IN201 Conception et Programmation Orientées Objet

TP : présentation et corrigés

Christophe Garion

DMIA – ISAE

License CC BY-NC-SA 3.0

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license (CC BY-NC-SA 3.0)

You are free to Share (copy, distribute and transmite) and to Remix (adapt) this work under the following conditions :

Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial – You may not use this work for commercial purposes.

Share Alike – If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

See `http://creativecommons.org/licenses/by-nc-sa/3.0/`.

Plan

1. **TP sur les classes et les objets**
   - Présentation
   - Utilisation de la classe `Point`
   - Implantation de la classe `Orbite`
   - Le CLASSPATH c'est pénible

2. **TP sur les tests unitaires et les associations**

3. **TP sur les interfaces**

4. **TP sur la spécialisation et l'héritage**

5. **TP sur le patron de conception observateur**

6. **TP sur les interfaces graphiques avec Swing**

7. **TP sur les exceptions**

Plan

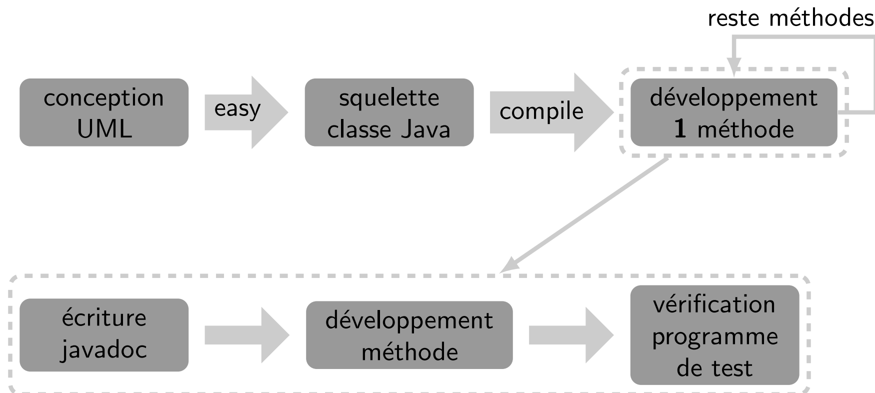
1. **TP sur les classes et les objets**
   - Présentation
   - Utilisation de la classe `Point`
   - Implantation de la classe `Orbite`
   - Le CLASSPATH c'est pénible
2. **TP sur les tests unitaires et les associations**
3. **TP sur les interfaces**
4. **TP sur la spécialisation et l'héritage**
5. **TP sur le patron de conception observateur**
6. **TP sur les interfaces graphiques avec Swing**
7. **TP sur les exceptions**

Espace de travail

Terminal compilation	**Terminal exécution**

- utilisez l'**historique** de bash pour éviter de retaper les commandes
- respectez les dossiers : `src` pour les sources, `classes` pour les bytecodes générés
- utilisez un éditeur de texte que vous maîtrisez pour les sources : `gedit`, `emacs`, `vi`

Le process à utiliser...



Plan

1. **TP sur les classes et les objets**
   - Présentation
   - Utilisation de la classe Point
   - Implantation de la classe Orbite
   - Le CLASSPATH c'est pénible
2. **TP sur les tests unitaires et les associations**
3. **TP sur les interfaces**
4. **TP sur la spécialisation et l'héritage**
5. **TP sur le patron de conception observateur**
6. **TP sur les interfaces graphiques avec Swing**
7. **TP sur les exceptions**

Création de la classe de test

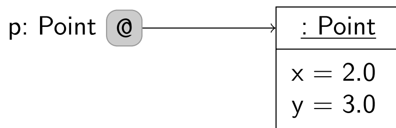
TestPoint.java

```java
class TestPoint {

    public static void main(String[] args) {

    }
}
```

Création d'un point et affichage

TestPoint.java

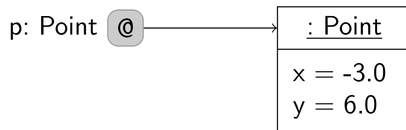
```
        Point p = new Point(2.0, 3.0);

        System.out.println("p apres creation : " + p);
```



Translation du point

TestPoint.java

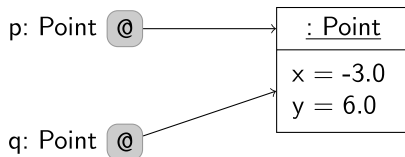
```
        p.translater(-5.0, 3.0);
        System.out.println("p apres translation : " + p);
```



Création d'un nouveau point et translation

TestPoint.java

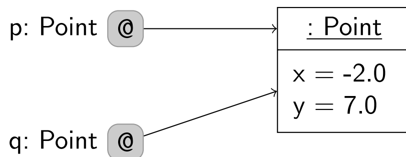
```
        Point q = p;

        q.translater(1.0, 1.0);
        System.out.println("p apres translation de q : " + p);
        System.out.println("q apres translation : " + p);
```



Création d'un nouveau point et translation

TestPoint.java

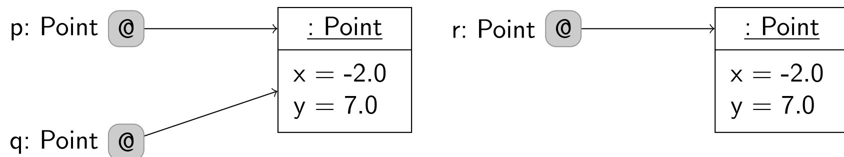
```
        Point q = p;

        q.translater(1.0, 1.0);
        System.out.println("p apres translation de q : " + p);
        System.out.println("q apres translation : " + p);
```



Clone d'un point

TestPoint.java

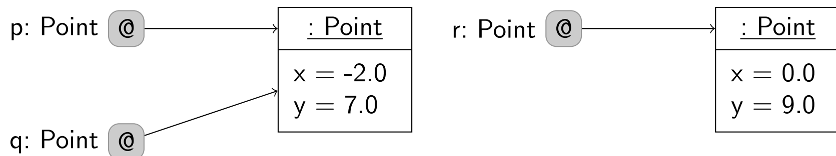
```
        Point r = p.clone();

        System.out.println("p apres copie : " + p);
        System.out.println("r apres copie : " + r);

        r.translater(2.0, 2.0);
        System.out.println("p apres translation de r : " + p);
        System.out.println("r apres translation de r : " + r);
```



Clone d'un point

TestPoint.java

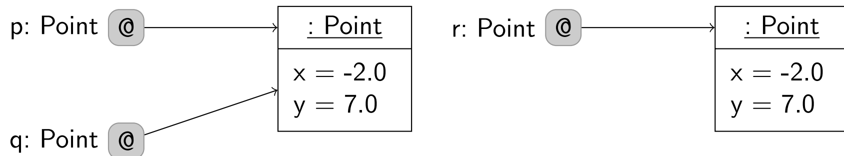
```
        Point r = p.clone();

        System.out.println("p apres copie : " + p);
        System.out.println("r apres copie : " + r);

        r.translater(2.0, 2.0);
        System.out.println("p apres translation de r : " + p);
        System.out.println("r apres translation de r : " + r);
```



Égalité avec ==

TestPoint.java

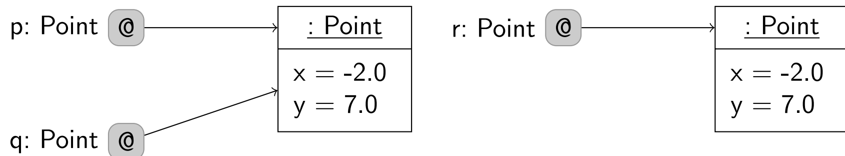
```
        r = p.clone();

        System.out.println("p == q ? " + (p == q));
        System.out.println("p == r ? " + (p == r));
```



Égalité avec equals

TestPoint.java

```
        System.out.println("p.equals(q) ? " + (p.equals(q)));
        System.out.println("p.equals(r) ? " + (p.equals(r)));
```



Plan

1. **TP sur les classes et les objets**
   - Présentation
   - Utilisation de la classe `Point`
   - Implantation de la classe `Orbite`
   - Le CLASSPATH c'est pénible
2. **TP sur les tests unitaires et les associations**
3. **TP sur les interfaces**
4. **TP sur la spécialisation et l'héritage**
5. **TP sur le patron de conception observateur**
6. **TP sur les interfaces graphiques avec Swing**
7. **TP sur les exceptions**

Quelques remarques

- construction du squelette de la classe → fastidieux, mais facile
- constructeurs : pas de difficulté particulière, attention à l'initialisation de l'instance de `Point`
  - ➥ utilisation de `clone` ou non ?
- méthodes de « calcul » : y réfléchir sur papier avant, développer tranquillement, aérer le code !
- comment tester l'implantation ?

