Conception Orientée Objet

Licence Informatique
semestre 5
parcours Info et Miage

Automne 2015

Responsable :
Damien Cassou
<damien.cassou@univ-lille1.fr>

Équipe enseignante :
Clément Bera
Damien Cassou
Bilel Derbel
Lucien Mousin
Xavier le Pallec
Hanaë Rateau
Jean-Christophe Routier

version: 2015-09-01 11:21

Règles d'organisation

Ce document présente les règles d'organisation du module de COO. Ce module est organisé de façon à ce que seuls les étudiants qui iront chercher les connaissances apprendront et réussiront le module : n'attendez pas que les enseignants vous déversent la connaissance sans que vous n'ayez fait d'effort préalable ! L'apprentissage est un acte individuel et l'enseignant ne peut pas apprendre à la place des étudiants. La réussite à ce module dépend donc fortement de votre capacité à travailler en autonomie.

1 Les cours en amphithéâtre

Enseignant Je m'engage à être à l'heure à chaque cours. En cas de retard, je m'engage à transmettre aux étudiants un courrier électronique détaillant les points qui n'auraient pas pu être discutés pendant le cours.

Étudiant Vous vous engagez à être à l'heure. En cas de retard, vous vous engagez à venir me demander l'autorisation d'assister au cours avant de vous asseoir ou vous attendez la pause au milieu de la séance. Dans tous les cas, vous devrez connaître le contenu du cours.

Enseignant Je vérifierai votre compréhension en vous posant des questions.

Étudiant Vous vous engagez à répondre aux questions que je vous poserai.

Enseignant Je m'engage à répondre aux questions que vous me poserez à tout moment de la séance.

Étudiant Vous me poserez des questions dès que vous ne comprendrez pas.

Enseignant Je m'engage à considérer au fur et à mesure du semestre toutes les remarques et critiques que vous me ferez sur l'organisation et le contenu du module.

Étudiant Vous me ferez des remarques et critiques sur l'organisation et le contenu du module au fur et à mesure du semestre.

2 Les devoirs surveillés

Le polycopié de COO (avec notes personnelles) est autorisé ainsi qu'un dictionnaire de langue (papier ou électronique *dédié*). Tout autre document est interdit.

3 Les travaux dirigés

Les règles qui suivent sont des règles générales qui peuvent être adaptées par chaque enseignant, à sa discrétion.

Enseignant Je m'engage à être à l'heure à chaque séance.

Étudiant Vous vous engagez à être à l'heure. En cas de retard, vous vous engagez à venir me demander l'autorisation d'assister à la séance avant de vous asseoir ou vous attendez la pause au milieu de la séance (si pause il y a). Dans tous les cas, vous devrez connaître le contenu de la séance.

Enseignant Je vérifierai votre compréhension en vous posant des questions.

Étudiant Vous vous engagez à répondre aux questions que je vous poserai.

Enseignant Je m'engage à répondre aux questions que vous me poserez.

Étudiant Vous me poserez des questions dès lors que vous ne comprendrez pas.

Enseignant Je m'engage à préparer les sujets.

Étudiant Vous vous engagez à préparer chaque séance à l'avance en faisant les questions préparatoires. Si vous n'y arrivez pas, vous devez préparer des questions à poser à l'enseignant et passer à la question préparatoire suivante.

Enseignant Je m'engage à ne pas corriger les questions préparatoires. Je m'engage à répondre à toutes les questions des étudiants qui ne seraient pas arrivés à faire une question préparatoire.

4 Les séances de projet

Les règles qui suivent sont des règles générales qui peuvent être adaptées par chaque enseignant, à sa discrétion.

Enseignant Je m'engage à être présent et à l'heure à toutes les séances de projet.

Étudiant Vous vous engagez à être présent et à l'heure à toutes les séances de soutenance (voir semainier).

Enseignant Je m'engage à répondre aux questions que vous me poserez.

Étudiant Vous me poserez des questions dès lors que vous ne comprendrez pas.

Enseignant Je m'engage à fournir des remarques constructives sur votre projet au plus tard 15 jours après la date de rendu.

Étudiant Vous vous engagez à rendre votre projet au plus tard à la date indiquée (voir le semainier).

Enseignant Je m'engage à sanctionner tous les retards.

Résumé des TDs et de leurs apprentissages

1 – Donjon À la fin de ce TD, l'étudiant devrait être capable
- d'implémenter une spécification d'algorithme simple au sein d'une classe ;
- de transformer une spécification impérative (comme des règles de jeu) en une conception objet ;
- de lire et de restructurer du code existant ;
- de séparer le code facilement testable du code difficilement testable (par exemple le code gérant l'aléatoire, les entrées utilisateurs, le réseau, etc.) et de minimiser le code difficilement testable.

2 – Tour de France À la fin de ce TD, l'étudiant devrait être capable
- de comprendre le lien entre une spécification écrite en langue naturelle et une spécification exécutable sous forme de tests unitaires ;
- de traduire une spécification simple écrite en langue naturelle vers une spécification sous forme de tests unitaires ;
- de comprendre et d'utiliser les *tables de hachages* (aussi appelées *dictionnaires*) ;
- de comprendre et d'implémenter des fonctions de comparaisons d'objets (`compareTo()`, `equals()`, `hashCode()`).

3 – Transport À la fin de ce TD, l'étudiant devrait être capable
- de concevoir une application objet avec de l'héritage ;
- de faire de l'héritage de classes abstraites pour réunir plusieurs propriétés et algorithmes communs au sein de mêmes classes ;
- d'utiliser le patron de conception *template method*.

4 – Actions À la fin de ce TD, l'étudiant devrait être capable
- de modéliser le temps avec des objets ;
- de tester une abstraction ;
- de différencier les différents types d'exceptions ;
- de créer de nouvelles classes d'exception ;
- de créer de fausses entités pour isoler une entité à tester ;
- de lire et de restructurer du code existant ;
- d'utiliser le patron de conception *composite* ;
- d'utiliser le patron de conception *factory method*.

5 – Problème de la piscine À la fin de ce TD, l'étudiant devrait être capable
- d'utiliser le patron de conception *factory method* ;
- de comprendre l'intérêt des classes génériques et de pouvoir les utiliser dans un contexte simple ;
- de modéliser un problème système (la gestion des ressources) avec des objets ;
- de combiner les fonctionnalités de plusieurs classes grâce à la délégation.

6 – Casse-briques À la fin de ce TD, l'étudiant devrait être capable
- de faire des arbres d'héritage parallèles ;
- de comprendre l'intérêt du *double dispatch* ;
- d'utiliser le patron de conception *decorator* ;
- d'utiliser le patron de conception *abstract factory*.

7 – Courriers À la fin de ce TD, l'étudiant devrait être capable
- d'utiliser le patron de conception *decorator* ;
- de concevoir des applications simples évoluant par extension et non par modification (principe ouvert/fermé, *OCP = Open/Closed Principle*) ;
- de concevoir des classes génériques dont les paramètres sont eux-mêmes génériques.

8 – Louables et contraintes À la fin de ce TD, l'étudiant devrait être capable
- d'utiliser le patron de conception *strategy* ;
- d'utiliser le patron de conception *composite* ;
- d'utiliser la réflexion pour garder à l'exécution des informations sur les paramètres de types génériques.

9 – Plugins À la fin de ce TD, l'étudiant devrait être capable
- d'utiliser le patron de conception *observer* ;
- de concevoir et d'implémenter des programmes simples réagissant à des évènements (ici, fonctions du temps et du système de fichiers) : on parle de programmation *réactive* ;
- de concevoir et d'implémenter des programmes dont le comportement est étendu à l'exécution.

