TME 6 : Tortues (entraînement au partiel sur machine)

Déroulement du TME et du partiel sur machine

Ce TME est basé sur le partiel de l'année 2015–2016.
Il a pour but de vous familiariser avec l'environnement qui sera utilisé lors du partiel sur machine de la semaine suivante.

L'examen se déroulera sur machine et sera individuel. Pour ce TME d'entraînement, toutefois, nous travaillerons comme d'habitude en binôme.

L'examen demandera d'écrire un certain nombre de fichiers en Java. Nous conseillons, sans que cela soit obligatoire, d'utiliser l'environnement de développement Eclipse. Les fichiers fournis seront évalués automatiquement et notés grâce à une série de tests JUnit 4 exécutés sur un serveur distant. Le serveur produit un compte-rendu détaillé de l'exécution (et éventuellement de l'échec) de la compilation et des tests et calcule votre note : la compilation sans erreur puis chaque test passé avec succès vous rapportera un certain nombre de points, suivant le barème précisé.

Pendant le partiel, vous travaillerez *off-line*, sans connexion au serveur de correction, sur un compte vierge. Les fichiers trouvés sur votre compte seront automatiquement ramassés à la fin de l'épreuve : ils constituent votre copie. Le sujet propose quelques exemples de tests JUnit pour vous auto-évaluer, mais la batterie complète des tests servant à la notation ne sera pas communiquée durant l'épreuve. Le rapport de correction sera consultable après l'épreuve : il aura valeur de copie corrigée.

Toutefois, pour ce TME d'entraînement, nous allons travailler en mode connecté : vous soumettrez vous-même au serveur de correction au fur et à mesure du TME une archive `zip` de votre travail pour correction immédiate. Nous utiliserons le serveur GitLab habituel, mais uniquement pour y déposer l'archive des sources, pas pour la gestion de projet git. Par ailleurs, nous fournissions ici directement l'ensemble des tests JUnit 4 qui ont servi à la notation, en plus des exemples fournis aux étudiants pendant l'épreuve.

Afin d'éviter que vos réponses soient rejetées, assurez-vous de :

- respecter les noms de classe, de package et de chemin des fichiers indiqués dans l'énoncé ;
- fournir des sources qui compilent, même si elles ne répondent pas complètement à la question (les méthodes qui fonctionnent pourront vous rapporter une partie des points) ;
- implanter les interfaces et hériter des classes spécifiées dans l'énoncé ;
- respecter les consignes de visibilité des méthodes ; les attributs seront toujours privés ;
- ne pas modifier les interfaces fournies dans l'énoncé ; sinon, les tests de notation, qui sont programmés vis à vis de ces interfaces, ne compileront plus.

Un source qui n'est pas trouvé par le correcteur ou qui ne compile pas ne rapporte aucun point.

Lors du partiel, tous les documents papier (transparents du cours, notes personnelles, etc.) seront autorisés. L'utilisation d'appareils électroniques, de moyens de communication ou d'accès à internet (téléphone, tablette, ordinateur personnel, etc.) sera interdite. L'utilisation d'une clé ou disque USB personnel, en lecture seule et étiqueté à votre nom, sera autorisée ; vous ne devrez pas le prêter à un camarade. Attention, en mode examen, seules les clés au format VFAT fonctionnement (le format NTFS n'est pas supporté).

Mise en place

Nous n'allons pas utiliser git dans ce TME. Le serveur GitLab habituel sera utilisé, mais uniquement pour lancer la correction automatique, pas pour héberger vos sources. Le squelette du projet sera par contre importé dans Eclipse depuis une archive ZIP, comme ce sera le cas lors du partiel.

Téléchargez l'archive `Tortue.zip` disponible sur le site de l'UE. Celle-ci contient les sources des classes et interfaces fournies par l'énoncé, les tests JUnit fournis lors du partiel 2015–2016 pour

vous aider pendant l'épreuve, ainsi que les tests JUnit qui ont servi à la notation mais n'étaient pas fournis durant l'épreuve.

Dans Eclipse, créez un projet à partir de l'archive. Pour cela, utilisez le menu « File > Import > General > Existing Projects into Workspace » ; sélectionnez « Select archive file » ; cliquez sur « Browse » et sélectionnez le ficher archive « `Tortue.zip` ». Vérifiez que le projet « Tortue » apparaît bien dans la boîte « Projects », et cliquez sur « Finish ». Vous devriez maintenant avoir un projet nommé « Tortue » dans le « Package Explorer » d'Eclipse. Les fichiers sont stockés dans le répertoire `~/eclipse-workspace/Tortue/`, en supposant que votre *workspace* Eclipse se trouve dans le répertoire `eclipse-workspace` à la racine de votre compte.