Plan

1. **TP sur les classes et les objets**
   - Présentation
   - Utilisation de la classe `Point`
   - Implantation de la classe `Orbite`
   - Le CLASSPATH c'est pénible
2. **TP sur les tests unitaires et les associations**
3. **TP sur les interfaces**
4. **TP sur la spécialisation et l'héritage**
5. **TP sur le patron de conception observateur**
6. **TP sur les interfaces graphiques avec Swing**
7. **TP sur les exceptions**

Ces histoires de CLASSPATH...

Comme il existe une **variable d'environnement** `PATH` sur les systèmes UNIX qui permet de savoir où trouver les exécutables, le JDK dispose d'une variable d'environnement, le CLASSPATH, pour savoir où trouver les fichiers **`.class`** dont le compilateur ou la JVM ont besoin.

On modifie le CLASSPATH lors de l'appel à `javac` ou `java` en utilisant l'option `-cp`.

Ces histoires de CLASSPATH...

Comme il existe une **variable d'environnement** `PATH` sur les systèmes UNIX qui permet de savoir où trouver les exécutables, le JDK dispose d'une variable d'environnement, le CLASSPATH, pour savoir où trouver les fichiers **`.class`** dont le compilateur ou la JVM ont besoin.

On modifie le CLASSPATH lors de l'appel à `javac` ou `java` en utilisant l'option `-cp`.

Ainsi, l'appel

```
javac -d ../classes -cp .:../classes TestPoint.java
```

signifie :

- compiler le fichier `TestPoint.java` situé dans le répertoire courant
- mettre le résultat de la compilation (`TestPoint`**`.class`**) dans le répertoire `../classes`
- si le compilateur a besoin d'autres fichiers **`.class`** (`Point`**`.class`**), aller les chercher soit dans le répertoire courant (`.`), soit dans le répertoire `../classes`

Ces histoires de CLASSPATH. . .

Comme il existe une **variable d'environnement** `PATH` sur les systèmes UNIX qui permet de savoir où trouver les exécutables, le JDK dispose d'une variable d'environnement, le CLASSPATH, pour savoir où trouver les fichiers **`.class`** dont le compilateur ou la JVM ont besoin.

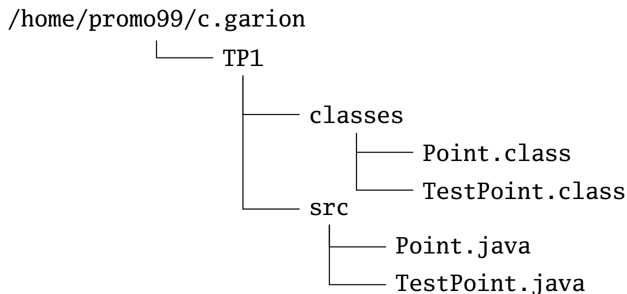
On modifie le CLASSPATH lors de l'appel à `javac` ou `java` en utilisant l'option `-cp`.

Remarque

Par défaut, le JDK connaît le chemin d'accès à toutes les classes de l'API Java (`String`, `Math` etc.).

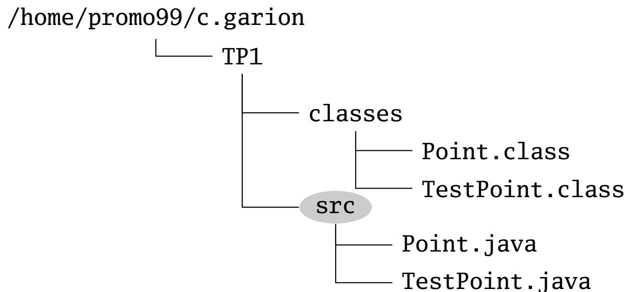
CLASSPATH : exemples

Considérons la hiérarchie suivante :



CLASSPATH : exemples

Considérons la hiérarchie suivante :

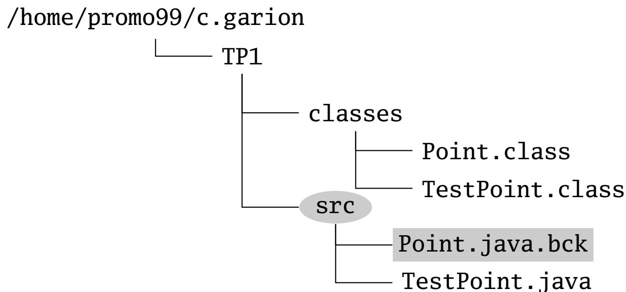


Si on est dans src, on peut compiler de la façon suivante :

```
javac -d ../classes -cp ../classes *.java
```

CLASSPATH : exemples

Considérons la hiérarchie suivante :

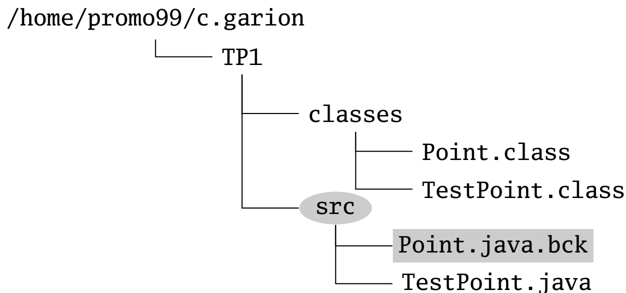


Si on renomme `Point.java`, on peut toujours compiler de la façon suivante (mais on ne recompile pas `Point`) :

```
javac -d ../classes -cp ../classes *.java
```

CLASSPATH : exemples

Considérons la hiérarchie suivante :

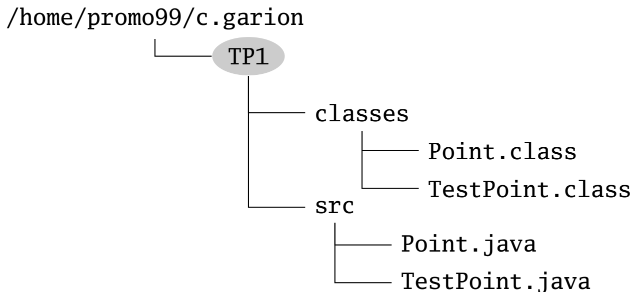


Par contre, on ne peut pas compiler comme cela (le compilateur ne trouve ni le source de Point, ni le *bytecode* associé) :

```
javac -d ../classes *.java
```

CLASSPATH : exemples

Considérons la hiérarchie suivante :

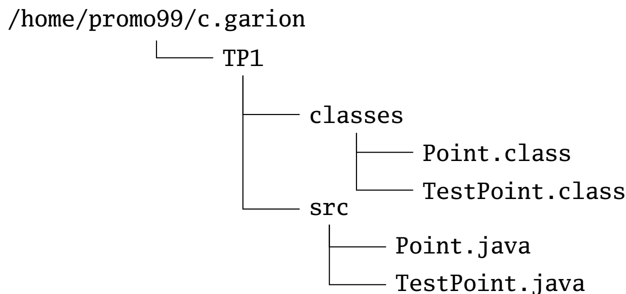


Depuis TP1, on peut exécuter TestPoint en précisant où se trouvent les *bytecodes* :

```
java -cp ./classes TestPoint
```

CLASSPATH : exemples

Considérons la hiérarchie suivante :



On peut même le faire depuis n'importe où :

```
java -cp ~/TP1/classes TestPoint
```

Plan

1. TP sur les classes et les objets
2. **TP sur les tests unitaires et les associations**
   - Présentation
   - Tester `FalseOrbite`
   - Implantation de `OrbiteDiscrete`
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Plan

1. TP sur les classes et les objets
2. **TP sur les tests unitaires et les associations**
   - Présentation
   - Tester `FalseOrbite`
   - Implantation de `OrbiteDiscrete`
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Élémentaire mon cher Watson...