10 – Questionnaire À la fin de ce TD, l'étudiant devrait être capable
- d'utiliser les mécanismes de base de l'API de réflexion Java ;
- d'écrire des tests pour du code d'interface utilisateur.

11 – Expressions À la fin de ce TD, l'étudiant devrait être capable
- d'appréhender la conception et l'implémentation de langages de programmation ;
- de simuler l'utilisation de fonctions de première classe avec des interfaces ;
- d'utiliser le patron de conception *visitor*.

Projets de COO

Les projets de COO se déroulent sur tout le semestre. Ces projets sont organisés pour que vous soyez complètement autonome. En dehors des dates de remise prévues et des soutenances, vous êtes libres de travailler quand vous le souhaitez. Les règles d'organisation générales du projet sont spécifiées ci-dessous et peuvent être adaptées par chaque enseignant, à sa discrétion.

Séances de projet Dans l'emploi du temps sont réservés, toutes les semaines, des créneaux d'1h30 pour travailler sur le projet. L'enseignant sera présent pour répondre aux questions (sur le projet ou autre). Vous devez être impérativement présents et à l'heure aux soutenances prévues dans ces mêmes créneaux.

Début Les projets commencent la semaine de la rentrée. Une séance de TD sera consacrée à l'introduction de chaque projet. N'attendez pas les séances de projet pour travailler : les projets ne sont pas réalisables en travaillant seulement 1h30 par semaine !

Binômes Les étudiants doivent être obligatoirement en équipes de 2 (une équipe de 3 par groupe est autorisée pour les groupes impairs, à vous de vous organiser pour qu'il n'y en ait qu'une). Les équipes de tout le semestre doivent être spécifiées à l'enseignant lors de la première séance de projet. Les équipes doivent être différentes pour chaque projet.

Remise de projet Les projets seront remis à l'enseignant sous forme d'une archive par équipe contenant les sources, la documentation, les tests, etc.

Dates de remise Les dates de remise des projets sont spécifiées dans le semainier sur le portail et doivent être absolument respectées. Vous devez remettre chaque livrable à 8h le matin du jour où vous avez la séance de projet.

Soutenances Les séances de projet les semaines de remise de projet seront consacrées aux soutenances. La présence de tous les membres de l'équipe est obligatoire. Lors de la soutenance, chaque équipe devra présenter son travail en tête à tête avec l'enseignant et répondre à ses questions. Les équipes qui n'auront pas eu le temps de soutenir pendant la séance devront prendre rendez-vous avec l'enseignant pour soutenir à un autre moment.

Langue du code Le code du projet (noms des classes, variables, méthodes) devra être entièrement écrit en anglais. Les enseignants pourront mettre un bonus si la documentation (dont les commentaires de code) est aussi écrite en anglais.

Critères de notation Chaque projet fera l'objet d'une note représentant une partie importante de la note finale du module (voir détail sur le portail). Cette note prendra en compte, entre autres, la capacité des étudiants à remettre les projets dans les temps, à présenter leur projet, à répondre aux questions et à écrire du *beau code objet* (la définition de cette expression étant le but du module avec tous ses cours et TDs). Réaliser toutes les fonctionnalités d'un projet est important mais moins que la qualité de son code : il est en effet plus facile d'ajouter une fonctionnalité à du code de qualité que de rendre beau le code déplorable d'un programme complet.

Couverture Le code devra être couvert à 60% au minimum par les tests unitaires. Tous les points difficile de chaque projet devront être couverts à 100%. Cette couverture devra être mesurée avec un outil dédié.

Plateforme de référence La plateforme d'implémentation de référence est Oracle Java 8. Celle-ci est disponible par défaut dans le parc informatique du M5 et du SUP.

Le but de chaque projet est d'implémenter complètement le sujet de TD correspondant. Les projets à réaliser sont, dans l'ordre :

1. le jeu de donjons ;
2. les actions et la piscine ;
3. les courriers ;
4. les plugins.

TD 1 — Donjon

Au terme de ce TD, vous devriez être capable
- d'implémenter une spécification d'algorithme simple au sein d'une classe ;
- de transformer une spécification impérative (comme des règles de jeu) en une conception objet ;
- de lire et de restructurer du code existant ;
- de séparer le code facilement testable du code difficilement testable (par exemple le code gérant l'aléatoire, les entrées utilisateurs, le réseau, etc.) et de minimiser le code difficilement testable.

À préparer avant le TD

Ce TD ne contient pas de travail préparatoire car il arrive la première semaine. La plupart des autres TD devront être préparés.

À faire pendant le TD

On se propose d'implémenter un jeu d'exploration de donjons (*dungeon*). Soit le code des classes `Dungeon` et `DungeonTest` ci-dessous.

```
import java.util.Scanner;

public class Dungeon {
  protected String currentRoom = "entrance";
  protected boolean gameIsFinished = false;
  protected final Scanner scanner = new Scanner(System.in);

  public String getCurrentRoom() {
    return currentRoom;
  }

  public void interpretCommand(String command) {
    switch (command) {
    case "go north":
      switch (currentRoom) {
      case "entrance":
        currentRoom = "intersection";
        break;
      case "intersection":
        currentRoom = "exit";
        break;
      }
      break;
    case "go west":
      switch (currentRoom) {
      case "intersection":
        currentRoom = "trap";
        break;
      case "entrance":
        System.out.println("Can't go west!");
        break;
      }
      break;
    default:
      System.out.println("I don't know what you mean");
    }

  }

  public static void main(String[] args) {
    Dungeon dungeon = new Dungeon();
    dungeon.start();
  }

  public void start() {
    do {
      System.out.println("You are in " + getCurrentRoom());
      System.out.println("What do you want to do?");
      System.out.print("> ");

      // Read a command from the user (stdin)
      String line = scanner.nextLine();
      interpretCommand(line);
    } while (!gameIsFinished());
```

```
    System.out.println("You are in " + getCurrentRoom());
    if (gameIsWon()) {
      System.out.println("You win!");
    } else {
      System.out.println("You loose!");
    }
  }

  public boolean gameIsFinished() {
    return gameIsLost() || gameIsWon();
  }

  public boolean gameIsLost() {
    return currentRoom.equals("trap");
  }

  public boolean gameIsWon() {
    return currentRoom.equals("exit");
  }
}
```

```
import org.junit.Before;
import org.junit.Test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

public class DungeonTest {

  protected Dungeon dungeon;

  @Before
  public void createDungeon() {
    dungeon = new Dungeon();
  }

  @Test
  public void initialRoomIsEntrance() {
    assertEquals("entrance", dungeon.getCurrentRoom());
  }

  @Test
  public void gameNotFinishedAtBeginning() {
    assertFalse(dungeon.gameIsFinished());
  }

  @Test
  public void gameWonWhenGoingNorth() {
    dungeon.interpretCommand("go north");
    assertEquals("intersection", dungeon.getCurrentRoom());
    assertFalse(dungeon.gameIsWon());
    dungeon.interpretCommand("go north");
    assertEquals("exit", dungeon.getCurrentRoom());
    assertTrue(dungeon.gameIsWon());
  }

  @Test
  public void gameLostWhenGoingToTrap() {
    dungeon.interpretCommand("go north");
    assertFalse(dungeon.gameIsLost());
    dungeon.interpretCommand("go west");
    assertEquals("trap", dungeon.getCurrentRoom());
    assertTrue(dungeon.gameIsLost());
  }

  @Test
  public void nothingHappensWhenGoingInNonExistingDirection() {
    dungeon.interpretCommand("go west");
    assertEquals("entrance", dungeon.getCurrentRoom());
    assertFalse(dungeon.gameIsFinished());
  }
}
```

Q 1 . Lire et comprendre le code ci-dessus. Dessiner le donjon correspondant à la méthode `interpretCommand()`.

