Correction du TME 6

LU3IN002 : Programmation par objets
L3, Sorbonne Université

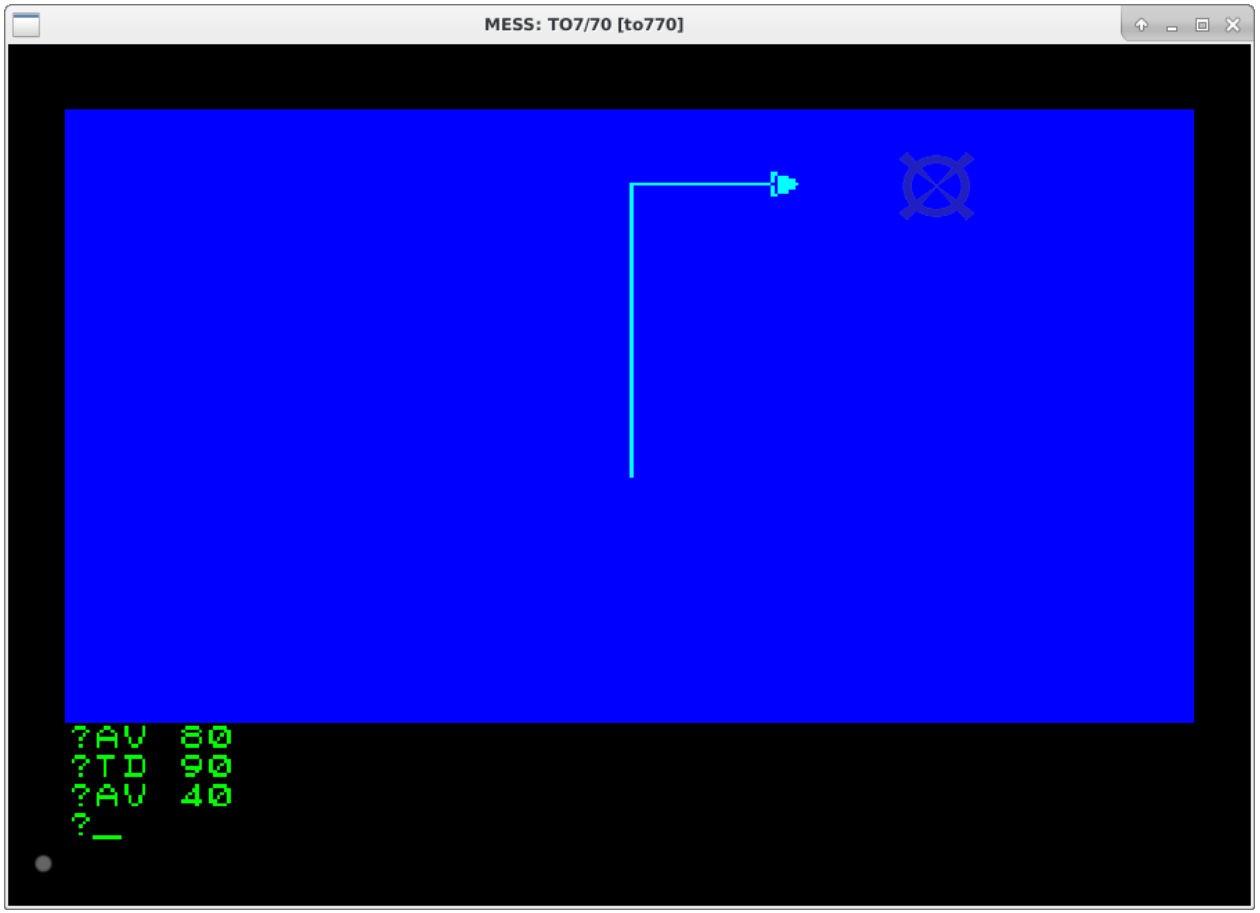
http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2019/ue/LU3IN002-2020fev

Antoine Miné

Année 2019–2020

Cours 7bis
6 mars 2020

Inspiration : le langage LOGO



LOGO : langage de programmation pour l'éducation, inventé en 1967.
Langage fonctionnel, dialecte de LISP (manipulations de listes).
Mais surtout connu pour l'utilisation de "tortues graphiques" pour dessiner.