Votre binôme a fait beaucoup de bêtises...

- il a copié la classe `Orbite`
- il a introduit **trois** erreurs
- il a perdu le code source

Élémentaire mon cher Watson...

Votre binôme a fait beaucoup de bêtises...

- il a copié la classe `Orbite`
- il a introduit **trois** erreurs
- il a perdu le code source

Conclusion

Changez de binôme...

Élémentaire mon cher Watson...

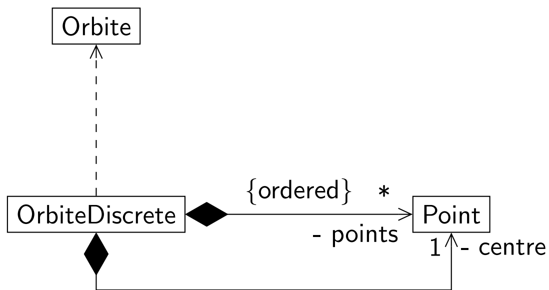
Votre binôme a fait beaucoup de bêtises...

- il a copié la classe `Orbite`
- il a introduit **trois** erreurs
- il a perdu le code source

Mise en œuvre du TP

- la classe s'appelle maintenant `FalseOrbite`
- écrire des tests JUnit pour trouver les erreurs
- on peut utiliser `Orbite` comme **oracle de test**

La classe `OrbiteDiscrete` : analyse



La classe OrbiteDiscrete : implantation

fr::isae::orbit::OrbiteDiscrete
- points: java::util::ArrayList<fr::isae::geometry::Point> - centre: fr::isae::geometry::Point
+ OrbiteDiscrete(o: Orbite, pasAngle: double) - OrbiteDiscrete(points: java::util::ArrayList<fr::isae::geometry::Point>, centre: fr::isae::geometry::Point) + getNbPoints(): int + getPoint(i: int): fr::isae::geometry::Point + getCentre(): fr::isae::geometry::Point + getCirconference(): double + getVecteurTangent(i: int): Point + translater(dx: double, dy: double) + homothetie(rapport: double) + clone(): OrbiteDiscrete + equals(o: OrbiteDiscrete): boolean + toString(): String

Plan

1. TP sur les classes et les objets
2. **TP sur les tests unitaires et les associations**
   - Présentation
   - Tester `FalseOrbite`
   - Implantation de `OrbiteDiscrete`
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Comment tester `FalseOrbite`

On peut utiliser les tests sur lesquels on avait travaillé durant la séance de cours.

On pouvait également utiliser les « vraies » orbites qui étaient disponibles via la classe `Orbite`.

Normalement, les tests unitaires utilisent des données attendues qui sont des constantes.

Utilisation de méthodes privées dans les tests

Vous remarquerez que j'ai utilisé des méthodes privés pour éviter de répéter du code :

FalseOrbiteTest.java

```
    @Test public void testDistanceBifocaleEXOSAT() {
        this.testDistanceFalseOrbite(this.fexosat);
    }

    private void testDistanceFalseOrbite(FalseOrbite o) {
        double c = o.getC();

        double distance = 2 * (o.getA() - c) + 2 * c;

        Point foyer1 = o.getFoyer();
        Point foyer2 = new Point(foyer1.getX() - 2 * c, foyer1.getY());

        Point p = null;

        for (double theta = 0; theta < 2 * Math.PI; theta += STEP) {
            p = o.calculerPointSurOrbite(theta);
            assertEquals(distance, p.distance(foyer1) + p.distance(foyer2), EPS);
        }
    }
```

Correction de l'erreur sur b

Il fallait « corriger » l'erreur sur b pour pouvoir effectuer d'autres tests :

FalseOrbiteTest.java

```java
    /**
     * Test method for getB for EXOSAT with correction.
     */
    @Test public void testGetBEXOSATCorrect() {
        this.fexosat.setB(this.exosat.getB());
        assertEquals(this.exosat.getB(),
                     this.fexosat.getB(), EPS);
    }

    /**
     * Testing if the sum of the distances to the focus is constant
     * for EXOSAT with correction.
     */
    @Test public void testDistanceBifocaleEXOSATCorrect() {
        this.fexosat.setB(this.exosat.getB());
        this.testDistanceFalseOrbite(this.fexosat);
    }
```

Plan

1. TP sur les classes et les objets
2. **TP sur les tests unitaires et les associations**
   - Présentation
   - Tester `FalseOrbite`
   - Implantation de `OrbiteDiscrete`
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Quelques remarques

L'implantation et le test n'étaient pas difficiles, il fallait réfléchir un peu :

- la visibilité des rôles imposait l'utilisation de `clone`
- attention à l'utilisation de `clone` sur `ArrayList` (*shallow copy*)
- beaucoup de méthodes sont des méthodes qui **délèguent** leur travail :

OrbiteDiscrete.java

```java
    public void translater(double dx, double dy) {
        for (Point p: this.points) {
            p.translater(dx, dy);
        }

        this.centre.translater(dx, dy);
    }
```

- attention, lors de l'homothétie, il fallait se ramener à un centre à l'origine

Plan

1 TP sur les classes et les objets

2 TP sur les tests unitaires et les associations

3 **TP sur les interfaces**
- Présentation
- Quelques éléments de correction...

4 TP sur la spécialisation et l'héritage

5 TP sur le patron de conception observateur

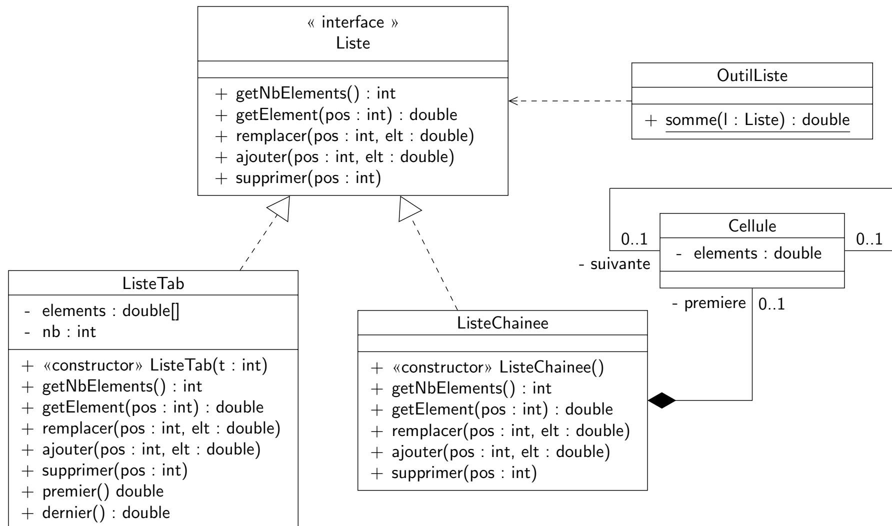
6 TP sur les interfaces graphiques avec Swing

7 TP sur les exceptions

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. **TP sur les interfaces**
   - Présentation
   - Quelques éléments de correction...
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Liste et ses réalisations



Retour sur la méthode somme...