Q 2 . Expliquer les mots clés `protected` et `final` et leur intérêt.

Q 3 . Expliquer l'intérêt des tests et leurs structures.

Examiner le code suivant dans lequel on a changé un peu le code ci-dessus (ce qui est similaire est remplacé par `[...]`) :

```
public void interpretCommand() {

    // Read a command from the user (stdin)
    String command = scanner.nextLine();

    switch (command) {
      [...]
    }

  }

  public void start() {
    do {
      [...]
      interpretCommand();
    } while ([...]);

    [...]
  }
```

Q 4 . Quel est l'impact de ce changement sur l'exécution du programme ? Quel est l'impact sur les tests unitaires ? Quelles bonnes pratiques pouvez vous en déduire ?

Tout en gardant pour le moment les seules directions Ouest (*west*) et Nord (*north*), on souhaite pouvoir spécifier plusieurs donjons avec des configurations de salles différentes sans à chaque fois devoir changer la méthode `interpretCommand()`.

Q 5 . Proposer une architecture de classes facilitant ce changement.

Q 6 . Donner le code qui change.

On souhaite maintenant que chaque salle puisse définir comment accéder aux salles voisines. Par exemple, une salle pourrait définir que la salle suivante se trouve derrière le tableau (*behind the painting*) ou sous le tapis (*below the carpet*). La méthode `interpretCommand()` doit juste s'assurer que le verbe de la commande est bien "go" mais doit déléguer l'interprétation de la direction à la salle en cours.

Q 7 . Proposer une nouvelle architecture et donner le code qui change.

Le jeu peut être rendu plus intéressant en gérant :

- différents types de salle :
  - la salle normale que le joueur ne fait que traverser ;
  - la salle avec monstre qu'il faut terrasser avant de pouvoir faire quoique ce soit ;
  - la salle avec un trésor qu'il faut récupérer ;
  - la salle avec un bouton qu'il faut presser.
- différents types de sorties de salle :
  - la sortie normale que le joueur voit et peut emprunter quand il le souhaite ;
  - la sortie cachée qui ne devient visible (lorsque l'utilisateur demande une description de la salle en cours) qu'après une action (par exemple appui sur un bouton) ;
  - la sortie verrouillée qui est tout le temps visible mais qu'il faut déverrouiller (par exemple avec une clé ou après avoir tué un monstre).
- différents objets transportés par le joueur :
  - des clés ;
  - des armes ;
  - des potions.
- différentes commandes à taper :
  - la commande de description de la salle en cours ;
  - la commande pour taper sur les monstres ;
  - la commande pour utiliser un objet ;
  - la commande pour appuyer sur un bouton.
- différents donjons : il doit exister plusieurs donjons et le jeu amène le joueur au donjon $n + 1$ dès que celui-ci termine le donjon n.

Q 8 . Implémenter les points ci-dessus en laissant libre cours à votre imagination.

TD 2 — Tour de France

Au terme de ce TD, vous devriez être capable
- de comprendre le lien entre une spécification écrite en langue naturelle et une spécification exécutable sous forme de tests unitaires ;
- de traduire une spécification simple écrite en langue naturelle vers une spécification sous forme de tests unitaires ;
- de comprendre et d'utiliser les *tables de hachages* (aussi appelées *dictionnaires*) ;
- de comprendre et d'implémenter des fonctions de comparaisons d'objets (`compareTo()`, `equals()`, `hashCode()`).

Le fil directeur de ce sujet est la modélisation d'un programme qui permet de gérer le classement du Tour de France. Le Tour de France est une course à étapes (*stages*). Dans une telle course, le résultat d'un coureur (*rider*) est déterminé par le cumul des temps qu'il a réalisés à chacune des étapes et le cumul des points qu'il a obtenus lors de ces étapes dans différents classements. Ces classements reposent sur différents critères : meilleur temps (maillot jaune (*yellow jersey*), plus petit temps cumulé), meilleur sprinter (maillot vert (*green jersey*), plus grand nombre de points verts), meilleur grimpeur (maillot à pois rouges (*polka dot jersey*), plus grand nombre de points montagne), meilleur jeune (*young rider*, plus petit temps cumulé des -25 ans), etc.

À préparer avant le TD

Temps Le temps réalisé par un coureur est défini par la donnée de 3 entiers représentant respectivement un nombre d'heures, de minutes et de secondes. Le nombre d'heures n'est pas borné, alors que les nombres de minutes et de secondes sont strictement inférieurs à 60. Lors de la création d'un temps, il doit être possible de passer des valeurs en dehors de ces bornes. Ainsi un objet créé à partir des valeurs (h,m,s)=(0,63,0) aura pour données (1,3,0).
Il doit être possible de comparer 2 temps entre eux et d'ajouter un temps à un autre.
Avant de procéder à l'implémentation de cette classe, il est important de traduire la spécification en français ci-dessus en une spécification exécutable. Voici un début de spécification exécutable (sous la forme d'un test unitaire) qui vérifie que `61 minutes` sont bien converties en `1 heure` et `1 minute` :

```
@Test
public void testUsingValuesOverMaximum() {
  Time time = new Time(0, 61, 0);
  assertEquals(1, time.getHours());
  assertEquals(1, time.getMinutes());
  assertEquals(0, time.getSeconds());
}
```

Q 1 . En suivant le même modèle, définissez un test qui vérifie la contrainte *"Il doit être possible [...] d'ajouter un temps à un autre"*.

Q 2 . Donnez la définition d'une classe `Time` respectant ces contraintes. Cette classe implémentera l'interface `java.lang.Comparable` et disposera de 2 constructeurs : l'un sans paramètre met le temps à 0 et le second prend pour paramètre les valeurs initiales des nombres d'heures, de minutes et de secondes.

À faire pendant le TD

Coureur Les coureurs (*rider*) participant à une course sont définis par la donnée, à la création, de
- ▷ leur nom, une chaîne de caractères,
- ▷ leur âge, un nombre,
- ▷ leur numéro de licence (*license*) (supposé unique pour chaque coureur, mais aucune vérification sur ce point n'est demandée), une chaîne de caractères.

On souhaite pouvoir lire ces informations.

Q 3 . Donnez la définition d'une telle classe `Rider`.

Résultats Les résultats d'un coureur sont donnés par une instance de la classe `Result` dont le diagramme de classe est donné ci-dessous. La méthode `add()` permet de cumuler aux données de l'instance de `Result` sur lequel cette méthode est invoquée les données définies dans l'objet `result` passé en paramètre.

tourdefrance : :Result
...
+Result(t : Time, greenPoints : int, climbingPoints : int) +getTime() : Time +getGreenPoints() : int +getClimbingPoints() : int +add(result : Result)

Q 4 . Écrivez un test unitaire pour la méthode `add()`.