Le sujet suppose que toutes les classes sont créées dans le package nommé `pobj.tme6`.

Connectez-vous maintenant sur le serveur GitLab habituel. Vous y trouverez un projet GitLab nommé `Tortue`. Faîtes-en un *fork*. Le projet ne contient que des fichiers utiles à la notation sur le serveur, il est donc inutile de l'importer localement.

Notation

Pour obtenir votre note, vous devez déposer à la racine de votre *fork* GitLab vos sources sous forme d'une archive nommée `upload.zip`. L'archive doit contenir un répertoire `pobj`, contenant un sous-répertoire `tme6`, contenant vos sources. Pour créer l'archive, en supposant que vous utilisez le répertoire *workspace* d'Eclipse par défaut, vous pouvez faire :

```
cd ~/eclipse-workspace/Tortue/src
zip -r upload.zip pobj
```

Pour déposer une première fois l'archive sous GitLab, vous pouvez cliquer dans l'interface Web sur le bouton `+` à droite du nom de répertoire courant (`Tortue/`), puis sur « Upload file ». Pour la remplacer par une nouvelle version, vous pouvez cliquer sur le nom du fichier, `upload.zip`, dans GitLab, puis sur le bouton « Replace ». Dans ce TME, il ne sera pas nécessaire d'utiliser git ni de synchroniser votre projet Eclipse avec le serveur GitLab : seul le dépôt de l'archive est demandé.

Après chaque dépôt de l'archive, le mécanisme d'intégration continue de GitLab (re)lance automatiquement le processus de correction automatique. Votre rapport de correction est disponible sous forme de page web dans l'onglet « CI/CD » de votre projet. La ligne la plus en haut correspond à la dernière correction effectuée. Le rapport est alors disponible via l'icône « Artifacts » dans la colonne de droite, sous forme d'archive `zip` contenant un seul fichier `index.html` que vous pouvez ouvrir dans un navigateur. Notez que l'intégration continue indiquera toujours un succès (icône V verte) quelle que soit votre note finale, même si votre copie n'est pas parfaite.

Durant l'examen, afin que le ramassage automatique de vos sources fonctionne correctement, il sera très important de respecter le nom des répertoires : le répertoire *workspace* (qui devra être `~/eclipse-workspace`) et le nom du projet Eclipse et des packages comme précisé dans l'énoncé.

Par exemple, si nous étions en mode examen, au vu des consignes ci-dessus, seul le répertoire `~/eclipse-worspace/Tortue/src/pobj/tme6` (et ses sous-répertoires) serait ramassé et corrigé.

Introduction

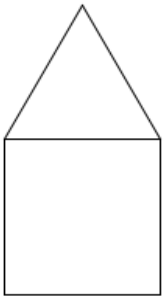
Le but de ce problème est de développer un système procédural de dessin inspiré des « tortues » du langage LOGO.

Une tortue est simplement un curseur, disposé à une certaine position de l'écran et orienté dans une certaine direction. La tortue peut être déplacée avec des ordres simples : avancer dans la direction dans laquelle elle est orientée, ou tourner sur elle-même. La tortue porte un crayon qu'elle

peut relever ou baisser. Si le crayon est baissé, la tortue laisse une trace sur l'écran lors de ses déplacements, ce qui permet d'élaborer des dessins.

Voici, par exemple, une séquence d'ordres simples et le résultat obtenu :

```
move 100
turn 90
move 100
turn 90
move 100
turn 30
move 100
turn 120
move 100
turn 120
move 100
```



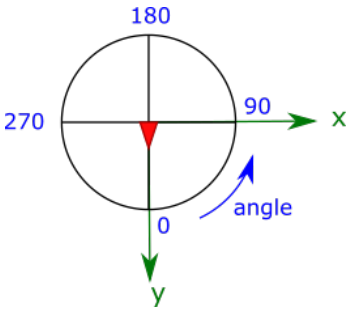
Question 1 : Tortue simple (implantation d'une interface)

Une tortue est définie par l'état suivant :

- ses coordonnées, `x` et `y`, de type `int` ;
- un angle en degrés, de type `int`, indiquant la direction courante ;
- l'état du crayon : levé ou baissé (un booléen sera utilisé).