OutilListe.java

```
public class OutilListe {

    public static double somme(Liste l) {
        double somme = 0;
        for (int i = 0; i < l.getNbElements(); i++) {
            somme += l.getElement(i);
        }

        return somme;
    }
}
```

Questions

1. peut-on utiliser somme avec une instance de ListeChainee ?
2. est-ce que somme est efficace sur une instance de ListeChainee ?

Parcours d'une liste

Il faudrait trouver un mécanisme permettant de **parcourir** une liste **quelle que soit son implantation**.

De quelles méthodes a-t-on besoin ?

Parcours d'une liste

Il faudrait trouver un mécanisme permettant de **parcourir** une liste **quelle que soit son implantation**.

De quelles méthodes a-t-on besoin ?

- `avancer()` pour avancer dans la liste
- `estTermine()` pour savoir si on a fini de parcourir la liste
- `elementCourant()` pour récupérer la valeur stockée dans l'élément courant

Parcours d'une liste

Il faudrait trouver un mécanisme permettant de **parcourir** une liste **quelle que soit son implantation**.

De quelles méthodes a-t-on besoin ?

- `avancer()` pour avancer dans la liste
- `estTermine()` pour savoir si on a fini de parcourir la liste
- `elementCourant()` pour récupérer la valeur stockée dans l'élément courant

Où déclarer ces méthodes ?

Parcours d'une liste

Il faudrait trouver un mécanisme permettant de **parcourir** une liste **quelle que soit son implantation**.

De quelles méthodes a-t-on besoin ?

- `avancer()` pour avancer dans la liste
- `estTermine()` pour savoir si on a fini de parcourir la liste
- `elementCourant()` pour récupérer la valeur stockée dans l'élément courant

Où déclarer ces méthodes ?

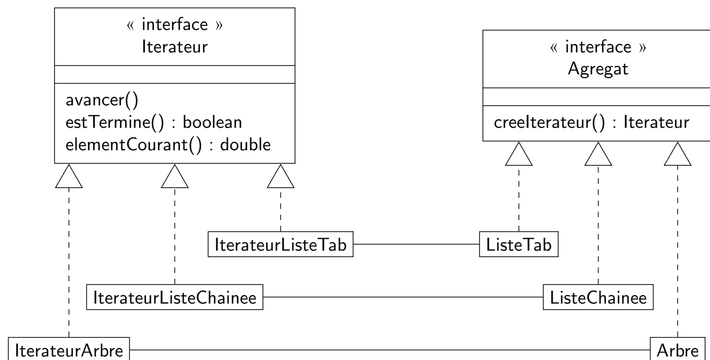
Dans `Liste` ? Pas forcément le bon endroit...

Définition (patron de conception)

Un patron de conception (*design pattern*) est une solution de **conception** (i.e. un ensemble de classes et les relations les liant) considéré comme une **bonne pratique** pour résoudre un problème de conception. La solution décrite par le patron est **générique** et s'adapte à différentes instances du problème.

Définition (itérateur)

Le patron de conception **itérateur** fournit un mécanisme de parcours **itératif** d'un **agrégat** d'objets (i.e. un conteneur d'objets) indépendamment de la structure même de l'agrégat.



Objectifs du TP

Objectifs

1. définir les interfaces `Agregat` et `Iterateur`
2. écrire une classe permettant de compter les éléments d'un agrégat quelconque
3. écrire un itérateur pour `ListeChainee`

Plan

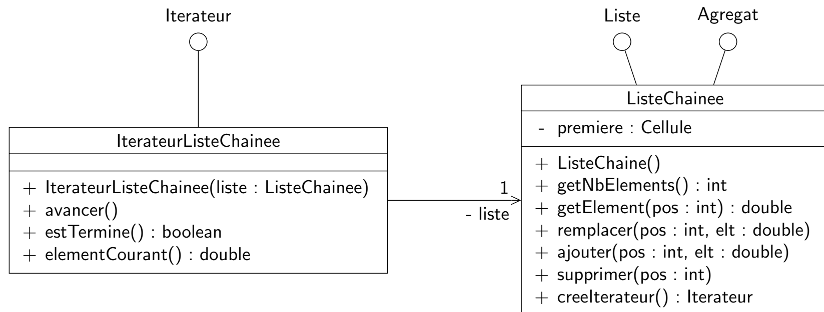
1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. **TP sur les interfaces**
   - Présentation
   - Quelques éléments de correction. . .
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

La classe Utilitaire

Utilitaire.java

```
    public static int nbElements(Agregat ag) {

        Iterateur i = ag.creeIterateur();
        int nb = 0;

        while (!(i.estTermine())) {
            nb++;
            i.avancer();
        }

        return nb;
```

Conception de IterateurListeChainee



Implantation de IterateurListeChainee

IterateurListeChainee.java

```
package fr.isae.lists;

public class IterateurListeChainee implements Iterateur {

    private Cellule cel;

    /**
     * Creer un iterateur sur une liste chainee.
     *
     * @param liste la liste sur laquelle on veut un iterateur
     */
    public IterateurListeChainee(ListeChainee liste) {
        this.cel = liste.getPremiereCellule();
    }

    @Override public void avancer() {
        this.cel = cel.getSuivante();
    }

    @Override public boolean estTermine() {
        return (cel == null);
    }
```

Tester `IterateurListeChainee`

Comment tester `IterateurListeChainee`

Règle « 0-1-n » :

1. tester l'itérateur avec une liste **vide**
2. tester l'itérateur avec une liste contenant **un seul élément**
3. tester l'itérateur avec une liste « quelconque »

Normalement, il faut vérifier à chaque fois que l'on parcourt bien **toute** la liste, **dans l'ordre** et que l'**on ne modifie pas la liste**.

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. **TP sur la spécialisation et l'héritage**
   - Présentation
   - Quelques éléments de correction...
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. **TP sur la spécialisation et l'héritage**
   - Présentation
   - Quelques éléments de correction...
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Exercice 1 : compréhension du polymorphisme

Exercice

En utilisant les classes `Point` et `PointNomme`, comprendre le polymorphisme et la liaison tardive via un programme « exhaustif ».

Par exemple :

```
        // Definir une poignee sur un point nomme
        PointNomme qn;

        // Attacher un point a q et l'afficher
        qn = p1;        // Est-ce autorise ? Pourquoi ?
        System.out.println ("> qn = p1;");
        System.out.print ("qn = ");     qn.afficher();  System.out.println ();
                        // Qu'est ce qui est affiche ?
```

Exercice 2 : redéfinir une méthode

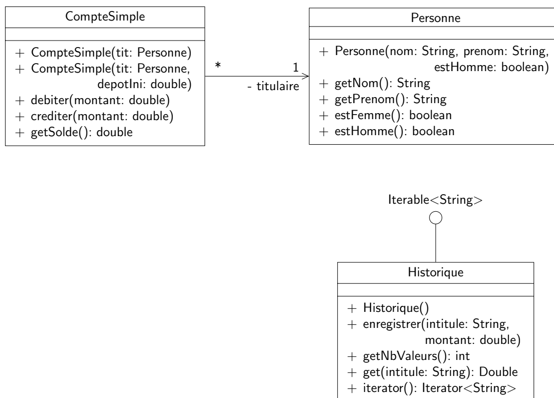
Exercice

Redéfinir la méthode `equals` de `Object` dans `Point` et `PointNomme` et comprendre son fonctionnement.