Étapes Une étape est caractérisée par un numéro d'ordre dans la course et ne peut être disputée (courue) qu'une seule fois. Au cours de cette étape les résultats de chaque coureur ayant terminé l'étape sont établis ainsi que l'ensemble des coureurs ayant abandonné au cours de l'étape. On suppose disposer de la classe `tourdefrance.Stage` avec au minimum les méthodes suivantes :

```
/**
 * Simulates the stage with the riders from <code>riders</code>. When this
 * method returns, this stage is considered run.
 *
 * @param riders
 *            the collection of riders starting this stage.
 */
public void run(Collection<Rider> riders) {
    ...
}

/**
 * Returns the results after this stage has been run.
 *
 * @return a map associating each rider finishing this stage to its result
 *         for this stage
 * @throws StageNotYetRunException
 *             when this stage has not been run yet
 */
public Map<Rider, Result> results() throws StageNotYetRunException {
    ...
}

/**
 * Returns the riders who have abandoned this stage.
 *
 * @return a collection of riders who abandoned during this stage.
 * @throws StageNotYetRunException
 *             when this stage has not been run yet
 */
public Collection<Rider> withdrawals() throws StageNotYetRunException {
    ...
}
```

Voici un début de spécification exécutable (sous la forme d'un test unitaire) qui vérifie que la méthode `results` lance bien une exception quand la course n'a pas encore été disputée :

```
@Test(expected=StageNotYetRunException.class)
public void testResultsThrowExceptionWhenStageNotRun() {
    new Stage().results();
}
```

Q 5 . Que signifie l'expression `expected=StageNotYetRunException.class` dans l'en-tête de la méthode ?

Q 6 . Que faut-il impérativement définir dans la classe `Rider`, pour que l'objet `Map` renvoyé par un appel à la méthode `results()` de `Stage` puisse être correctement exploité ?
Donnez le code associé à votre réponse.

Course à étapes Une course à étapes est définie par

- ▷ la liste des étapes qui la composent, ordonnées selon leur déroulement dans la course,
- ▷ la donnée des coureurs qui participent à cette course ainsi que leur résultat : cette information est rangée dans une table associant chaque coureur à son résultat.

On souhaite définir la classe `TourDeFrance` (une course à étapes) telle que :

- ▷ le constructeur prenne en paramètre une liste d'étapes et une liste de coureurs, chaque coureur aura initialement un temps de 0 seconde et 0 point dans chacun des classements,
- ▷ l'on ait une méthode `runAll()` qui fait disputer successivement et dans l'ordre chacune des étapes aux coureurs,
- ▷ l'on puisse récupérer la liste des coureurs encore en course ainsi que la liste de ceux ayant abandonné pendant les étapes déjà disputées.

Q 7 . Définissez la classe `TourDeFrance`.

Classements Un classement correspond au tri d'une liste de coureurs selon un critère choisi. Comme indiqué précédemment plusieurs classements sont établis en fonction des temps réalisés et des différents points acquis.
On souhaite donc ajouter à la classe `TourDeFrance` les méthodes suivantes :

- — `public Rider yellowJersey()` qui renvoie le coureur détenteur du maillot jaune. En cas d'égalité, n'importe quel coureur meilleur temps conviendra.

— `public Rider youngRider()` qui renvoie le meilleur jeune (moins de 25 ans) parmi les coureurs en course. En cas d'égalité entre deux jeunes, n'importe quel coureur jeune meilleur temps conviendra. Si aucun coureur de la course n'est jeune, cette méthode lève une exception `NoSuchElementException`.
— `public Rider greenRider()` qui renvoie le coureur détenteur du maillot vert. En cas d'égalité, n'importe quel coureur en tête du classement des sprinters conviendra.
— `public Rider polkaDotJersey()` qui renvoie le coureur détenteur du maillot à pois rouges. En cas d'égalité, n'importe quel coureur en tête du classement des grimpeurs conviendra.

La classe `java.util.Collections` possède les méthodes statiques suivantes qui peuvent être utiles :

```
public static <T> T min(Collection<? extends T> coll, Comparator<? super T> comp);
public static <T> T max(Collection<? extends T> coll, Comparator<? super T> comp);
```

qui retournent l'élément le plus petit (respectivement le plus grand) dans la collection `coll` selon la relation d'ordre définie par l'objet `comp` de type `Comparator`.
L'interface `Comparator` est définie ainsi (la javadoc de la méthode `compare` est donnée en annexe) :

```
package java.util;
public interface Comparator<T> {
  public int compare(T o1, T o2);
}
```

Q 8 . Faites une proposition d'implémentation de ces méthodes évitant au maximum la répétition de code.

Appendice

interface `java.lang.Comparable<T>`

```
public int compareTo(T o)
    Compares this object with the specified object for order. Returns  a
    negative integer, zero, or a positive integer as this object is less than,
    equal to, or greater than the specified object.

    The implementor must ensure sgn(x.compareTo(y)) ==
    -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must
    throw an exception iff y.compareTo(x) throws an exception.)

    The implementor must also ensure that the relation is transitive:
    (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

    Finally, the implementer must ensure that x.compareTo(y)==0 implies that
    sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all  z.

    It is strongly recommended, but not strictly required that
    (x.compareTo(y)==0) == (x.equals(y)). Generally speaking, any class that
    implements the Comparable interface and violates this condition should
    clearly indicate this fact. The recommended language is "Note: this class
    has a natural ordering that is inconsistent with equals."

    Parameters:
        o - the Object to be compared.
    Returns:
        a negative integer, zero, or a positive integer as this object is less
        than, equal to, or greater than the specified object.
    Throws:
        ClassCastException - if the specified object's type prevents
        it from being compared to this Object.
```

interface `java.util.Comparator<T>`

```
public int compare(T o1, T o2)
    Compares its two arguments for order. Returns a negative integer, zero, or a
    positive integer as the first argument is less than, equal to, or greater
    than the second.

    The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x))
    for all x and y. (This implies that compare(x, y) must throw an exception if
    and only if compare(y, x) throws an exception.)

    The implementor must also ensure that the relation is transitive:
    ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

    Finally, the implementer must ensure that compare(x, y)==0 implies that
    sgn(compare(x, z))==sgn(compare(y, z)) for all z.

    It is generally the case, but not strictly required that (compare(x, y)==0)
```

```
== (x.equals(y)). Generally speaking, any comparator that violates this
condition should clearly indicate this fact. The recommended language is
"Note: this comparator imposes orderings that are inconsistent with equals."

Parameters:
    o1 - the first object to be compared.
    o2 - the second object to be compared.
Returns:
    a negative integer, zero, or a positive integer as the first
    argument is less than, equal to, or greater than the second.
```

TD 3 — Transport

Au terme de ce TD, vous devriez être capable
- de concevoir une application objet avec de l'héritage ;
- de faire de l'héritage de classes abstraites pour réunir plusieurs propriétés et algorithmes communs au sein de mêmes classes ;
- d'utiliser le patron de conception *template method*.

À préparer avant le TD

Ce TD ne contient pas de travail préparatoire pour vous laisser le temps de travailler sur le projet.

À faire pendant le TD

Exercice 1 : Transport de marchandises
On souhaite modéliser en Java le calcul de coûts de transport de marchandises (*goods*) :

Goods
#weight : int #volume : int
+Goods(weight : int, volume : int) +getWeight() : int +getVolume() : int

Les marchandises sont transportées sous la forme de cargaisons (*shipment*). Les seules fonctionnalités publiques des cargaisons sont :

add qui permet d'ajouter une marchandise dans cette cargaison si cela est encore possible.

cost qui retourne, sous la forme d'un nombre entier d'euros, le coût total du transport de cette cargaison.