Une tortue est toujours créée dans l'état suivant :

- sa position initiale est $x = 0$, $y = 0$;
- son angle initial est 0 (elle se déplacera donc vers le bas, dans le sens des `y` croissants, voir dessin ci-contre) ;
- le crayon de la tortue est baissé (un déplacement dessinera un trait).



Une tortue obéit à l'interface `ITurtle` suivante :

```java
package pobj.tme6;

/**
 * Interface pour une tortue logo.
 *
 */
public interface ITurtle {

   /**
    * Avance la tortue d'une certaine distance dans la direction courante
    * Trace une ligne entre le point de départ et le point d'arrivée si le crayon
    * est baissé.
    * @param length distance à avancer
    */
   public void move(int length);

   /**
    * Tourne la tortue sur elle-même, en chageant la direction courante.
    * @param angle angle en degrés à ajouter à la direction courante.
    */
   public void turn(int angle);
```

```
    /**
     * Lève le crayon.
     * Les appels suivants à move ne laisseront pas de trace.
     */
    public void up();

    /**
     * Baissse le crayon.
     * Les appels suivants à move laisseront une trace.
     */
    public void down();
}
```

Ces méthodes ont pour effet de modifier l'état interne de la tortue, et éventuellement de dessiner des traits :

- `move(length)` déplace la tortue d'une distance `length` dans la direction courante. Par ailleurs, la méthode trace un trait entre la position de départ et la position d'arrivée, mais seulement si le crayon est baissé (si le crayon est levé, la tortue se déplace, mais sans dessiner de trait). Nous utiliserons les formules suivantes pour calculer la nouvelle position `newX`, `newY` de la tortue en fonction de l'ancienne position `oldX`, `oldY`, de la distance `length` parcourue et de l'angle courant `angle` en degrés :

```
newX = oldX + (int)(length * Math.sin(angle * Math.PI / 180.));
newY = oldY + (int)(length * Math.cos(angle * Math.PI / 180.));
```

- `turn(angle)` ajoute `angle` degrés à la direction courante (rotation dans le sens trigonométrique) ;
- `up()` lève le crayon ;
- `down()` baisse le crayon.

Cette question vous demande d'écrire une classe `Turtle` implantant l'interface `ITurtle` et avec un constructeur sans argument.

Nous ne demandons pas ici d'affichage graphique. Le tracé de chaque ligne sera matérialisé par la sortie, en utilisant `System.out.println`, d'une ligne de texte de la forme :

```
x1 y1 x2 y2
```

où `x1` et `y1` sont les coordonnées de la première extrémité de la ligne, et `x2` et `y2` sont les coordonnées de la deuxième extrémité de la ligne. Il s'agit donc de quatre entiers, séparés par des espaces, et suivis d'un retour à la ligne. Une ligne de cette forme est émise par chaque appel à `move` quand le crayon est baissé.

Afin de faciliter les évolutions demandées aux questions suivantes, vous isolerez la fonctionnalité qui affiche la ligne de texte demandée dans une méthode :

```
void draw(int x1, int y1, int x2, int y2);
```

La méthode `draw` sera appelée par `move`, et pourra être redéfinie indépendemment de la partie de `move` qui calcule et met à jour les coordonnées de la tortue.

Voici, à gauche, un exemple d'utilisation de la tortue et, à droite, le résultat attendu de l'exécution (trois traits) :

```
ITurtle t = new Turtle();
t.move(10);
t.turn(45);
t.move(20);
t.up();
t.move(30);
t.down();
t.move(10);
```

```
0 0 0 10
0 10 14 24
35 45 42 52
```

Respectez bien la syntaxe demandée pour la sortie et les formules données pour le calcul ! Votre implantation est notée par comparaison entre votre sortie et la sortie d'une implantation de référence. Toute différence, même minime (espace au mauvais endroit, valeur légèrement différente, etc.) pourra rendre votre réponse invalide !

À fournir :

Un fichier `pobj/tme6/Turtle.java`

```java
package pobj.tme6;

public class Turtle implements ITurtle {

   private int x;
   private int y;
   private int angle;
   private boolean isDown;

   public Turtle() {
      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;
   }


   // Accesseurs, permettant de garder les attributs privés

   public int getX() { return x; }
   public int getY() { return y; }
   public int getAngle() { return angle; }
   public boolean isDown() { return isDown; }

   void setX(int x) { this.x = x; }
   void setY(int y) { this.y = y; }

}
```

Fourni :

L'archive `Tortue.zip` fournit l'interface `pobj.tme6.ITurtle`, l'exemple ci-dessus sous forme de test JUnit 4 permettant de valider votre réponse (`pobj.tme6.test.Question1Test`, fourni lors de l'examen) et les classes de test ayant servi à la notation (dans `pobj.tme6.notation`, *non fournies* lors de l'examen).