Exercice 3 : spécialiser une classe

Exercice

Définir à partir d'une classe `CompteSimple` une classe `CompteCourant` permettant de gérer un historique.



Exercice 4 : créer une classe `LDD`

Exercice

Peut-on créer une classe `LDD` en spécialisant la classe `CompteSimpe` ?

Plan

1 TP sur les classes et les objets

2 TP sur les tests unitaires et les associations

3 TP sur les interfaces

4 **TP sur la spécialisation et l'héritage**
- Présentation
- Quelques éléments de correction...

5 TP sur le patron de conception observateur

6 TP sur les interfaces graphiques avec Swing

7 TP sur les exceptions

Exercice 2 : redéfinition de `equals`

La méthode `equals` de `Object` prend un `Object` en paramètre. Il faut donc lorsqu'on la redéfinit :

- vérifier que la référence passée en paramètre peut être transtypée vers `Point` ou `PointNomme` (via **`instanceof`**)
- transtyper la référence pour effectuer les tests d'égalité

Exercice 2 : redéfinition de `equals`

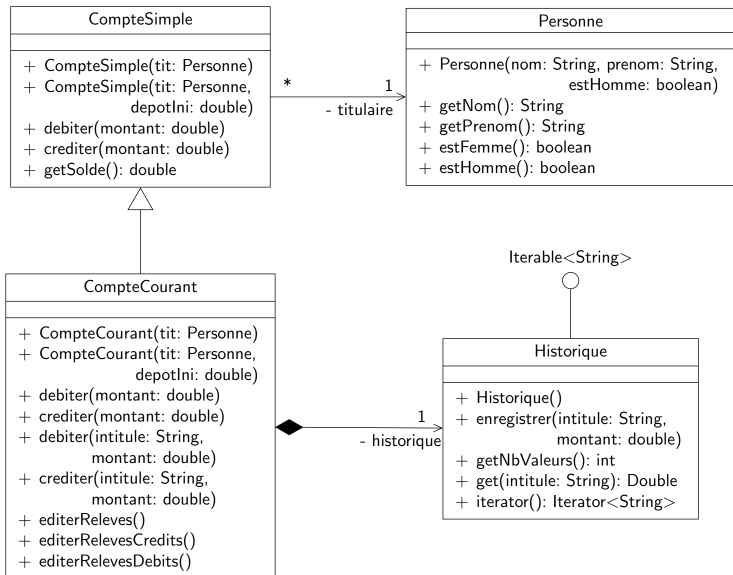
La méthode `equals` de `Object` prend un `Object` en paramètre. Il faut donc lorsqu'on la redéfinit :

- vérifier que la référence passée en paramètre peut être transtypée vers `Point` ou `PointNomme` (via **`instanceof`**)
- transtyper la référence pour effectuer les tests d'égalité

Attention

`p.equals(pn)` n'implique pas `pn.equals(p)` !

Exercice 3 : conception



Exercice 3 : implantation

Méthode **redéfinie** :

CompteCourant.java

```
52      /**
53       * <code>crediter</code> credite le compte du montant fourni (en euros) et
54       * enregistre l'operation. Il n'y a pas d'intitule.
55       *
56       * @param montant un <code>double</code> qui est le montant a crediter
57       */
58      @Override public void crediter(double montant) {
59          this.crediter("", montant);
60      }
```

Exercice 3 : implantation

Méthode **surchargée** :

CompteCourant.java

```
74      * enregistre l'operation. Elle est declaree final car elle est utilisee d
75      *  le constructeur.
76      *
77      * @param intitule une instance de <code>String</code> qui est l'intitule
78      *                 de l'operation
79      * @param montant un <code>double</code> qui est le montant a crediter
80      */
81     public final void crediter(String intitule, double montant) {
82         super.crediter(montant);
83         this.historique.enregistrer(intitule, montant);
84     }
85
86     /**
```

Exercice 4

Ce n'est pas possible de spécialiser `CompteSimple` en `LDD` **à cause du principe de substitution** :

- `crediter` dans `CompteSimple` doit effectivement ajouter le montant passé en paramètre au solde du compte
- `crediter` dans `LDD` peut ne pas ajouter le montant au solde du compte si on dépasse le solde autorisé par la loi

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. **TP sur le patron de conception observateur**
   - Présentation
   - Quelques éléments de correction...
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions

Plan

1 TP sur les classes et les objets

2 TP sur les tests unitaires et les associations

3 TP sur les interfaces

4 TP sur la spécialisation et l'héritage

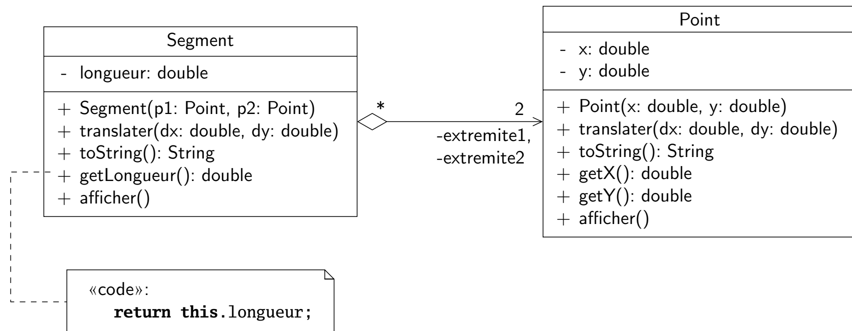
5 **TP sur le patron de conception observateur**
- Présentation
- Quelques éléments de correction. . .

6 TP sur les interfaces graphiques avec Swing

7 TP sur les exceptions

Une classe Segment

On décide de réaliser une classe Segment comme suit :



Préparation de la validation

Exercice

Indiquer ce qui devrait apparaître à l'écran lors de l'exécution du programme de test.

Exercice

Écrire un diagramme de séquence représentant le scénario.

Préparation de la validation : sortie console

shell

```
p2 = (5.0,0.0)

s = [(0.0,0.0);(5.0,0.0)]

longueur de s = 5.0

p2 = (3.0,0.0)

s = [(0.0,0.0);(3.0,0.0)]

longueur de s = 3.0
```

Préparation de la validation : diagramme de séquence

Exercice

En utilisant les sources fournies, compléter le diagramme de séquence précédent.

Exercice

Indiquer ce qui devrait apparaître à l'écran lors de l'exécution du programme de test en utilisant les sources fournies.

Première implantation : diagramme de séquence

: TestSegment
«create» (0,0)
p1: Point
«create» (5,0)
p2: Point
«create» (p1,p2)
s: Segment
distance(p2)
5
afficher()
toString()
afficher()
toString()
"(0,0)"
toString()
"(5,0)"
getLongueur()
5
translater(-2,0)
afficher()
toString()
afficher()
toString()
"(0,0)"
toString()
"(3,0)"
getLongueur()
5

Première implantation : sortie console

shell

```
p2 = (5.0,0.0)

s = [(0.0,0.0);(5.0,0.0)]

longueur de s = 5.0

p2 = (3.0,0.0)

s = [(0.0,0.0);(3.0,0.0)]

longueur de s = 5.0
```

Hypothèse

- la relation entre `Segment` et `Point` reste inchangée
- l'attribut `longueur` et la méthode `getLongueur()` de `Segment` restent inchangés

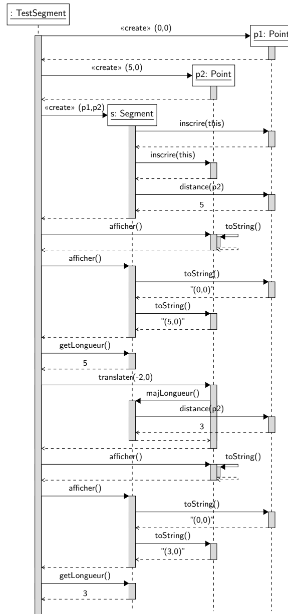
Exercice

Indiquer les modifications à apporter en complétant le diagramme de séquence précédent.

