La programmation orientée objet

Exercice 1 : les objets mathématiques (extrait des médians A12 et A13)

On désire écrire une application de type Matlab qui permet de manipuler des objets mathématiques. On s'intéresse ici aux nombres complexes et aux polynômes du second degré.

1. Ecrire une classe *Complexe* qui représente un nombre complexe. Cette classe devra suivre la spécification suivante :

- Un nombre complexe sera caractérisé par sa partie réelle et sa partie imaginaire.
- On pourra obtenir le conjugué d'un nombre complexe.
- On pourra calculer le module d'un nombre complexe.
- On pourra multiplier un nombre complexe par un autre et stocker le résultat dans le premier.
- On pourra ajouter un nombre complexe à un autre et stocker le résultat dans le premier.

Pour rappel, voilà quelques éléments sur les complexes :

- Addition : $(a+bi)+(c+di)=(a+c)+(b+d)i$
- Multiplication : $(a+bi)(c+di)=(ac-bd)+(ad+bc)i$.
- Conjugué : $\overline{a+bi}=a-bi$;
- Module : $|a+bi|=\sqrt{a^2+b^2}$

2. Implémenter les méthodes *String toString()* pour produire tous les affichages relatifs à un objet de cette classe et *boolean equals (Object o)* pour la comparaison entre objets.

3. Tester la manipulation d'objets par références à travers différents tests faits dans la méthode *main()*. Mettre en évidence le fait que le programmeur ne manipule pas les objets directement mais des références sur ces objets.

4. Implémenter les deux méthodes suivantes :

- *ajouterReference(Complexe c)* : additionne l'objet courant au paramètre *c* et place le résultat dans *c*.
- *ajouterReference(int pr, int pi)* : additionne l'objet courant aux paramètres *pr* et *pi* et place le résultat dans *pr* et *pi*.

Conclure quant au passage de paramètres par valeur et par référence dans le langage Java.

```
public class Complexe {

    private double partieReelle;
    private double partieImaginaire;

    public Complexe (double pr, double pi) {
      this.partieReelle = pr;
      this.partieImaginaire = pi;
    }

    public double module() {
      return Math.sqrt(Math.pow(this.partieReelle, 2) +
Math.pow(this.partieImaginaire, 2));
    }

    public void ajouter (Complexe c) {
      this.partieReelle += c.partieReelle;
      this.partieImaginaire += c.partieImaginaire;
    }

    public void multiplier (Complexe c) {
```

```
      double pr = this.partieReelle * c.partieReelle -
this.partieImaginaire * c.partieImaginaire;
      double pi = this.partieReelle * c.partieImaginaire +
this.partieImaginaire * c.partieReelle;

      this.partieReelle = pr;
      this.partieImaginaire = pi;
    }

    public Complexe conjugue () {
      return new Complexe (this.partieReelle, -this.partieImaginaire);
    }

    public void ajouterReference(Complexe c) {
      c.partieReelle += this.partieReelle;
      c.partieImaginaire += this.partieImaginaire;
    }

    public void ajouterReference(int pr, int pi) {
      pr += this.partieReelle;
      pi += this.partieImaginaire;
    }

    public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append("(");
      sb.append(this.partieReelle);
      sb.append(" + ");
      sb.append(this.partieImaginaire);
      sb.append("i)");
      return sb.toString();
    }

    public static void main(String args[]) {
      Complexe c1 = new Complexe(1,2);
      System.out.println(c1);
      Complexe c2 = new Complexe (1,1);
      c1.ajouter(c2);
      System.out.println(c1);

      Complexe c3 = new Complexe(-1, -1);
      c1.multiplier(c3);
      System.out.println(c1);

      c1 = new Complexe(1,2);
      c1.ajouterReference(c2);
      System.out.println(c1);
      System.out.println(c2);

      int pr = 2;
      int pi = 2;
      c1.ajouterReference(pr, pi);
      System.out.println("(" + pr + ", " + pi + ")");
    }
}
```

Exercice 2 : La classe polynôme de degré deux

On souhaite maintenant intégrer la classe *Complexe* dans le calcul des racines d'un polynôme de second degré. Pour ce faire, produire une classe *PolynomeDegreDeux* sur la base de la solution de l'exercice 1 du TD précédent. Donner le code d'une méthode appelée *getRoots*() qui permet de retourner les racines du polynôme quelle que soit la valeur de son discriminant. Les racines seront