Attention : ne modifiez pas le fichier `ITurtle.java` car les classes de test, programmées vis à vis de cette interface, ne fonctionneront alors plus, et votre classe `Turtle` sera considérée comme incorrecte ! Ceci restera valable pour tous les fichiers fournis aux questions suivantes, et nous ne répéterons pas cet avertissement.

Notation : 3 points

- 1 point si la classe `Turtle` compile et implante bien l'interface `ITurtle` spécifiée.
- 2 points si la tortue dessine correctement.

Question 2 : Tortue colorée (héritage)

Nous souhaitons maintenant ajouter de la couleur.

Pour cela, l'état de la tortue est enrichi d'une information de couleur. Nous choisissons pour type la classe `javafx.scene.paint.Color`, disponible dans la partie graphique de l'API standard Java.

L'utilisation de JavaFX nécessite la version de Java 8 d'Oracle ou supérieure. JavaFX ne fonctionnera pas forcément avec OpenJDK. Assurez-vous donc que :

- le projet est bien configuré pour Java 8 ou supérieur : clic droit sur le projet « Properties > Java Compiler » (ou menu « Window > Preferences > Java > Compiler » pour les préférences globales) ;
- la VM est bien configurée : « Window > Preferences > Java > Installed JREs » et choisir une version en 1.8 d'Oracle ou supérieure ;
- en cas d'erreur « *Access restriction: The type Color is not accessible due to restriction on required library [...]/lib/ext/jfxrt.jar* », une solution consiste à changer la configuration de la bibliothèque système avec la manipulation suivante : clic droit sur le projet, « Properties > Java Build Path > Libraries > JRE System Library > Edit... > Workspace default JRE > Finish ».

Notre nouvelle tortue doit posséder une méthode permettant de changer la couleur :

```
public void setColor(Color color);
```

Une première étape consiste à **créer une nouvelle interface, `IColorTurtle`**, qui enrichit `ITurtle` par **héritage** en ajoutant cette méthode. Une deuxième étape consiste à **fournir une implantation, `ColorTurtle`**, de l'interface `IColorTurtle`. Afin de ne pas repartir de rien, cette classe **héritera** de la classe `Turtle` développée à la question précédente.

Quand elle est créée, une tortue colorée a pour couleur initiale le noir, c'est à dire `Color.BLACK`.

Le format de sortie est légèrement changé, puisqu'on y ajoute la couleur en fin de ligne :

```
x1 y1 x2 y2 couleur
```

Le changement de la sortie pourra être réalisé en redéfinissant la méthode `draw` de la classe `Turtle` dont on hérite.

L'affichage de la couleur se fera en utilisant la méthode `toString` de la classe `Color`, qui retourne une représentation sous forme d'entier hexadécimal de la couleur.

Voici, à gauche, un exemple de code et, à droite, le résultat attendu de l'exécution :

```
IColorTurtle t = new ColorTurtle();
t.setColor(Color.RED);
t.move(10);
t.turn(90);
t.setColor(Color.BLUE);
t.move(20);
```

```
0 0 0 10 0xff0000ff
0 10 20 10 0x0000ffff
```

Là encore, respectez bien la syntaxe demandée pour la sortie.

À fournir :

- un fichier `pobj/tme6/IColorTurtle.java`
- un fichier `pobj/tme6/ColorTurtle.java`

Attention : la classe `Turtle` que vous avez livrée pour la question précédente sera également utilisée dans la correction de cette question, puisque `CorlorTurtle` hérite de `Turtle`.

```
package pobj.tme6;

import javafx.scene.paint.Color;

public interface IColorTurtle extends ITurtle {
   public void setColor(Color color);
}
```

```
package pobj.tme6;

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; }
}
```

Fourni :

L'archive fournit l'exemple ci-dessus sous forme de test JUnit 4, `pobj.tme6.test.Question2Test` (fourni lors de l'examen), ainsi que les tests utilisés pour la notation (non fournis durant l'examen).

Notation : 2,5 points

- 1 point si `ColorTurtle` et `IColorTurtle` compilent et respectent bien les relations d'héritage demandées.
- 1,5 points si la tortue dessine correctement.