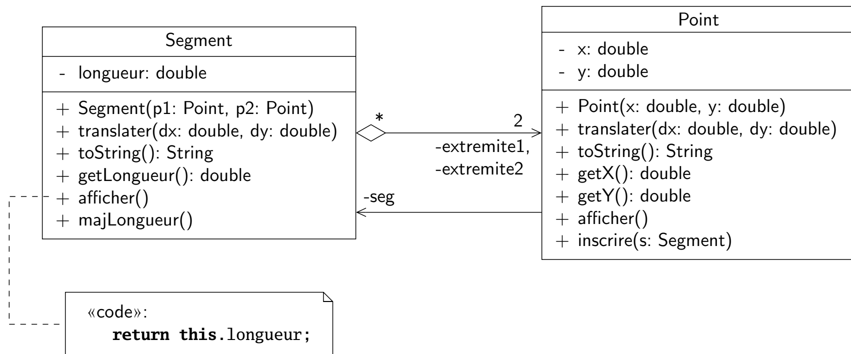
Exercice

Modifier le diagramme de classes initial en conséquence.

Correction des classes : diagramme de séquence



Correction des classes : diagramme de classes

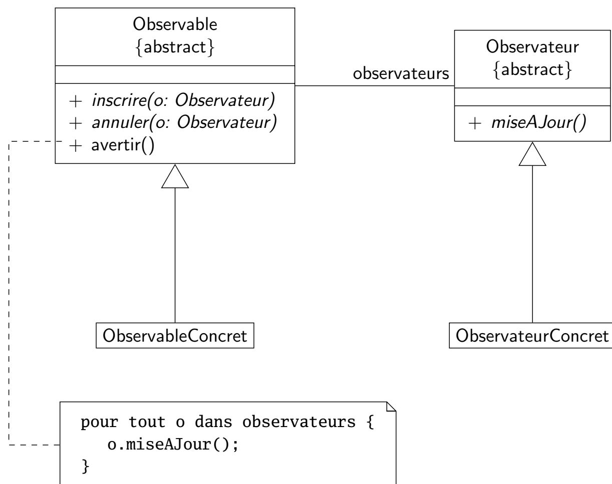


Dernières mises au point

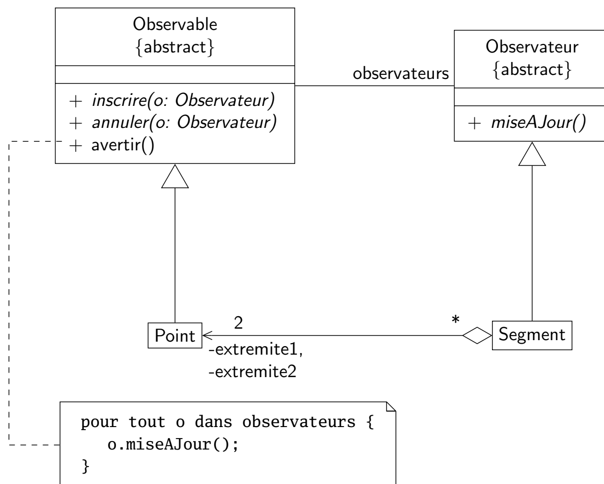
Exercice

- un point peut-il être extrémité de plusieurs segments ?
- la solution peut-elle adaptée à un cas plus général ?

Le patron de conception observateur



Observateur pour notre problème



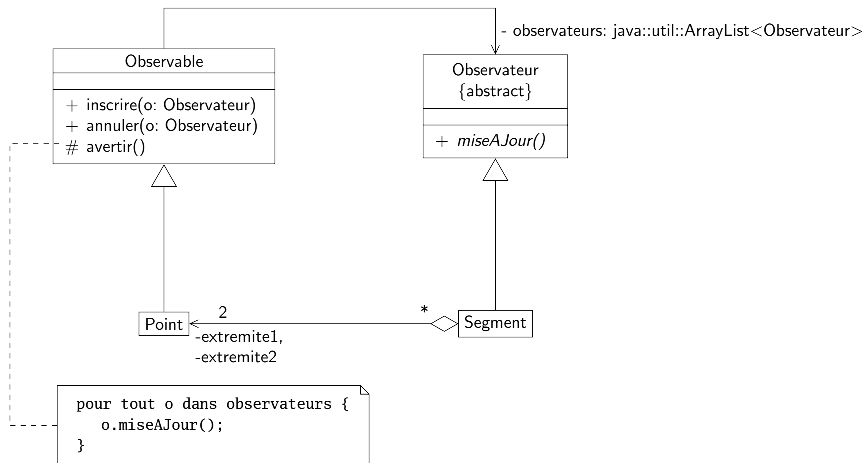
Adapter le patron

Exercice

Adapter le patron :

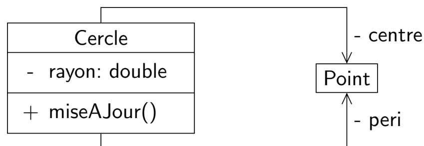
- peut-on utiliser des interfaces ?
- `Observable` est-elle abstraite ?
- visibilité des méthodes de `Observable` ?
- méthode `miseAJour` « efficace » ?

Adapter le patron : classe Observable



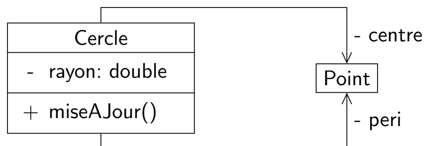
Méthode miseAJour

Supposons que l'on ait une classe Cercle :

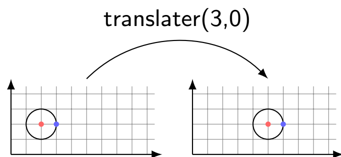


Méthode `miseAJour`

Supposons que l'on ait une classe `Cercle` :

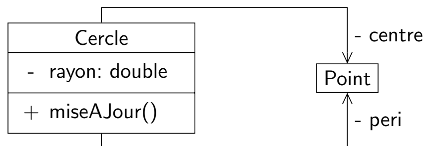


Translation du centre :

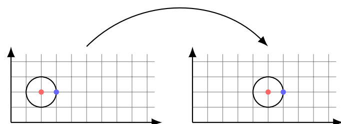


Méthode `miseAJour`

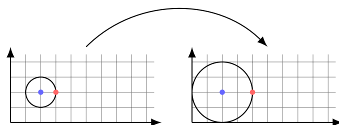
Supposons que l'on ait une classe `Cercle` :



Translation du centre :

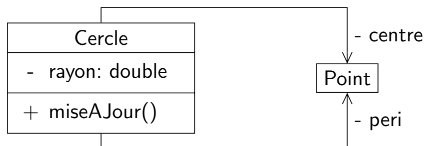


Translation du point périphérique :

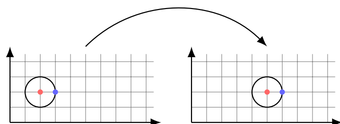


Méthode `miseAJour`

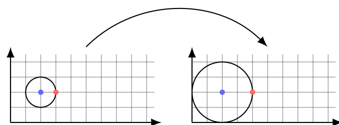
Supposons que l'on ait une classe `Cercle` :



Translation du centre :

