

Exercice D

1. Why is it not possible for a java.awt.Point to be used as a shape in FramPict?

java.awt.Point cannot be used directly as a shape because:

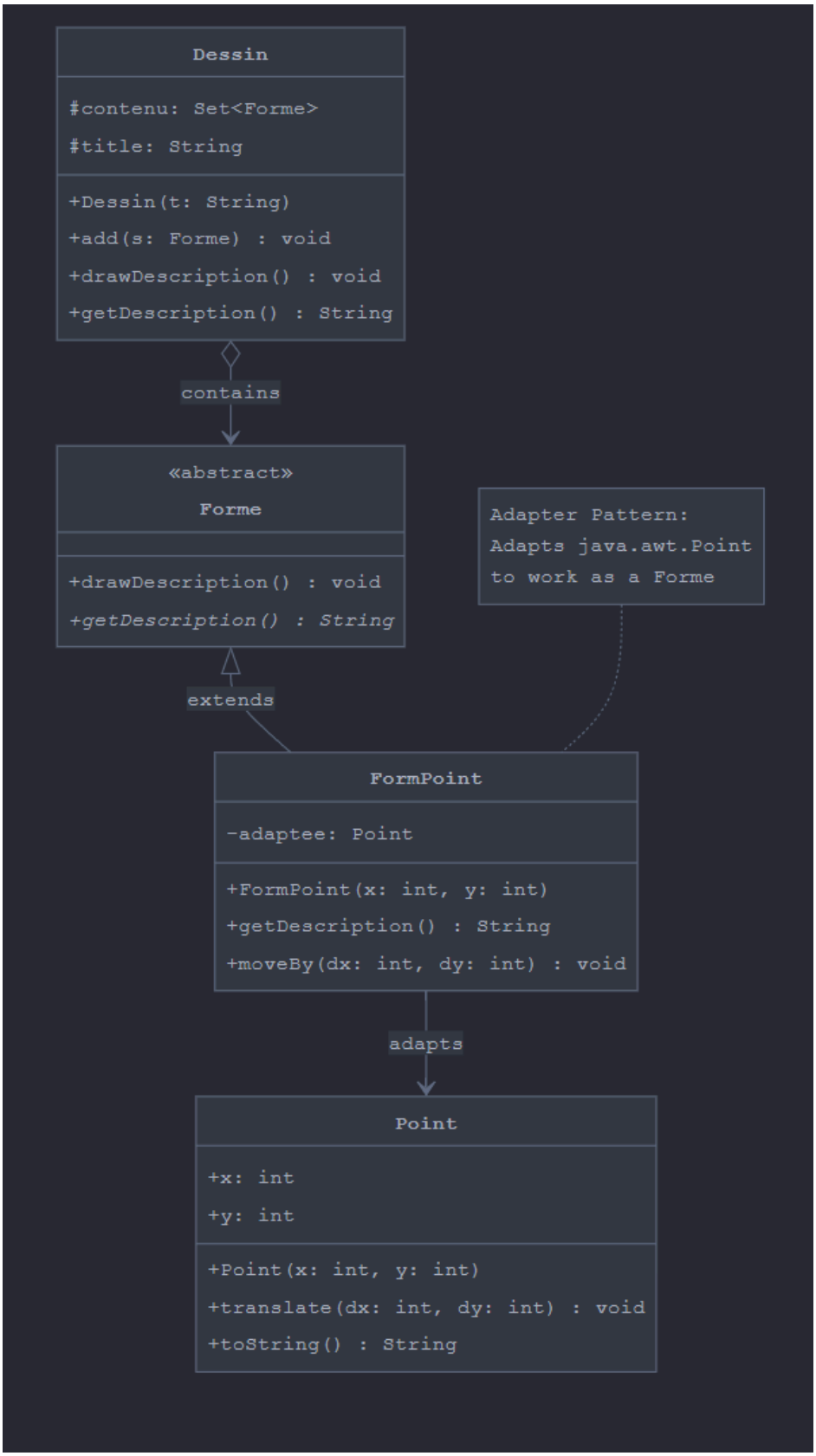
1. It doesn't extend our Forme class (Shape class)
2. We can't modify the java.awt.Point class since it comes from the Java library
3. It doesn't have the necessary methods (like getDescription()) that our shapes need
4. We can't make Point inherit from Forme after the fact - Java doesn't allow multiple inheritance