Une cargaison est également caractérisée par la distance sur laquelle elle est transportée. Ce renseignement est communiqué à la construction de la cargaison sous la forme d'un nombre entier de kilomètres. On précise qu'une cargaison ne peut réunir qu'une quantité limitée de marchandises (*quantity*) : cette quantité est exprimée soit en termes de poids (*weight*) soit de volume (*volume*), selon le type de transport utilisé. Ce dernier influe aussi sur le calcul du coût de transport de la cargaison qui, de la même façon, dépend de la quantité des marchandises de la cargaison. On distingue donc plusieurs types de cargaisons selon le moyen de transport utilisé :

Type de cargaison	Quantité	Coût	Limite
Fluviale (*fluvial*)	`poids`	distance $\times \sqrt{\text{quantité}}$	quantité ≤ 300000
Routière (*road*)	`poids`	$4 \times$ distance $\times$ quantité	quantité ≤ 38000
Aérienne (*air*)	`volume`	$10 \times$ distance $+ 4 \times$ quantité	quantité ≤ 80000
Aérienne urgente (*urgent*)	`volume`	$2 \times$ coût d'une cargaison aérienne	quantité ≤ 80000

Q 1 . Identifiez les caractéristiques communes des types de transport.

Q 2 . Proposez une hiérarchie de classes réunissant les caractéristiques communes et séparant les caractéristiques différentes.

Q 3 . Donner l'implémentation de la classe `Shipment` ainsi que de ses sous-classes.

TD 4 — Actions

Au terme de ce TD, vous devriez être capable
- de modéliser le temps avec des objets ;
- de tester une abstraction ;
- de différencier les différents types d'exceptions ;
- de créer de nouvelles classes d'exception ;
- de créer de fausses entités pour isoler une entité à tester ;
- de lire et de restructurer du code existant ;
- d'utiliser le patron de conception *composite* ;
- d'utiliser le patron de conception *factory method*.

À préparer avant le TD

Q 1 . Lire la fiche sur le patron de conception *template method* et indiquer comment ce patron s'applique dans le TD précédent.

Q 2 . Lire au moins les sections suivantes sur les exceptions Java : `http://docs.oracle.com/javase/tutorial/essential/exceptions` :
- *What Is an Exception ?*
- *The Catch or Specify Requirement*
- *Specifying the Exceptions Thrown by a Method*
- *Creating Exception Classes*
- *Unchecked Exceptions — The Controversy*

Dans ce TD nous définissons les notions générales d'action et d'ordonnanceur d'actions (*scheduler*). Dans le TD suivant nous exploiterons ces notions pour réaliser une application de simulation d'utilisation de ressources partagées connue sous le nom du *problème de la piscine*.
Nous définissons une action comme un objet qui progresse de son état initial (*ready*) jusqu'à son état final (*finished*) par *appels successifs* de sa méthode `doStep()`. Une action commencée se trouve dans l'état *en cours* (*in progress*) tant qu'elle n'est pas terminée.
Certaines actions passent de l'état initial à l'état final par une seule exécution de `doStep()`, d'autres peuvent nécessiter plusieurs exécutions de cette méthode.
On peut considérer d'une certaine manière que le nombre de fois où il faut appeler la méthode `doStep()` correspond au "temps que prend l'action à s'exécuter". Chaque appel de `doStep()` représente le passage d'une unité de temps pendant laquelle une et une seule action peut avancer d'une et une seule étape.
Certaines actions ne sont terminées qu'au bout d'un nombre paramétrable d'invocations de `doStep()` (*foreseeable action*). Une action de ce type prend donc un "certain temps à s'exécuter".
Un ordonnanceur possède un ensemble d'actions et on peut s'adresser à lui pour faire progresser ces actions. En ce sens un ordonnanceur est une action dont la méthode `doStep()` consiste à faire progresser la prochaine action prévue.
Un ordonnanceur peut être vu comme une action composée : ses *sous-actions* peuvent contenir des ordonnanceurs.
Un ordonnanceur est terminé quand toutes ses sous actions sont terminées. Il faut prévoir la possibilité d'ajouter une nouvelle sous-action aux ordonnanceurs. Chaque appel à `doStep()` sur un ordonnanceur fait seulement *avancer une des actions* d'une et une seule étape. Il faudra en général appeler de nombreuses fois `doStep()` pour terminer toutes les sous-actions.

Q 3 . Le code suivant est une implémentation possible de la notion d'action. Lire ce code, lister ses problèmes et proposer une solution en quelques mots.

```
public class BadActionTest {

  private BadAction createAction(int timeToEnd) {
    return new BadAction(timeToEnd);
  }

  @Test
  public void foreseeableAction() {
    BadAction action = createAction(2);

    // 2 steps remaining
    assertTrue(action.isReady());
    assertFalse(action.isInProgress());
    assertFalse(action.isFinished());

    action.doStep();

    // 1 step remaining
    assertFalse(action.isReady());
```

```
  assertTrue(action.isInProgress());
  assertFalse(action.isFinished());

  action.doStep();

  // 0 step remaining
  assertFalse(action.isReady());
  assertFalse(action.isInProgress());
  assertTrue(action.isFinished());
}

@Test
public void scheduler() {
  BadAction action1 = createAction(2);
  BadAction action2 = createAction(1);

  BadAction scheduler = createAction(0);
  scheduler.addAction(action1);
  scheduler.addAction(action2);

  assertTrue(action1.isReady());
  assertTrue(action2.isReady());

  scheduler.doStep();

  assertTrue(action1.isInProgress());
  assertTrue(action2.isReady());

  scheduler.doStep();

  assertTrue(action1.isFinished());
  assertTrue(action2.isReady());

  scheduler.doStep();

  assertTrue(action1.isFinished());
  assertTrue(action2.isFinished());
}

@Test
public void schedulerWithScheduler() {
  BadAction action1 = createAction(2);

  BadAction subScheduler = createAction(0);
  BadAction scheduler = createAction(0);
  subScheduler.addAction(action1);
  scheduler.addAction(subScheduler);

  assertTrue(action1.isReady());
  assertTrue(subScheduler.isReady());

  scheduler.doStep();

  assertTrue(action1.isInProgress());
  assertTrue(subScheduler.isInProgress());

  scheduler.doStep();

  assertTrue(action1.isFinished());
  assertTrue(subScheduler.isFinished());
}

@Test
public void onlyOneValidStateAtEachMomentForForeseebleAction() {
  onlyOneValidStateAtEachMoment(createAction(10));
}

@Test
public void onlyOneValidStateAtEachMomentForScheduler() {
  BadAction scheduler = createAction(0);
  scheduler.addAction(createAction(1));
  onlyOneValidStateAtEachMoment(scheduler);
}

protected void onlyOneValidStateAtEachMoment(BadAction action) {
  assertTrue(action.isReady());
  assertFalse(action.isInProgress());
  assertFalse(action.isFinished());

  while (!action.isFinished()) {
```

```
      action.doStep();
      assertFalse(action.isReady());
      // isFinished xor isInProgress
      assertTrue(action.isFinished() == !action.isInProgress());
    }

    assertFalse(action.isReady());
    assertFalse(action.isInProgress());
    assertTrue(action.isFinished());
  }
}
```

```
public class BadAction {

  protected final int totalTime;
  protected int remainingTime;

  protected boolean isReady = true;
  protected boolean isInitialized = false;

  protected final boolean isScheduler;

  // this variable is only used for schedulers (isScheduler = true)
  protected final ArrayList<BadAction> actions = new ArrayList<BadAction>();

  /**
   * Either create a foreseeable action or a scheduler based on the value of
   * <code>timeToEnd</code>.
   *
   * @param timeToEnd
   *            For a foreseeable action, indicate the number of
   *            <code>doStep()</code> calls required. For a scheduler, must be
   *            0.
   */
  public BadAction(int timeToEnd) {
    this.totalTime = timeToEnd;

    if (timeToEnd == 0) {
      isScheduler = true;
    } else {
      isScheduler = false;
      this.remainingTime = timeToEnd;
    }
  }

  public boolean isReady() {
    if (!isScheduler) {
      return remainingTime == totalTime;
    } else {
      return isInitialized && isReady;
    }
  }

  public boolean isInProgress() {
    if (!isScheduler) {
      return !isReady() && !isFinished();
    } else {
      return isInitialized && !isReady() && !isFinished();
    }
  }

  public boolean isFinished() {
    if (!isScheduler) {
      return remainingTime <= 0;
    } else {
      return isInitialized && !isReady() && actions.isEmpty();
    }
  }

  public void doStep() {
    if (!isScheduler) {
      remainingTime--;
    } else {
      isReady = false;
      BadAction nextAction = actions.get(0);
      nextAction.doStep();
      if (nextAction.isFinished()) {
```

```
        actions.remove(0);
      }
    }
  }

  public void addAction(BadAction subAction) {
    isInitialized = true;
    if (subAction.isFinished()) {
      throw new IllegalArgumentException(
          "Can't add an already finished action");
    }
    if (isFinished()) {
      throw new IllegalStateException(
          "You can't add an action to a finished scheduler");
    } else {
      actions.add(subAction);
    }
  }
}
```