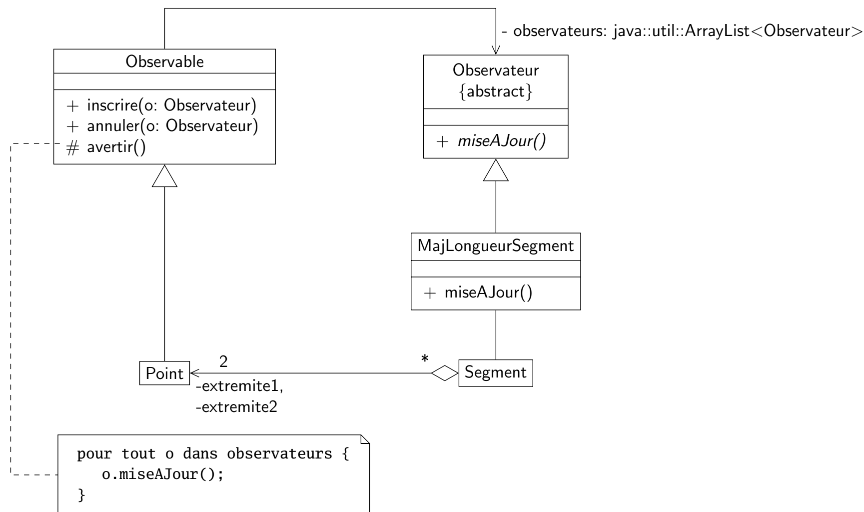


Translation du point périphérique :



Comment coder `miseAJour` ?

Méthode miseAJour



Implantation en TP

Exercice

Implanter la solution « simple » dans un premier temps, puis essayer de coder `MajLongueurSegment`, voire d'utiliser la classe `Observable` et l'interface `Observer` fournies par l'API.

Plan

1 TP sur les classes et les objets

2 TP sur les tests unitaires et les associations

3 TP sur les interfaces

4 TP sur la spécialisation et l'héritage

5 **TP sur le patron de conception observateur**
- Présentation
- Quelques éléments de correction...

6 TP sur les interfaces graphiques avec Swing

7 TP sur les exceptions

Implantation simple

Rien de bien difficile, il fallait :

1. implanter les classes `Observable` et `Obsever`
2. modifier `Segment` et `Point` en conséquence
3. utiliser le diagramme de séquence pour écrire un scénario de test

Les pièges « classiques » :

- oubli de l'inscription du segment dans le constructeur de `Segment`
- oubli de l'appel à `avertir` dans `translater`

java.util.Observable et java.util.Observer

java::util::Observable
+ Observable() + addObserver(o: Observer) + deleteObserver(o: Observer) + countObserver(): int # clearChanged() # setChanged() + hasChanged(): boolean + notifyObservers() + notifyObservers(arg: Object)

« interface » java::util::Observer
update(o: Observable, arg: Object)

Voir le corrigé pour plus de détail sur leur utilisation.

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. **TP sur les interfaces graphiques avec Swing**
   - Présentation
   - Quelques éléments de correction. . .
7. TP sur les exceptions

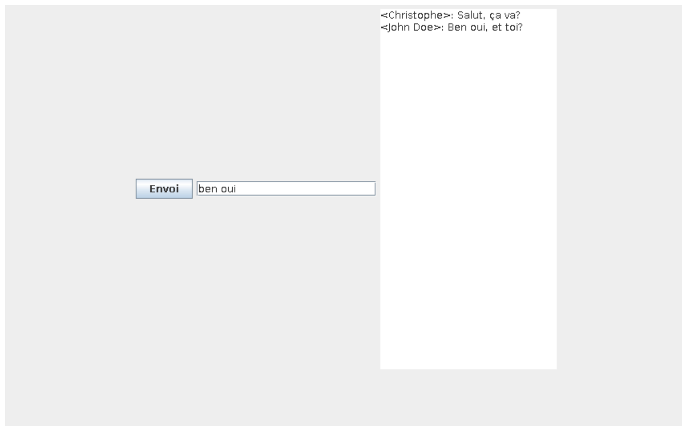
Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. **TP sur les interfaces graphiques avec Swing**
   - Présentation
   - Quelques éléments de correction. . .
7. TP sur les exceptions

Un « chat » en Swing

Exercice

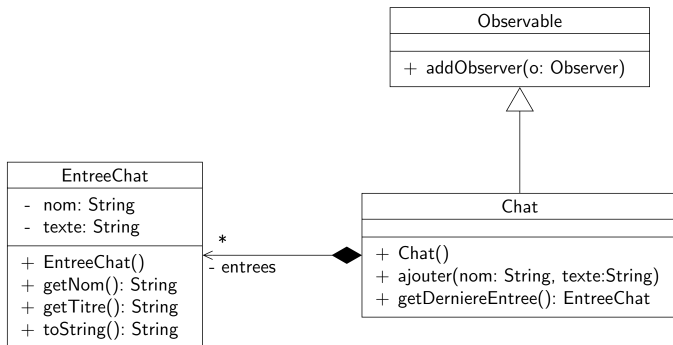
Créer un chat avec l'API Swing avec plusieurs fenêtres devant être mises à jour lors de l'envoi d'un message.



Chat : MVC

Modèle (fourni)

- stocker les entrées (nom utilisateur + texte) dans une liste
- récupérer la dernière entrée de la liste



Chat : MVC

Vue (à faire)

- un `JButton`, un `JTextField` et une `JTextArea`
- layout à choisir

Contrôleur (à faire)

- c'est le `JButton` qui demande au modèle de se mettre à jour
- le MVC est **passif**, on utilisera le patron Observateur

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. **TP sur les interfaces graphiques avec Swing**
   - Présentation
   - Quelques éléments de correction...
7. TP sur les exceptions

Correction

Rien de bien difficile, il ne fallait pas oublier :

- de **créer** les objets, en particulier les composants graphiques
- d'utiliser le patron Observateur pour mettre à jour les vues
- de coder les méthodes relatives au patron de conception Observateur
- d'inscrire les vues comme observateurs du *chat* lors de leur création

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. **TP sur les exceptions**
   - Présentation
   - Quelques éléments de correction...

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. **TP sur les exceptions**
   - Présentation
   - Quelques éléments de correction. . .

Exercice : acquérir des données

Problème

On souhaite récupérer des données de pression provenant d'une expérience. Ces données sont donc des **réels qui doivent normalement être positifs**. On va donc créer une classe qui devra offrir les services suivants :

- lecture du fichier de données et vérification de la cohérence des données ;
- stockage des données ;
- renvoi d'un itérateur sur l'ensemble de données.

Lors d'erreurs d'écriture des données, il faudra que l'utilisateur puisse récupérer toutes les valeurs posant problème.

Le fichier de données

Le fichier de données sera au format XML :

data1.xml

```
<experience resp-name="Allan Bonnet">
  <pression>25.754</pression>
  <pression>10.432</pression>
  <pression>30.6754</pression>
  <pression>42.543</pression>
  <pression>100.678</pression>
</experience>
```

- `experience` et `pression` sont des éléments XML
- `resp-name` est un attribut
- **tout est du texte !**

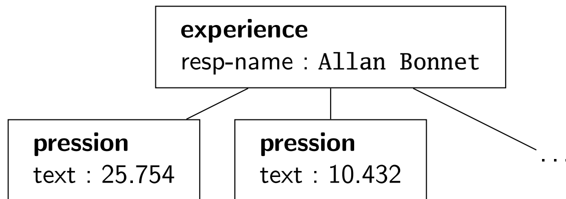
Le fichier de données

Le fichier de données sera au format XML :

data1.xml

```xml
<experience resp-name="Allan Bonnet">
  <pression>25.754</pression>
  <pression>10.432</pression>
  <pression>30.6754</pression>
  <pression>42.543</pression>
  <pression>100.678</pression>
</experience>
```

Représentation sous forme d'arbre :