alors des objets de la classe *Complexe*. Implanter les méthodes *String toString()* pour produire tous les affichages relatifs au polynôme et *boolean equals (Object o)* pour la comparaison entre objets.

```
public class PolynomeDegreDeux {

    // x2, x1, x0
    private int[] polynome;

    public PolynomeDegreDeux(int x2, int x1, int x0) {
      polynome = new int[3];
      polynome[0] = x2;
      polynome[1] = x1;
      polynome[2] = x0;
    }

    public double delta() {
      return polynome[1] * polynome[1] - 4 * polynome[0] * polynome[2];
    }

    public boolean hasComplexRoots() {
      return (this.delta() < 0);
    }

    public boolean hasDoubleRoot() {
      return (this.delta() == 0);
    }

    public boolean hasRealRoots() {
      return (this.delta() > 0);
    }

    private double moinsBSur2A () {
      return -polynome[1] / (2 * polynome[0]);
    }

    private double racineDeltaSur2A () {
      return Math.sqrt(Math.abs(this.delta())) / (2 * polynome[0]);
    }

    public int getX2() {
      return polynome[0];
    }

    public int getX1() {
      return polynome[1];
    }

    public int getX0() {
      return polynome[2];
    }

    public boolean equals(Object o) {
      if (o instanceof PolynomeDegreDeux) {
          PolynomeDegreDeux p = (PolynomeDegreDeux)o;
          return ((p.getX2() == this.getX2()) & (p.getX1() == this.getX1())
& (p.getX0() == this.getX0()));
      } else {
          return false;
      }

    }
```

```
    public Complexe[] getRoots() {
      Complexe[] racines;
      if (this.hasDoubleRoot()) {
          racines = new Complexe[1];
          racines[0] = new Complexe (this.moinsBSur2A(), 0);
      } else {
          racines = new Complexe[2];
          if (this.hasRealRoots()) {
            racines[0] = new Complexe(this.moinsBSur2A() +
this.racineDeltaSur2A(), 0);
            racines[1] = new Complexe(this.moinsBSur2A() -
this.racineDeltaSur2A(), 0);
          } else {
            racines[0] = new Complexe(this.moinsBSur2A(),
this.racineDeltaSur2A());
            racines[1] = racines[0].conjugue();
          }
      }
      return racines;
    }

    public String toString() {
      return new String(polynome[0] + "x2 + " + polynome[1] + "x + " +
polynome[2]);
    }

    public static void main(String[] args) {

      PolynomeDegreDeux p1 = new PolynomeDegreDeux (1, -1, 1);
      PolynomeDegreDeux p2 = new PolynomeDegreDeux (1, 0, 1);
      PolynomeDegreDeux p3 = new PolynomeDegreDeux (1, -1, 1);

      // A faire pour p2 et p3
      if (p1.hasRealRoots()) {
          System.out.print("Discrimant positif : ");
      } else if (p1.hasDoubleRoot()) {
          System.out.print("Discrimant nul : ");
      } else {
          System.out.print("Discrimant nÈgatif : ");
      }
      System.out.println(p1.delta());

      System.out.println("Racines : " + p1.getRoots());

      // Tests de la méthode toString()
      System.out.println(p1);
      System.out.println(p2);
      System.out.println(p3);

      // Tests de la méthode equals()
      if (p1.equals(p2)) {
          System.out.println("P1 = P2");
      } else {
          System.out.println("P1 != P2");
      }

      if (p1.equals(p3)) {
          System.out.println("P1 = P3");
      } else {
          System.out.println("P1 != P3");
      }
    }
}
```

Exercice 3 : Implantation des lampes

Implanter les classes du TD UML sur la lampe à puissance variable. On ajoutera à l'implantation le fait que le nombre de lampes de pleine puissance puisse être variable. Par exemple, un système peut être composée d'une lampe de 50W et deux de 100W et un autre d'une lampe de 50W et quatre de 100W. La fonction de commutation doit être identique pour ces deux systèmes.

```
package fr.utt.sit.lo02.td2.lampe;

public class Lampe {

    public final static int PUISSANCE_STANDARD = 100;

    private int puissance;
    private boolean allumee;

    public Lampe(int puissance) {
      this.puissance = puissance;
      allumee = false;
    }

    public int getPuissance() {
        return puissance;
    }

    public boolean isAllumee() {
        return allumee;
    }

    public void allumer() {
        this.allumee = true;
    }

    public void eteindre() {
        this.allumee = false;
    }

    public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append("Lampe (");
      sb.append("Puissance : " + puissance);
      sb.append("\t");
      sb.append("Allumée : " + allumee);
      sb.append(")\n");
      return sb.toString();
    }
}
```

```
package fr.utt.sit.lo02.td2.lampe;

public class Commutateur {

    private Lampe[] lampes;
    private int etat;
    private int nombreEtats;

    public Commutateur (int nombreLampes) {

      etat = 0;
```

```
      nombreEtats = nombreLampes * 2;
      lampes = new Lampe[nombreLampes];

      Lampe lampeFaible = new Lampe (Lampe.PUISSANCE_STANDARD / 2);
      lampes[0] = lampeFaible;

      for (int i = 1; i < nombreLampes; i++) {
          Lampe lampeStandard = new Lampe (Lampe.PUISSANCE_STANDARD);
          lampes[i] = lampeStandard;
      }
    }

    public void commuter() {
      etat = (etat + 1) % nombreEtats;

      int puissanceRequise = etat * Lampe.PUISSANCE_STANDARD / 2;

      // Allumage/eteignage de la lampe faible
      Lampe lampeFaible = lampes[0];

      if ((puissanceRequise % Lampe.PUISSANCE_STANDARD) != 0) {
          lampeFaible.allumer();
      } else {
          lampeFaible.eteindre();
      }

      // Allumage/Eteignage des autres lampes
      for (int i = 1; i <= (puissanceRequise / Lampe.PUISSANCE_STANDARD);
i++) {
          Lampe lampe = lampes[i];
          lampe.allumer();
      }

      for (int i = Math.max((puissanceRequise / Lampe.PUISSANCE_STANDARD) +
1, 1); i < lampes.length; i++) {
          Lampe lampe = lampes[i];
          lampe.eteindre();
      }
    }

    public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append("Commutateur (");
      sb.append("Etat : " + etat);
      sb.append(")\n");
      for (int i = 0; i < lampes.length; i++) {
          sb.append(lampes[i]);
      }
      return sb.toString();
    }
}
```

```
package fr.utt.sit.lo02.td2.lampe;

public class Interrupteur {
    private Commutateur commutateur;

    public Interrupteur () {
      commutateur = new Commutateur(5);
    }
```

```
    public void appuyer() {
      commutateur.commuter();
    }

    public String toString() {
      StringBuffer sb = new StringBuffer();
      sb.append("Interrupteur\n");
      sb.append(commutateur);
      return sb.toString();
    }

    public static void main (String[] args) {

      Interrupteur inter = new Interrupteur();

      for (int i = 0; i < 20; i++) {
          inter.appuyer();
          System.out.println(inter);
      }
    }
}
```