Question 3 : Motif adaptateur

Supposons qu'un programmeur, Alice, ait développé des nouvelles implantations de l'interface `ITurtle`. De son côté, Bob a développé une application en se basant sur l'interface `IColorTurtle`. Bob souhaite maintenant tester son application avec les implantations d'Alice. Malheureusement, l'application de Bob nécessite une tortue ayant l'interface `IColorTurtle` et ne fonctionnera pas avec une tortue ayant l'interface `ITurtle`. Il est hors de question de modifier toutes les classes d'Alice ou de dériver, pour chaque classe d'Alice, une nouvelle classe qui y ajoute l'interface `IColorTurtle` : il y en a bien trop.

Pour résoudre ce problème sans avoir à reprogrammer les implantations d'Alice et Bob, nous allons développer un adaptateur, c'est à dire une classe unique, capable de transformer toute tortue d'interface `ITurtle` en une tortue d'interface `IColorTurtle`.

Nous demandons donc de créer une classe `ColorTurtleAdapter` qui a les caractéristiques suivantes :

- la classe obéit à l'interface `IColorTurtle` ;
- son constructeur prend un objet `ITurtle` en argument ;
- la classe implante les méthodes de `ITurtle` par délégation à cet objet.

Par ailleurs, la classe `ColorTurtleAdapter` ignorera les ordres de changement de couleur (`setColor`), elle et n'affichera pas de couleur lors de ses déplacements : la sortie est donc similaire à celle demandée à la question 1.

Par exemple, le code de gauche aura la même sortie que le code de droite :

```
ITurtle t = new Turtle();
IColorTurtle ct = new ColorTurtleAdapter(t);
ct.setColor(Color.RED);
ct.turn(90);
ct.move(10);
```

```
ITurtle t = new Turtle();
t.turn(90);
t.move(10);
```

Attention : nous demandons que l'adaptateur implante les méthodes de `IColorTurtle` par délégation, et non par héritage. Donc, `ColorTurtleAdapter` ne doit pas hériter des classes `Turtle` ou `ColorTurtle` des questions précédentes : sa classe parent est `Object`.

À fournir :

Un fichier `pobj/tme6/ColorTurtleAdapter.java`

La classe `Turtle` et l'interface `IColorTurtle` que vous avez livrées pour les questions 1 et 2 seront également compilées dans la correction de cette question.

```java
package pobj.tme6;

import javafx.scene.paint.Color;

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) {
      // rien
   }
}
```

Fourni :

L'archive fournit l'exemple ci-dessus sous forme de test JUnit 4 (fourni lors de l'examen), ainsi que les tests utilisés pour la notation (non fournis durant l'examen).

Notation : 2,5 points

- 1 point si `ColorTurtleAdatper` compile et respecte bien les relations d'héritage demandées.
- 1,5 point si l'adaptateur fonctionne correctement.

Question 4 : Sortie paramétrique (motif stratégie)

Dans les questions précédentes, l'exécution de la tortue a pour effet de décrire les traits à dessiner sous forme de texte. Nous souhaitons maintenant proposer des sorties alternatives.

Plutôt que de créer une implantation différente de `IColorTurtle` pour chaque nouveau type de sortie, nous proposons de définir une fois pour toutes une classe tortue qui prend en argument un objet précisant comment doivent être dessinés les traits.

Une première étape consiste à **définir l'interface `IContext` des objets sachant dessiner des traits colorés**, de la manière suivante :

```
package pobj.tme6;

import javafx.scene.paint.Color;

public interface IContext {

   public void drawLine(int x1, int y1, int x2, int y2, Color color);

}
```

Une classe implante l'interface en programmant, dans la méthode `drawLine`, l'effet souhaité à chaque fois qu'une tortue trace une ligne de couleur `color` des coordonnées `x1` et `y1` aux coordonnées `x2` et `y2`.

Une deuxième étape consiste à **implanter une classe générique de tortue `ContextTurtle`** qui implante l'interface `IColorTurtle`. Son constructeur prend en argument un objet de type `IContext`. Enfin, sa méthode `move` délègue à l'objet `IContext` le tracé de lignes : à chaque fois que la tortue avance avec le crayon baissé, la méthode `drawLine` est appelée au lieu d'afficher la ligne avec `System.out.println`.

Comme pour la question 2, puisque seul le format de sortie change, vous pourrez hériter de la classe `ColorTurtle` et redéfinir uniquement sa méthode `draw`.