1 – Interface ITurtle

eclipse-workspace/Tortue/src/pobj/tme6/ITurtle.java

```
package pobj.tme6;

public interface ITurtle
{
    public void move(int length);
    public void turn(int angle);
    public void up();
    public void down();
}
```

L'interface est **imposée**, donc :

- interdit de la modifier ;
  le correcteur implante aussi l'interface et ne fonctionnerait plus !
- ne pas oublier les `implements` ou `extends ITurtle` dans la suite.
  vos objets seront convertis en `ITurtle` par le correcteur

Les interfaces servent à minimiser les dépendances entre objets et à abstraire :

- facilitent la programmation à grande échelle ;
- facilitent l'évolution, le re-engineering ;

$\Longrightarrow$ programmer vis à vis d'une interface, pas d'une implantation.

1 – Tortue simple : classe Turtle

eclipse-workspace/Tortue/src/pobj/tme6/Turtle.java

```
public class Turtle implements ITurtle
{
    private int x, y, angle;
    private boolean isDown = true;

    void draw(int x1, int y1, int x2, int y2)
    {
        System.out.println(x1 + " " + y1 + " " + x2 + " " + y2);
    }

    @Override public void move(int length)
    {
        int newX = x + (int)(length * Math.sin(angle * Math.PI / 180.));
        int newY = y + (int)(length * Math.cos(angle * Math.PI / 180.));
        if (isDown) draw(x, y, newX, newY);
        x = newX;
        y = newY;
    }

    @Override public void turn(int angle)   { this.angle += angle; }
    @Override public void up()              { isDown = false; }
    @Override public void down()            { isDown = true; }
}
```

- `x` et `y` doivent être mis à jour, même si `isDown` est faux !
- `turn` doit incrémenter `angle` (pas faire : `this.angle = angle`).

1 – Test de notation de Turtle

test JUnit de la question 1

```
import org.junit.*;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;

public class Question1Notation
{
    private ByteArrayOutputStream out = new ByteArrayOutputStream();
    @Before public void atBefore() { System.setOut(new PrintStream(out)); }
    private String getOut()        { System.out.flush(); return out.toString(); }

    private String normalize(String r) { return r.replaceAll("[\n ]", ""); }

    private void assertEqualsNormalized(String a, String b)
    { assertEquals(normalize(a), normalize(b)); }

    @Test public void testMoveOne()
    {
        ITurtle t = new Turtle();
        t.move(10);
        assertEqualsNormalized(getOut(), "0 0 0 10\n");
    }
}
```

- redirection de la sortie standard dans une chaîne : `setOut`, `out.toString` ;
- comparaison de chaînes en ignorant les espaces (avec `replaceAll`).

2 – Tortue colorée : IColorTurtle et ColorTurtle

IColorTurtle.java

```
public interface IColorTurtle extends ITurtle
{
    public void setColor(javafx.scene.paint.Color color);
}
```

ColorTurtle.java

```
import javafx.scene.paint.Color;
public class ColorTurtle extends Turtle implements IColorTurtle
{
    private Color color;

    public ColorTurtle() { this.color = Color.BLACK; }

    @Override public void draw(int x1, int y1, int x2, int y2)
    { System.out.println(x1 + " " + y1 + " " + x2 + " " + y2 + " " + color); }

    @Override public void setColor(Color color) { this.color = color; }

    public Color getColor() { return color; }
}
```

- `draw` doit avoir la même signature dans `ColorTurtle` et dans `Turtle` ;
  sinon, `move` continue à appeler la version de `draw` de `Turtle`
- la définition d'un accesseur `getColor` facilitera les questions suivantes
  tout en évitant d'exposer l'attribut `color` au package ou au monde.

2 – Notation de ColorTurtle et IColorTurtle

Question2NotationClass.java

```
@Test public void testClass()
{
    IColorTurtle t = new ColorTurtle();
    assertTrue(t instanceof ITurtle);
    assertTrue(t instanceof Turtle);
    assertTrue(ColorTurtle.class.getSuperclass() == Turtle.class);
    assertTrue(IColorTurtle.class.isInterface());
}
```

Question2NotationMethods.java

```
@Test public void testMove()
{
    IColorTurtle t = new ColorTurtle();
    t.setColor(Color.RED);
    t.move(10);
    t.setColor(Color.BLUE);
    t.move(20);
    assertEqualsNormalized(getOut(), "0 0 0 10 0xff0000ff\n0 10 0 30 0x0000ffff\n");
}
```

Notation en combinant des tests indépendants sur :

- la compilation et le respect des relations d'héritage, d'interface ;
- la sortie (tests fonctionnels).