Exercices à faire en temps hors-encadrement (THE)

Exercice 4 : Méthodes statiques : le singleton (implémentation du patron vu en TD UML)

Le singleton est un patron de conception qui permet de ne créer qu'une seule instance d'une classe. Pour cela, on utilise une méthode de classe appelée *getInstance()* pour récupérer l'unique instance de la classe.

Illustrer le fonctionnement du singleton avec une classe de logging. Cette classe permettra d'afficher à l'écran des messages formatés pour le log. On ajoutera en particulier la date à tout message loggé.

```
package fr.utt.sit.lo02.td2.singleton;

import java.util.Date;

public class Logger {

    public final static byte DEBUG = 0;
    public final static byte INFO = 1;
    public final static byte WARN = 2;
    public final static byte ERROR = 3;
    public final static byte FATAL = 4;

    public final static byte OUTPUT_STREAM = 0;
    public final static byte ERROR_STREAM = 1;

    private final static String[] LEVEL_NAMES = {"DEBUG", "INFO", "WARN",
"ERROR", "FATAL"};

    private static Logger logger = null;

    private byte level;
    private Date date;
    private String separator;
    private byte output;

    public static Logger getInstance() {
      if (logger == null) {
          logger = new Logger();
      }

      return logger;
    }

    private Logger() {
      level = Logger.INFO;
      date = new Date();
      separator = "\t";
      output = Logger.OUTPUT_STREAM;
    }

    public void setLevel(byte newLevel) {
      if ((newLevel < Logger.DEBUG) | (newLevel > Logger.FATAL)) {
          this.error("Niveau de log invalide : " + level);
      } else {
          level = newLevel;
      }
    }

    public byte getLevel() {
```

```
    return level;
  }

  public String getStringLevel() {
    return Logger.LEVEL_NAMES[level];
  }


  public void setOutputStream(byte outputStream) {
    if (outputStream == Logger.OUTPUT_STREAM) {
        output = outputStream;
    } else if (outputStream == Logger.ERROR_STREAM) {
        output = outputStream;
    } else {
        this.error("Sortie de log invalide: " + outputStream);
    }
  }

  public void debug(String message) {
    if (level <= Logger.DEBUG) {
        this.log(Logger.DEBUG, message);
    }
  }

  public void info(String message) {
    if (level <= Logger.INFO) {
        this.log(Logger.INFO, message);
    }
  }

  public void warn(String message) {
    if (level <= Logger.WARN) {
        this.log(Logger.WARN, message);
    }
  }

  public void error(String message) {
    if (level <= Logger.ERROR) {
        this.log(Logger.ERROR, message);
    }
  }

  public void fatal(String message) {
    if (level <= Logger.FATAL) {
        this.log(Logger.FATAL, message);
    }
  }

  private void log(byte messageLevel, String message) {
    String formattedMessage = new String(
          date.toString() + separator +
          Logger.LEVEL_NAMES[messageLevel] + separator +
          message);
    switch (output) {
    case Logger.OUTPUT_STREAM:
        System.out.println(formattedMessage);
        break;
    case Logger.ERROR_STREAM:
        System.err.println(formattedMessage);
        break;
    default:
```

```
            System.err.println("Logger: invalid output: " + output);
        }

    }

    public static void main(String[] args) {
        Logger logger = Logger.getInstance();
        logger.setLevel(Logger.WARN);
        logger.setOutputStream(Logger.OUTPUT_STREAM);
        logger.debug("test debug");
        logger.info("test info");
        logger.warn("test warn");
        logger.error("test error");
        logger.fatal("test fatal");
    }
}
```