À faire pendant le TD

Q 4 . Proposez une nouvelle architecture permettant de pouvoir ajouter de nouveaux types d'actions au delà des *foreseeable* et des ordonnanceurs. Votre nouvelle architecture doit continuer à faire passer les tests de la classe `BadActionTest` sans la changer (à la construction des instances de `BadAction` près).

Q 5 . Proposez une nouvelle architecture pour les tests de la classe `BadActionTest`. En particulier, réfléchissez au placement du test de `onlyOneValidStateAtEachMoment()` qui devra être réutilisé, sans duplication, pour chaque classe d'actions, y compris celles non encore écrites.

Dans la définition d'ordonnanceur ci-dessus, il peut exister plusieurs interprétations possibles à l'expression "la prochaine action prévue" :

- *séquentiel* (*sequential scheduler*) : dans ce cas, la prochaine action prévue est celle qui vient d'être exécutée si elle n'est pas encore terminée et la suivante sinon. Ce type d'ordonnanceur termine donc l'action commencée avant de passer à la suivante ;
- *en temps partagé* (*fair scheduler*) : ici, la prochaine action prévue est celle qui suit celle qui vient d'être exécutée. La progression des actions se fait en *parallèle*. L'ordonnanceur dispose d'une action courante a (non terminée) dont il invoque la méthode `doStep()`. Lors du prochain appel de la méthode `doStep()` de l'ordonnanceur, celui-ci fera avancer la première action non terminée qui suit a. L'ordonnanceur reviendra à a quand il aura invoqué une fois la méthode `doStep()` sur chacune de ses actions. Puis il fera avancer la première action non terminée qui suit a et ainsi de suite.

Q 6 . Revoyez votre architecture pour prendre en compte le point ci-dessus.

Quand un test échoue, on aime savoir rapidement pourquoi il a échoué et comment résoudre le problème. C'est pourquoi, quand on écrit un test d'une classe, on essaye de la tester en isolation des autres classes. Quand on respecte cette pratique, si un test échoue, on sait que le problème est soit dans la classe testée, soit dans le test lui même : il ne peut pas être ailleurs dans le système. Pour arriver à cette isolation, il peut-être nécessaire de créer des *fausses* entités dont la classe à tester va dépendre : ces entités vont faire office de dépendances contrôlées. Ces fausses entités sont appelées *Test Double*, *Mock* ou *Stub* (avec quelques différences entre ces notions).

Q 7 . Donnez le code d'une fausse entité `Action` pour pouvoir tester les ordonnanceurs. Cette fausse entité devra permettre d'écrire ce test :

```
@Test
public void with1OneStepAction() {
  OneStepAction action1 = new OneStepAction();
  Scheduler scheduler = createScheduler(action1);

  assertFalse(scheduler.isFinished());
  assertFalse(action1.isFinished());

  scheduler.doStep();

  assertTrue(scheduler.isFinished());
  assertTrue(action1.isFinished());
}
```

Q 8 . Donner les tests permettant de vérifier la spécification des ordonnanceurs. Beaucoup de tests sont communs aux deux types d'ordonnanceurs et doivent être partagés.

Q 9 . Donner le code Java des ordonnanceurs. Faites attention à ce que le maximum de code soit partagé.

Une fois une action terminée, l'exécution de la méthode `doStep()` entraîne la levé d'une exception `ActionFinishedException`. Cette spécification peut être vérifiée grâce à un test comme celui-ci :

```
public abstract class ActionTest {

  protected abstract Action createAction();

  @Test(expected = ActionFinishedException.class, timeout = 2000)
  public void doStepWhileFinishedThrowsException() throws ActionFinishedException {
    Action action = createAction();
    while (!action.isFinished()) {
      try {
        action.doStep();
      } catch (ActionFinishedException e) {
        fail("action was not supposed to be finished, we just checked");
      }
    }

    assertTrue(action.isFinished());
    action.doStep();
  }
}
```

La méthode `createAction()` doit être déclarée abstraite pour pouvoir être implémentée différemment suivant le type d'action à tester. Cette méthode va retourner une nouvelle instance de la classe d'actions à tester.

Q 10 . Donner un code JAVA pour la classe `ActionFinishedException`.

Q 11 . Mettre à jour votre code pour que le test ci-dessus passe.

TD 5 — Problème de la piscine

Au terme de ce TD, vous devriez être capable
- d'utiliser le patron de conception *factory method* ;
- de comprendre l'intérêt des classes génériques et de pouvoir les utiliser dans un contexte simple ;
- de modéliser un problème système (la gestion des ressources) avec des objets ;
- de combiner les fonctionnalités de plusieurs classes grâce à la délégation.

À préparer avant le TD

Q 1 . Revoir et finir (si nécessaire) le TD sur les actions.

Q 2 . Lire la fiche sur le patron de conception *composite* et indiquer comment ce patron s'applique dans le TD précédent.

Q 3 . Lire au moins les sections suivantes sur les génériques JAVA : `http://docs.oracle.com/javase/tutorial/java/generics` :
- *Why Use Generics ?*
- *Generic Types*
- *Raw Types*
- *Bounded Type Parameters*

Après avoir défini les notions générales d'*action* et d'*ordonnanceur d'actions* dans le TD précédent, nous exploitons maintenant ces notions pour réaliser une application de simulation d'utilisation de ressources partagées connue sous le nom du *problème de la piscine* (*pool of resources*).
Une ressource (*resource*) est un objet dont la classe implémente l'interface suivante :

```
public interface Resource {
   public String description();
}
```

Un gestionnaire de ressources dispose d'un certain nombre initial, paramétrable, de ressources. Ces ressources sont créées à la construction du gestionnaire, on parle de *pool* de ressources.
On doit pouvoir demander à un gestionnaire de ressources (`ResourcePool`) de fournir une ressource de son pool (en étant informé si aucune ressource n'est disponible) et on doit pouvoir l'informer qu'une ressource est libérée. Cette classe disposera (au moins) des méthodes suivantes :

▷ `provideResource` pour prendre une des ressources du pool ; cette méthode a pour résultat la ressource prise. Une exception `NoSuchElementException` est levée s'il n'y a pas de ressource disponible.

▷ `freeResource` pour indiquer qu'une ressource du pool a été libérée. Cette ressource est passée en paramètre. Une exception `IllegalArgumentException` est levée si la ressource n'est pas une ressource gérée par le gestionnaire.