3 – Motif adaptateur : classe ColorTurtleAdapter

ColorTurtleAdapter.java

```
public class ColorTurtleAdapter implements IColorTurtle
{
    private ITurtle turtle;

    public ColorTurtleAdapter(ITurtle turtle)    { this.turtle = turtle; }

    @Override public void move(int length)       { turtle.move(length); }
    @Override public void turn(int angle)        { turtle.turn(angle); }
    @Override public void up()                   { turtle.up(); }
    @Override public void down()                 { turtle.down(); }

    @Override public void setColor(Color color)  { /* vide */ }
}
```

Il s'agit de déléguer, pas d'hériter ni de réimplanter.

La délégation :

- ne mobilise pas le lien d'héritage (possibilité de déléguer à plusieurs classes) ;
- est polymorphe (tout objet implantant `ITurtle` peut servir) ;
- permet de changer l'interface (extension, restriction).

3 – Notation de l'adaptateur, classe simulacre (*mock*)

Question3NotationMethods.java

```
public class Question3NotationMethods
{
    private class MockTurtle implements ITurtle
    {
        public boolean moveCalled, turnCalled, upCalled, downCalled;
        public void move(int length)   { assertEquals(length,999); moveCalled = true; }
        public void turn(int angle)    { assertEquals(angle,888); turnCalled = true; }
        public void up()               { upCalled = true; }
        public void down()             { downCalled = true; }
    }

    @Test public void test1()
    {
        MockTurtle t = new MockTurtle();
        IColorTurtle c = new ColorTurtleAdapter(t);
        c.up(); c.down(); c.move(999); c.turn(888);
        assertTrue(t.moveCalled && t.turnCalled && t.upCalled && t.downCalled);
    }
}
```

Le test utilise une classe interne implantant un simulacre de tortue (*mock object*) permettant de vérifier que `ColorTurtleAdapter` délègue bien :

- un appel à `ColorTurtleAdapter` se traduit par un appel à MockTurtle ;
- les arguments passés à `ColorTurtleAdapter` sont bien passés à MockTurtle ;
- l'interface `ITurtle` n'a pas été modifiée.

4 – Motif stratégie : classe ContextTurtle

IContext.java

```
public interface IContext
{
    public void drawLine(int x1, int y1, int x2, int y2, Color color);
}
```

ContextTurtle.java

```
public class ContextTurtle extends ColorTurtle
{
    private IContext context;

    public ContextTurtle(IContext context)
    { super(); this.context = context; }

    @Override public void draw(int x1, int y1, int x2, int y2)
    { context.drawLine(x1, y1, x2, y2, getColor()); }
}
```

- Séparer la gestion de la tortue (création, état, méthodes), programmée une fois pour toute dans ContextTurtle ;
- et la gestion de l'affichage (sortie), paramétrable par une stratégie IContext.

Permet de réutiliser une implantation pour différentes sorties sans hériter, mais en déléguant l'opération qui varie à une classe dédiée par opération.

4 – Notation de la tortue avec délégation à la stratégie

Question4NotationTurtle.java

```
public class Question4NotationTurtle
{
  private class MockContext implements IContext
  {
      public void drawLine(int x1, int y1, int x2, int y2, Color color) {
      {  System.out.println("*" + x1 + " " + y1 + " " + x2 + " " + y2 + " " + color); }
  }

  @Test public void test1()
  {
    IColorTurtle t = new ContextTurtle(new MockContext());
    t.turn(90); t.move(10);
    t.setColor(Color.BLUE); t.turn(45); t.move(20);
    t.up(); t.move(30);
    t.down(); t.setColor(Color.RED); t.move(10);
    assertEqualsNormalized(
        getOut(),
        "*0 0 10 0 0x000000ff\n*10 0 24 -14 0x0000ffff\n*45 -35 52 -42 0xff0000ff\n");
  }
}
```

Encore une utilisation d'un simulacre pour tester la délégation.

4 – Affichage : PrintContext

PrintContext.java

```
public class PrintContext implements IContext
{
    @Override public void drawLine(int x1, int y1, int x2, int y2, Color color)
    {
        System.out.println(x1 + " " + y1 + " " + x2 + " " + y2 + " " + color);
    }
}
```

Un exemple de stratégie : affichage à l'écran.