Une troisième étape consiste à **fournir des implantations variées de `IContext`**. Nous demandons deux implantations différentes :

- `PrintContext` simule le comportement original de la question 2, c'est à dire que chaque appel à la méthode `drawLine` affiche sur une ligne `x1 y1 x2 y2 color` avec `System.out.println`.
- `BoundContext` n'affiche rien à l'écran, mais calcule les coordonnées minimales et maximales du dessin au fur et à mesure que l'on ajoute des lignes. Une fois le dessin terminé, ces bornes sont accessibles grâce aux méthodes publiques suivantes de la classe `BoundContext` :

```
public int getMinX();
public int getMinY();
public int getMaxX();
public int getMaxY();
```

Ainsi, le code de gauche donnera l'affichage indiqué à droite :

```
BoundContext c = new BoundContext();
IColorTurtle t = new ContextTurtle(c);
t.move(100);
t.turn(90);
t.move(100);
System.out.println(c.getMinX());
System.out.println(c.getMinY());
System.out.println(c.getMaxX());
System.out.println(c.getMaxY());
```

```
0
0
100
100
```

À fournir :

- un fichier `pobj/tme6/ContextTurtle.java`
- un fichier `pobj/tme6/PrintContext.java`
- un fichier `pobj/tme6/BoundContext.java`

```java
package pobj.tme6;

public class ContextTurtle extends ColorTurtle {

   protected 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());
   }

}
```

```java
package pobj.tme6;

import javafx.scene.paint.Color;

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);
   }

}
```

```java
package pobj.tme6;

```

```java
import javafx.scene.paint.Color;

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;
   }

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

}
```

Les classes et interfaces `Turtle`, `IColorTurtle`, `ColorTurtle` que vous avez livrées pour les questions 1 et 2 seront également compilées dans la correction de cette question.

Fourni :

L'archive fournit l'interface `IContext` ainsi que l'exemple ci-dessus sous forme de test JUnit 4 (fourni lors de l'examen), ainsi que les tests utilisés pour la notation (non fournis durant l'examen).

En bonus, vous trouverez dans le package `pobj.tme6.extra` de l'archive une classe `JFXContext` implantant `IContext` et qui affiche réellement le dessin de la tortue sur l'écran, ainsi que deux programmes d'exemple `MaisonMain` et `SpiraleMain`, exécutables dans Eclipse par le menu contextuel du bouton droit, option « Run As > Java Application ».

Notation : 4 points

– 1 point si la classe `PrintContext` compile et fonctionne seule (test unitaire).
– 1 point si la classe `BoundContext` compile et fonctionne seule (test unitaire).
– 1 point si la classe `ContextTurtle` compile et fonctionne seule (test unitaire).
– 1 point si nos tests combinant ces trois classes fonctionnent (test d'intégration).

Question 5 : Réification des commandes (motif commande)

Dans cette question, nous allons créer des **objets** représentant des **commandes** à faire exécuter par des tortues, comme par exemple `move 10`, `turn 90`, ou encore `up`. En pratique, il existe une classe de commande pour chaque méthode de `IColorTurtle`, et chaque objet stocke, en attribut, les paramètres de la méthode (`length` pour une commande `move`, `angle` pour une commande `turn`, etc.).

Une fois créé, un objet-commande peut être exécuté sur une tortue arbitraire. Les classes des commandes obéissent donc toutes à l'interface `ICommand` suivante :

```
package pobj.tme6;

public interface ICommand {

   public void execute(IColorTurtle turtle);

}
```

où un appel à `execute` sur un objet-commande avec une tortue `turtle` en argument exécute la commande par délégation, en appelant la méthode correspondante de `turtle`.

Cette question vous demande de **fournir des classes `CommandMove`, `CommandSetColor`, `CommandTurn`, `CommandUp` et `CommandDown` implantant l'interface `ICommand`**, et correspondant respectivement aux commandes `move`, `setColor`, `turn`, `up` et `down`.

Ainsi le programme de gauche et le programme de droite auront le même effet :

```
IColorTurtle t = new ColorTurtle();
ICommand cmd = new CommandMove(10);
cmd.execute(t);
cmd.execute(t);
```

```
IColorTurtle t = new ColorTurtle();
t.move(10);
t.move(10);
```

L'exemple montre qu'une même commande peut être exécutée plusieurs fois.