Q 4 . Écrivez un ensemble de tests unitaires pour cette classe ;

À faire pendant le TD

Q 5 . Donnez le code d'une classe `ResourcePool` pour représenter ces gestionnaires. Chaque instance de cette classe ou sous classe pourra être *spécialisée* pour un type de ressource particulier (sous type de `Resource`).

Q 6 . Donnez le code d'une classe `BasketPool` qui fournit des ressources de type `Basket` (un *panier* pour mettre des vêtements), une classe supposée définie implémentant l'interface `Resource`.

On souhaite définir une classe `TakeResourceAction` et une classe `FreeResourceAction` qui sont toutes les deux des actions (voir TD précédent). Ces actions vont s'adresser à un gestionnaire de ressources communiqué à la construction pour, comme on peut le deviner, respectivement prendre une ressource et libérer un ressource. L'utilisateur de la ressource, de type `ResourcefulUser` (son code est fourni en annexe), est également communiqué à la construction de l'action. C'est cet utilisateur qui prend et libère la ressource.
Les instances de `TakeResourceAction` ne se terminent que quand la requête correspondante a abouti, c'est-à-dire que l'utilisateur a effectivement réussi à prendre une ressource. Il peut ne pas y avoir de ressource disponible à la première tentative. Une action peut donc prendre un certain temps, c'est-à-dire nécessiter plusieurs appels à `doStep()` avant d'être terminée.

Q 7 . Modélisez les classes `TakeResourceAction` et `FreeResourceAction` qui permettent à un `ResourcefulUser` de prendre, respectivement libérer, une ressource fournie par un `ResourcePool`.

Indiquez comment ces classes se raccrochent aux éléments précédents.

Q 8 . Donnez les tests spécifiant les classes `TakeResourceAction` et `FreeResourceAction`. Les tests spécifiant la classe `Action` doivent pouvoir être appliqués sur ces 2 nouveaux types d'action.

Q 9 . Donnez le code associé.

Réalisation d'une simulation : tous à la piscine ! Nous allons utiliser les différentes notions mises en œuvre dans les questions précédentes pour réaliser une simulation.
Chaque personne voulant accéder à la piscine doit dérouler le scénario suivant : prendre un panier (*basket*), aller dans une cabine (*cubicle*), se déshabiller (*getting undressed*), libérer la cabine, se baigner (*swim*), retrouver une cabine, s'habiller (*getting dressed*), libérer sa cabine et rendre son panier.
Panier et cabine sont des ressources.
Un nageur (*swimmer*) est caractérisé par (même ordre que dans les appels au constructeur du code ci-après) :

1. son nom,
2. le gestionnaire des paniers à qui s'adresser,
3. le gestionnaire des cabines à qui s'adresser,
4. le temps qui lui est nécessaire pour se déshabiller,
5. la durée pendant laquelle il va se baigner,
6. le temps qui lui est nécessaire pour se rhabiller.

Il suffit, maintenant, qu'un ordonnanceur s'occupe de faire agir les différents nageurs en temps partagé pour produire la simulation du fonctionnement d'une piscine, comme cela est fait dans la classe `Pool` décrite ci-dessous :

```
public class Pool {

  public static void main(String[] args) throws ActionFinishedException {
    BasketPool baskets = new BasketPool(6);
    CubiclePool cubicles = new CubiclePool(3);
    FairScheduler s = new FairScheduler();

    s.addAction(new Swimmer("Camille", baskets, cubicles, 6, 4, 8));
    s.addAction(new Swimmer("Loïs", baskets, cubicles, 2, 10, 4));
    s.addAction(new Swimmer("Maé", baskets, cubicles, 10, 18, 10));
    s.addAction(new Swimmer("Ange", baskets, cubicles, 3, 7, 5));
    s.addAction(new Swimmer("Louison", baskets, cubicles, 18, 3, 3));
    s.addAction(new Swimmer("Charlie", baskets, cubicles, 3, 6, 10));
    s.addAction(new Swimmer("Alexis", baskets, cubicles, 6, 5, 7));

    int nbSteps = 0;
    while (!s.isFinished()) {
      nbSteps++;
      s.doStep();
    }
    System.out.println("Finished in " + nbSteps + " steps");
  }

}
```

Q 10 . Faites un diagramme de classes UML donnant une vue globale de l'architecture de cette simulation ;

Q 11 . Donnez le code Java de la classe `Swimmer`.

Appendice

Class `ResourcefulUser`

```
public class ResourcefulUser<R extends Resource> {

  protected R resource;

  public R getResource() {
    return resource;
  }

  public void setResource(R resource) {
    this.resource = resource;
  }

  public void resetResource() {
    this.resource = null;
  }

}
```

Trace d'exécution

Voici un bout de trace d'exécution pour la simulation :

```
Camille's turn
 Camille trying to take resource from pool basket... success
Loïs's turn
 Loïs trying to take resource from pool basket... success
Maé's turn
 Maé trying to take resource from pool basket... success
Ange's turn
 Ange trying to take resource from pool basket... success
Louison's turn
 Louison trying to take resource from pool basket... success
Charlie's turn
 Charlie trying to take resource from pool basket... success
Alexis's turn
 Alexis trying to take resource from pool basket... failed
Camille's turn
 Camille trying to take resource from pool cubicle... success
Loïs's turn
 Loïs trying to take resource from pool cubicle... success
Maé's turn
 Maé trying to take resource from pool cubicle... success
Ange's turn
 Ange trying to take resource from pool cubicle... failed
Louison's turn
 Louison trying to take resource from pool cubicle... failed
Charlie's turn
 Charlie trying to take resource from pool cubicle... failed
Alexis's turn
 Alexis trying to take resource from pool basket... failed
Camille's turn
 undressing (1/6)
Loïs's turn
 undressing (1/2)
Maé's turn
 undressing (1/10)
Ange's turn
 Ange trying to take resource from pool cubicle... failed
Louison's turn
 Louison trying to take resource from pool cubicle... failed
Charlie's turn
 Charlie trying to take resource from pool cubicle... failed
Alexis's turn
 Alexis trying to take resource from pool basket... failed
Camille's turn
 undressing (2/6)
Loïs's turn
 undressing (2/2)
Maé's turn
 undressing (2/10)
Ange's turn
[...]
Maé's turn
 Maé freeing resource from pool basket
Alexis's turn
 dressing (4/7)
Alexis's turn
 dressing (5/7)
Alexis's turn
 dressing (6/7)
Alexis's turn
 dressing (7/7)
Alexis's turn
 Alexis freeing resource from pool cubicle
Alexis's turn
 Alexis freeing resource from pool basket
Finished in 242 steps
```

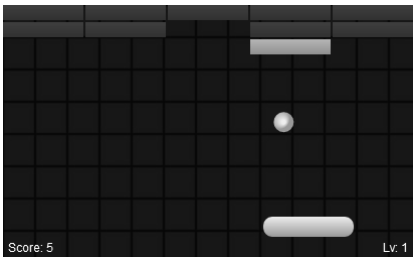
TD 6 — Casse-briques

Au terme de ce TD, vous devriez être capable
- de faire des arbres d'héritage parallèles ;
- de comprendre l'intérêt du *double dispatch* ;
- d'utiliser le patron de conception *decorator* ;
- d'utiliser le patron de conception *abstract factory*.