4 – Calcul de la boîte englobante : BoundContext

BoundContext.java

```
public class BoundContext implements IContext
{
    private int minX, minY, maxX, maxY;
    private boolean first = true;

    public int getMinX() { return minX; }
    public int getMinY() { return minY; }
    public int getMaxX() { return maxX; }
    public int getMaxY() { return maxY; }

    private void addPoint(int x, int y)
    {
        if (first) { minX = x; minY = y; maxX = x; maxY = y; first = false; }
        else {
            if (x < minX) { minX = x; } else if (x > maxX) { maxX = x; }
            if (y < minY) { minY = y; } else if (y > maxY) { maxY = y; }
        }
    }

    @Override public void drawLine(int x1, int y1, int x2, int y2, Color color)
    { addPoint(x1,y1); addPoint(x2,y2); }
}
```

- attention à tenir compte des deux extrémités de chaque ligne ;
- attention au cas particulier du premier point ;
  la boîte englobante ne contient pas forcément le point $(0, 0)$!

5 – Motif commande : ICommand

ICommand.java

```
public interface ICommand
{
    public void execute(IColorTurtle turtle);
}
```

CommandMove.java

```
public class CommandMove implements ICommand
{
    private int length;
    public CommandMove(int length) { this.length = length; }

    @Override public void execute(IColorTurtle turtle)
    { turtle.move(length); }
}
```

idem pour CommandUp, CommandDown, CommandTurn, CommandSetColor...

- inversion de la relation objet / action :

  ```
  turtle.cmd(arg);
  ```

  devient :

  ```
  cmd = new Cmd(arg);
  cmd.execute(turtle);
  ```

- réification des actions ;
  elles peuvent être stockées, passées en argument, réexécutés, etc.

6 – Liste de commandes : CommandList

CommandList.java

```
public class CommandList implements ICommand
{
    private List<ICommand> commands = new Vector<ICommand>();

    public void addCommand(ICommand command)
    { commands.add(command); }

    @Override public void execute(IColorTurtle turtle)
    {
        for (ICommand c : commands)
            c.execute(turtle);
    }
}
```

- accumulation des commandes dans la liste avec `addCommand` ;
- puis exécution d'une liste de commandes avec `execute` ;
  $\Longrightarrow$ l'exécution est récursive et explore les listes imbriquées ;
- attention à bien initialiser `commands` dans le constructeur !

Une liste de commandes est vue comme une commande
$\Longrightarrow$ motif composite.

6 – Accumulateur de commandes : SaveTurtle

SaveTurtle.java

```
public class SaveTurtle implements IColorTurtle
{
    private CommandList buffer = new CommandList();

    @Override public void move(int length)
    { buffer.addCommand(new CommandMove(length)); }

    @Override public void turn(int angle)
    { buffer.addCommand(new CommandTurn(angle)); }

    @Override public void up()
    { buffer.addCommand(new CommandUp()); }

    @Override public void down()
    { buffer.addCommand(new CommandDown()); }

    @Override public void setColor(Color color)
    { buffer.addCommand(new CommandSetColor(color)); }

    public CommandList getCommand() { return buffer; }
}
```

Une tortue qui enregistre les commandes à réaliser au lieu de les exécuter.

Note : substitutions et L-systems

Systèmes de réécriture, inventés par le biologiste Lindenmayer en 1968 pour modéliser le développement des plantes.

Exemple original : modèle d'algue

- départ : A
- règles : $A \to AB$, $B \to A$

Calcul par substitutions dans une chaîne de caractères :

- étape 0 : A
- étape 1 : AB
- étape 2 : ABA
- étape 3 : $ABAAB$
- ...

Autre exemple, fractale de von Koch :

- départ : $F--F--F$
- règle : $F \to F+F--F+F$
- après quelques itérations, interprétation de la chaîne produite :
  $F \Rightarrow \texttt{move}, \quad + \Rightarrow \texttt{turn}(60), \quad - \Rightarrow \texttt{turn}(-60)$

7 – Substitution des commandes : Substitution

Substitution.java

```
public class Substitution
{
    static public ICommand substitute(ICommand org, final ICommand subst)
    {
        SaveTurtle s = new SaveTurtle()
        {
            @Override public void move(int length)
            { getCommand().addCommand(subst); }
        };
        org.execute(s);
        return s.getCommand();
    }
}
```

`s` est un objet d'une classe anonyme :