À fournir :

– un fichier `pobj/tme6/CommandMove.java`
– un fichier `pobj/tme6/CommandTurn.java`
– un fichier `pobj/tme6/CommandUp.java`
– un fichier `pobj/tme6/CommandDown.java`
– un fichier `pobj/tme6/CommandSetColor.java`

```
package pobj.tme6;

public class CommandMove implements ICommand {

   private int length;

```

```
   public CommandMove(int length) {
      this.length = length;
   }

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

}
```

```
package pobj.tme6;

public class CommandTurn implements ICommand {

   private int angle;

   public CommandTurn(int angle) {
      this.angle = angle;
   }

   @Override
   public void execute(IColorTurtle turtle) {
      turtle.turn(angle);
   }

}
```

```
package pobj.tme6;

public class CommandUp implements ICommand {

   @Override
   public void execute(IColorTurtle turtle) {
      turtle.up();
   }

}
```

```
package pobj.tme6;

public class CommandDown implements ICommand {

   @Override
   public void execute(IColorTurtle turtle) {
      turtle.down();
   }

}
```

```java
package pobj.tme6;

import javafx.scene.paint.Color;

public class CommandSetColor implements ICommand {

   private Color color;

   public CommandSetColor(Color color) {
      this.color = color;
   }

   @Override
   public void execute(IColorTurtle turtle) {
      turtle.setColor(color);
   }

}
```

Les classes et interfaces que vous avez livrées pour les questions 1 et 2 seront également compilées dans la correction de cette question.

Fourni :

L'archive fournit l'interface `ICommand` ainsi que l'exemple ci-dessus sous forme de test JUnit 4 (fourni lors de l'examen), ainsi que les tests utilisés pour la notation (non fournis durant l'examen).

Notation : 3 points

- 0,5 point pour chacune des 5 classes demandées, si elles compilent et fonctionnent (test unitaire).
- 0,5 point si nos tests combinant toutes les classes fonctionnent (test d'intégration).

Question 6 : Liste de commandes (motif composite)

Dans les questions précédentes, le dessin était affiché au fur et à mesure des appels aux méthodes de la tortue. Dans cette question, nous montrons comment différer l'exécution des commandes grâce aux objets-commandes définis à la question précédente, et à une tortue spéciale qui, au lieu d'exécuter ses commandes directement, accumule des objets-commandes dans une liste.

Vous commencerez par **définir un conteneur `CommandList`** proposant des méthodes de signature suivante :

```java
public void addCommand(ICommand command);
public void execute(IColorTurtle turtle);
```

capables respectivement :

- d'accumuler une succession de commandes ;
- de faire exécuter la liste des commandes accumulées par une tortue (dans le même ordre que l'ordre d'ajout dans la liste).

Votre classe `CommandList` devra par ailleurs **implanter l'interface** `ICommand` ; nous avons en effet pris soin ci-dessus de demander la présence d'une méthode `execute` de signature compatible avec `ICommand`. Il s'agit donc du motif composite.

Ensuite, vous **écrirez une classe** `SaveTurtle` qui implante `IColorTurtle`, mais qui stocke dans un objet `CommandList` les commandes au lieu de les exécuter. La classe `SaveTurtle` proposera une méthode :

```
public CommandList getCommand();
```

qui retourne la liste des commandes accumulées. Cette liste pourra alors être exécutée par une autre tortue.

Ainsi, le programme suivant :

```
SaveTurtle t = new SaveTurtle();
t.move(10); t.turn(90);
t.move(10); t.turn(90);
t.move(10); t.turn(90);
t.move(10); t.turn(90);
ICommand square = t.getCommand();

IColorTurtle out = new ColorTurtle();
for (int i=0; i<3; i++) {
  square.execute(out);
  out.up();
  out.move(30);
  out.down();
}
```

dessinera trois carrés côte à côte, en réutilisant trois fois la même liste de commandes `square`, créée une fois pour toutes par la tortue `t`.

À fournir :

- un fichier `pobj/tme6/CommandList.java`
- un fichier `pobj/tme6/SaveTurtle.java`

```
package pobj.tme6;

import java.util.List;
import java.util.Vector;

public class CommandList implements ICommand {

   private List<ICommand> commands;

   public CommandList() {
      commands = new Vector<ICommand>();
   }

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

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

}
```

```
package pobj.tme6;

import javafx.scene.paint.Color;

public class SaveTurtle implements IColorTurtle {

   private CommandList buffer;

   public SaveTurtle() {
      buffer = new CommandList();
   }

   public CommandList getCommand() {
      return buffer;
   }

   @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));
   }

}
```

Les classes et interfaces que vous avez livrées pour les questions 1, 2 et 5 seront également compilées dans la correction de cette question.