2. We imagine a solution via an adaptation class FormPoint. If d1 is the drawing instance from listing 2, then we could write: d1.add(new FormPoint(4,5)). d1.drawDescription() would consequently display the following text at the terminal: '- un point de coordonnees 4.0@5.0'. Give a UML schema of the solution and the key code elements that make this work. If you use a known design pattern, comment on it.

```
public class FormPoint extends Forme {
        private Point adaptee;
         // The wrapped Point object
        public FormPoint(int x, int y) {
                // Create and store the adaptee
                Point this.adaptee = new Point(x, y); }
        @Override
        public String getDescription() {
                return "un point de coordonnees " + adaptee.x + "@" + adaptee.y; } }
```

This is using the Adapter pattern because:

- FormPoint adapts Point to work with our Forme interface
- It wraps a Point object and provides the required interface
- It translates between the two incompatible interfaces
- It allows Point to work in our system without modifying Point itself


Dessin
#contenu: Set<Forme>
#title: String
+Dessin(t: String)
+add(s: Forme) : void
+drawDescription() : void
+getDescription() : String
contains
«abstract»
Forme
+drawDescription() : void
+getDescription() : String
Adapter Pattern:
Adapts java.awt.Point
to work as a Forme
extends
FormPoint
-adaptee: Point
+FormPoint(x: int, y: int)
+getDescription() : String
+moveBy(dx: int, dy: int) : void
adapts
Point
+x: int
+y: int
+Point(x: int, y: int)
+translate(dx: int, dy: int) : void
+toString() : String

3. In connection with the message redirection used in this solution, give an example in this context of an occurrence of the problem of losing the initial receiver.

The problem of losing the initial receiver could occur if we need to preserve the identity of the original Point while delegating methods.

```
public class FormPoint extends Forme {
        private Point adaptee;
        public void moveBy(int dx, int dy) {
                // If another method needs to know which Point was moved,
                // they'd only see the adaptee Point, not the FormPoint
                adaptee.translate(dx, dy);
                // Any code that needs to know which FormPoint was moved
                // has lost that information - they only see the Point } }
```

When we redirect the movement operation to the adaptee Point, we lose the context that this was part of a FormPoint operation. If other parts of the system need to know which FormPoint was moved (for example, to update the drawing), they can't get this information from the Point object alone.

Exercise E

```
// First, we define the Command interface
public interface Command {
        void execute();  // Perform the operation
        void undo(); // Reverse the operation }
// Create a concrete command for adding shapes
public class AddShapeCommand implements Command {
        private Dessin dessin; private Forme forme;
        public AddShapeCommand(Dessin dessin, Forme forme) {
                this.dessin = dessin; this.forme = forme; }
        @Override
        public void execute() { dessin.addShape(forme); // Actually add the shape }
        @Override
        public void undo() { dessin.removeShape(forme); // Remove the shape } }
 // Modify the Dessin class to use commands
public class Dessin {
        private Vector<Forme> contenu; // Now using Vector instead of HashSet
        private Stack<Command> history; // Stack to keep track of commands
        public Dessin(String title) {
                this.contenu = new Vector<>(); this.history = new Stack<>(); this.title =
        title; }
                // Modified add method to use command
        public void add(Forme forme) {
                Command cmd = new AddShapeCommand(this, forme); cmd.execute();
                history.push(cmd); } // Internal methods used by the command
        protected void addShape(Forme forme) { contenu.add(forme); }
```

```
protected void removeShape(Forme forme) { contenu.remove(forme); }
// New undo method
public void undo() { if (!history.isEmpty()) { Command lastCommand =
history.pop(); lastCommand.undo(); } } }
```