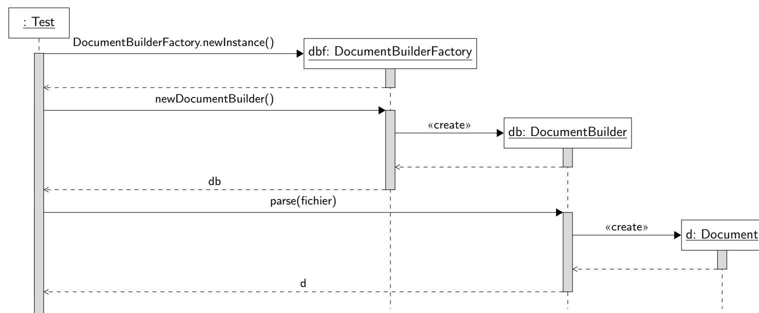


Acquérir des données

Questions

1. conception de la classe `Acquisition`
2. quelles sont les exceptions qui peuvent être potentiellement levées ?
3. comment gérer ces exceptions ?
4. implantation et test de `Acquisition`

L'API JAXP : créer le document XML



L'API JAXP : travailler avec l'arbre

Interface `Node`

- **public short** `getNodeType()`
- **public** `String getTextContent`
- **public** `NodeList getChildNodes`

Interface `Element`

- **public** `String getAttribute(String name)`

Les entrées/sorties en Java

Deux types de flux en entrée/sortie représentés par 4 classes **abstraites** :

	Caractères	Bytes
Entrée	Reader	InputStream
Sortie	Writer	OutputStream

Transformer des flux d'octets en flux de caractères : `InputStreamReader` et `OutputStreamWriter`.

Quelques classes concrètes (`XXX` peut être `Reader`, `Writer`, `InputStream` ou `OutputStream`) :

- `BufferedXXX`
- `FileXXX`
- ...

Toutes ces classes appartiennent au paquetages `java.io`.

`java.util.Scanner` permet d'« analyser » un flux (e.g. séparer une chaîne de caractères en mots).

I/O en Java : un exemple (cf. sujet)

EntierClavier.java

```
import java.io.*;

public class EntierClavier {
    public static void lireEntier() {
        try {
            System.out.print("Entrez un entier : ");

            InputStreamReader aux = new InputStreamReader(System.in);
            BufferedReader in = new BufferedReader(aux);

            String s = in.readLine().trim();

            int n = Integer.parseInt(s);

            System.out.println("Le nombre est : " + n);
        } catch (NumberFormatException  e) {
            System.out.println("Ce n'est pas un entier !"); }
        catch (IOException e) {
            System.out.println("Erreur d'entree/sortie !"); }
    }
}
```

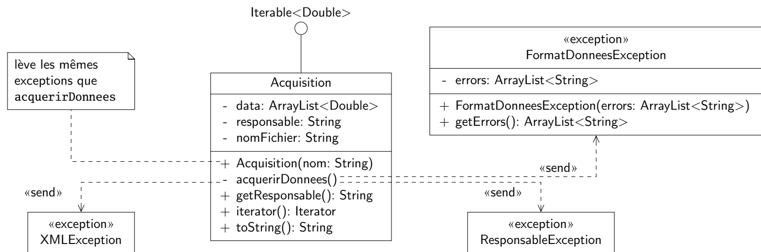
Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. **TP sur les exceptions**
   - Présentation
   - Quelques éléments de correction...

Gestion des exceptions

Toutes les exceptions sont propagées, on ne sait pas les traiter localement. Attention, ce sont des choix personnels, rien n'empêche de les traiter localement, mais il me semble que c'est moins cohérent.

Conception



Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions
8. **TP sur la généricité**
   - Présentation

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions
8. **TP sur la généricité**
   - Présentation

Exercice : un ensemble bizarre. . .

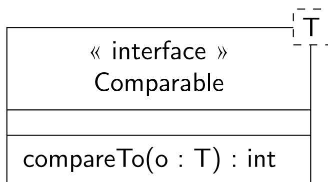
Problème

On souhaite créer un ensemble d'éléments qui sont comparables entre eux. Les ensembles doivent également être comparables entre eux.

Opérations demandées sur les ensembles de ce type :

- ajouter un élément ;
- enlever un élement ;
- obtenir un itérateur ;
- récupérer le minimum ;
- construire l'union de deux ensembles de types compatibles.

L'interface `Comparable`



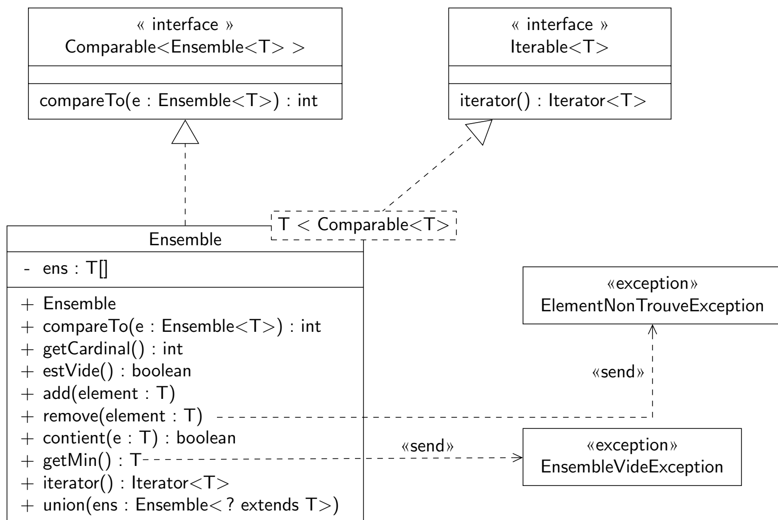
Soit une classe `C`. La relation définissant l'**ordre naturel** sur `C` est définie par :

$$\{(x, y) \mid \texttt{x.compareTo(y)} \leq 0 \text{ et } \texttt{x} \text{ et } \texttt{y} \text{ instances de } \texttt{C}\}$$

Plan

1. TP sur les classes et les objets
2. TP sur les tests unitaires et les associations
3. TP sur les interfaces
4. TP sur la spécialisation et l'héritage
5. TP sur le patron de conception observateur
6. TP sur les interfaces graphiques avec Swing
7. TP sur les exceptions
8. **TP sur la généricité**
   - Présentation

Diagramme UML



La méthode getMin

Ensemble.java

```
    public T getMin() throws EnsembleVideException {
        if (this.getCardinal() == 0) {
            throw new EnsembleVideException("L'ensemble est vide !");
        }

        T min = this.ens.get(0);

        for (T o : this.ens) {
            if (min.compareTo(o) > 0) {
                min = o;
            }
        }

        return min;
    }
```

La méthode union

Ensemble.java

```java
    public Ensemble<T> union(Ensemble<? extends T> e) {
        Ensemble<T> res = new Ensemble<T>();

        for (T o : this.ens) {
            res.add(o);
        }

        for (T o : e.ens) {
            res.add(o);
        }

        return res;
    }
```

La méthode somme

TestEnsemble.java

```
    public static double somme(Ensemble<? extends Number> ens) {
        double res = 0;

        Iterator<?> i = ens.iterator();
        while (i.hasNext()) {
            res += ((Number) i.next()).doubleValue();
        }

        return res;
    }
```

La méthode somme

TestEnsemble.java

```
    public static double somme(Ensemble<? extends Number> ens) {
        double res = 0;

        Iterator<?> i = ens.iterator();
        while (i.hasNext()) {
            res += ((Number) i.next()).doubleValue();
        }

        return res;
    }
```

Remarquez l'utilisation du *wildcard* pour l'itérateur sur l'ensemble. On aurait également pu utiliser une boucle **for** car l'ensemble est itérable :

```
    for (Number n : ens) {
        res += n.doubleValue();
    }
```