Fourni :

L'archive fournit l'exemple ci-dessus sous forme de test JUnit 4 (fourni lors de l'examen), ainsi que les tests utilisés pour la notation (non fournis durant l'examen).

Notation : 3 points

- 1 point si la classe `CommandList` compile et fonctionne seule (test unitaire).
- 1 point si la classe `SaveTurtle` compile et fonctionne seule (test unitaire).
- 1 point si nos tests combinant ces classes fonctionnent (test d'intégration).

Question 7 : Manipulation des commandes

Dans cette dernière question, nous allons modifier nos dessins en créant une classe capable de transformer une liste de commandes en une autre.

La transformation étudiée est la substitution, dans une liste de commandes, de toutes les commandes `CommandMove` par une commande arbitraire passée en argument. En particulier, les `CommandMove` peuvent être substituées par une liste de commandes `CommandList`. Cette opération permet de facilement générer des dessins fractals.

Vous fournirez une **classe `Substitution`, ayant une méthode `substitute` statique** :

```
static public ICommand substitute(ICommand org, ICommand subst);
```

qui parcourt récursivement `org` et construit une nouvelle commande identique à `org` où toutes les `CommandMove` sont remplacées par `subst`.

Voici, à gauche, un exemple de code et, à droite, le résultat attendu de l'exécution :

```
CommandList l1 = new CommandList();
l1.addCommand(new CommandMove(10));
l1.addCommand(new CommandTurn(90));
l1.addCommand(new CommandMove(10));

CommandList l2 = new CommandList();
l2.addCommand(new CommandSetColor(Color.RED));
l2.addCommand(new CommandMove(10));
l2.addCommand(new CommandSetColor(Color.BLUE));
l2.addCommand(new CommandMove(10));

ICommand c = Substitution.substitute(l2, l1);

c.execute(new ColorTurtle());
```

```
0 0 0 10 0xff0000ff
0 10 10 10 0xff0000ff
10 10 20 10 0x0000ffff
20 10 20 0 0x0000ffff
```

Vous êtes libres d'adopter la technique de votre choix pour implanter `substitute`.

Nous vous proposons néanmoins une technique permettant de réutiliser au maximum ce qui a été fait aux questions précédentes. Observons d'abord que, en faisant exécuter une liste de commandes `org` par la tortue `SaveTurtle` de la question précédente, la résultat de `getCommand` sera une nouvelle liste contenant exactement les mêmes commandes que `org`. Il suffit alors de modifier le comportement de la méthode `move` de `SaveTurtle` afin que, au lieu d'insérer un objet-commande `CommandMove` dans sa liste, celle-ci insère la commande `subst`. Vous devrez donc définir une nouvelle classe héritant de `SaveTurtle`. Elle s'utilisera de la manière suivante : nous créons une instance en lui passant la commande `subst` à la construction, puis nous lui faisons exécuter la commande `org`, enfin,

nous appelons `getCommand` qui donne, comme résultat, la liste de commandes après substitution recherchée.

À fournir :

– un fichier `pobj/tme6/Substitution.java`

```java
package pobj.tme6;

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();
   }
}
```

Les classes et interfaces que vous avez livrées pour les questions 1, 2, 5 et 6 seront également compilées dans la correction de cette question.

Fourni :

L'archive fournit l'exemple ci-dessus sous forme de test JUnit 4 (fourni lors de l'examen), ainsi que les tests utilisés pour la notation (non fournis durant l'examen).

En bonus, vous trouverez dans le package `pobj.tme6.extra` de l'archive une copie de la classe `JFXContext` de la question 4 et un exemple `KochMain` qui affiche une belle fractale à l'écran.

Notation : 2 points

– 1 point si la classe `Substitution` compile et fonctionne seule (test unitaire).

– 1 point si nos test combinant les deux dernières questions fonctionnent (test d'intégration).

Rendu de TME (OBLIGATOIRE)

Pour le rendu de ce TME particulier, nous demandons de placer sur le serveur GitLab uniquement le fichier `upload.zip`. Faites ensuite, comme d'habitude, un *tag*. Assurez-vous que vous respectez bien les chemins de fichiers prescrits, et que la notation automatique tient bien compte de vos classes.

Dans la partie « Release notes » du *tag*, vous indiquerez la note attribuée par la correction automatique.