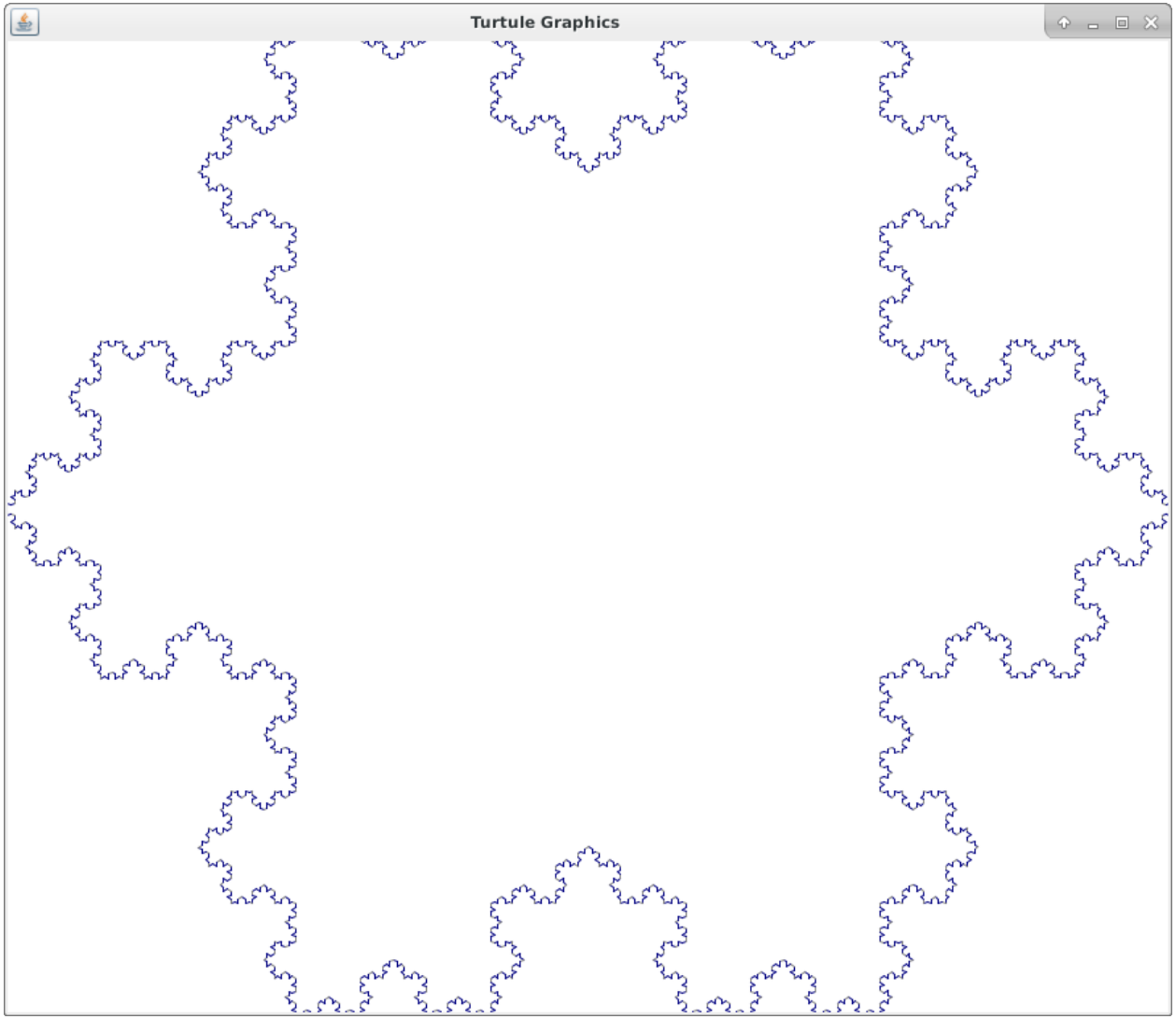
- héritant de `SaveTurtle` ;
- mais dont la méthode `move` ajoute la commande `subst` au lieu de `new CommandMove(length)`.

L'exécution de `org` par `s` va donc créer une nouvelle liste de commandes égale à `org`, sauf que chaque `CommandMove` est remplacé par `subst`.

en pratique, `org` et `subst` sont des `CommandList`

Note : les objets `ICommand` sont tous immuables ; inutile donc d'effectuer des copies et d'implanter `clone`...

Résultat final



Classe `KochMain`, utilisant le contexte graphique `JFXContext`.