À préparer avant le TD

Q 1 . Lire la fiche sur le patron de conception *factory method* et indiquer comment ce patron s'applique dans le TD précédent ;

On s'intéressera dans cet exercice à la modélisation d'un certain nombre d'entités d'un jeu de casse-briques (*breakout-style game*).
Ce jeu se joue seul et le joueur contrôle une raquette (*paddle*) qu'il peut déplacer latéralement. La raquette permet de renvoyer une balle (*ball*) vers des briques (*blocks*) qui sont cassées lorsqu'elles sont frappées par la balle, tout en renvoyant la balle. Le but est de détruire toutes les briques sans laisser tomber la balle. Le joueur dispose de quelques balles en réserve qu'il peut utiliser lorsqu'il perd une balle lui donnant ainsi quelques droits à l'erreur.



Ce jeu d'arcade fait partie des ancêtres des jeux vidéos et a connu des variantes dans lesquelles les briques sont de différents types et déclenchent des effets variables lorsqu'elles sont détruites.
La classe `Player` modélise l'entité joueur du casse-briques, avec notamment sa raquette (la taille pouvant varier) et ses balles en réserve.
Les briques sont caractérisées par un nombre de points (reçus par le joueur lorsqu'il casse la brique) et par l'effet que leur destruction a sur le joueur.
On trouve les types de briques suivants :
- les briques *simple* rapportent 100 points et n'ont aucun effet particulier autre qu'ajouter les points au score du joueur ;
- les briques *jackpot* rapportent aléatoirement de 1 à 10 fois plus de points qu'une brique de base et n'ont aucun effet particulier ;
- les briques *grande raquette* rapportent autant qu'une brique simple et doublent la taille de la raquette ;
- les briques *bonus balle* rapportent 3 fois plus qu'une brique simple et augmentent de 1 le nombre de balles en réserve du joueur ;
- les briques *piège* rapportent 0 point et ont pour effet d'enlever un nombre de points variable, précisé à la construction de la brique, et de supprimer une balle de la réserve du joueur (s'il lui en reste).

D'autres types de briques doivent pouvoir être envisagés et ajoutés facilement.

Q 2 . Donnez un diagramme de classes UML.

À faire pendant le TD

À chaque type de brique correspond une représentation graphique (couleur, forme, ...) disposant d'une méthode l'affichant sur un canevas. Pour pouvoir garder un modèle (les briques, le joueur, la raquette) stable lorsque la représentation graphique change, il est important de ne pas introduire de dépendance de la partie modèle vers la partie graphique.

Q 3 . Mettez à jour votre diagramme.

Q 4 . Indiquez comment, à partir d'une liste de briques (décrivant un niveau), vous créez une liste de représentations graphiques de ces briques.

Q 5 . Donnez les tests et le code Java représentant les briques *grande raquette* et *bonus balle*. Le cas échéant vous donnerez également le code des super types nécessaires.

On propose d'enrichir les briques possibles : pour chaque type de brique déjà proposé, une variante est créée. Cette variante appelée *double effet* (*double-effect block*) a comme conséquence de doubler le nombre de points attribués pour chacun des types ainsi que l'effet appliqué. Ainsi :

— une brique *bonus balle double effet* rapportera 6 fois plus qu'une brique simple et 2 nouvelles balles pour le joueur ;
— une brique *jackpot double effet* rapportera aléatoirement de 2 à 20 fois plus de points qu'une brique de base et n'aura pas d'autre effet.

Évidemment pour chaque nouveau type de brique que l'on peut imaginer on veut pouvoir en avoir une version double effet. Graphiquement, une brique *double effet* se traduit par un objet graphique dont l'affichage est le même que celui de la brique de base avec en plus un halo lumineux.

Q 6 . Adaptez votre proposition de conception pour prendre en compte les briques *double effet*.

Q 7 . Donnez le code JAVA pour les briques *double effet* (sans l'aspect représentation graphique).

Q 8 . Donnez le code JAVA permettant de créer une *brique bonus balle double effet*.

TD 7 — Courriers

Au terme de ce TD, vous devriez être capable
- d'utiliser le patron de conception *decorator* ;
- de concevoir des applications simples évoluant par extension et non par modification (principe ouvert/fermé, *OCP = Open/Closed Principle*) ;
- de concevoir des classes génériques dont les paramètres sont eux-mêmes génériques.

À préparer avant le TD

Q 1 . Lire la fiche sur le patron de conception *decorator* et indiquer comment ce patron s'applique dans le TD précédent.

Q 2 . Lire la fiche sur le patron de conception *abstract factory* et indiquer comment ce patron s'applique dans le TD précédent.

Q 3 . Lire de la documentation sur OCP, par exemple au moins les 5 premières pages de `http://www.objectmentor.com/resources/articles/ocp.pdf` (ce document est pour C++, mais s'adapte très bien à Java).

Q 4 . Lire au moins les sections suivantes sur les génériques Java : `http://docs.oracle.com/javase/tutorial/java/generics` :
- *Wildcards*
- *Upper Bounded Wildcards*
- *Unbounded Wildcards*

Ce sujet va se faire en versions successives. Chaque version doit être complètement réalisée avant de passer à la suivante. Une bonne conception doit faciliter l'intégration des extensions apportées par les versions successives.

À faire pendant le TD

On se propose de modéliser la distribution du courrier : chaque jour, des habitants envoient des courriers à d'autres habitants.
Ces courriers doivent être mis dans des boîtes aux lettres et le courrier est distribué le lendemain.
Dans cet exercice, les villes sont considérées comme des bureaux distributeurs : une ville réunit toutes les lettres postées le jour même par ses habitants et les distribuent le lendemain aux destinataires, où qu'ils soient.
Une ville (*city*) possède un nom et des habitants (*inhabitant*).
- La méthode `City.sendLetter()` ajoute un courrier dans la boîte aux lettres de la ville (*postbox*).
- La méthode `City.distributeLetters()` s'occupe de la distribution des courriers de la boîte aux lettres de la ville.

Les habitants sont caractérisés par un nom, une ville et un compte en banque (*bank account*). On doit pouvoir créditer (*credit*) ou débiter (*debit*) ce compte d'un montant (*amount*) donné.
Un courrier comporte un expéditeur (*sender*) et un destinataire (*receiver*) (deux habitants) et un contenu (*content*, a priori, n'importe quoi). À tout courrier est associée une action, qui sera accomplie lorsque ce courrier sera reçu par son destinataire. De plus tout courrier a un coût.

Q 5 . Établissez un diagramme de classes UML représentant les différents types et leurs relations.

Version 1.0 Dans cette première version, un courrier peut être :
- Une lettre simple (*simple letter*) : son contenu est un texte. Le coût d'une lettre simple est fixe (1 par exemple).
- Une lettre de change (*promissory note*) : elle contient une certaine somme d'argent, qui provient du compte de l'expéditeur, et sera versée au compte du destinataire. Le destinataire, reconnaissant, envoie alors une lettre de remerciement à son bienfaiteur. Le coût d'une lettre de change est de 1 + 1% de la somme transférée.

Q 6 . Modélisez ces types sous la forme d'un diagramme de classes UML.

Q 7 . Implémentez les 2 tests suivants :
1. tout courrier a un coût strictement positif ;
2. dès la compilation, toute lettre simple a un contenu de type texte.

Q 8 . Donnez l'en-tête des classes `Letter` et `SimpleLetter`.

Version 2.0 Il doit maintenant être possible d'envoyer tout courrier *en recommandé* (*registered letter*). Dans ce cas, *en plus* de l'effet lié au courrier, un accusé de réception (*acknowledgment of receipt*) est alors envoyé à l'expéditeur en retour. Un recommandé a un surcoût de 15 par rapport au courrier initial.

Q 9 . Modélisez les courriers recommandés, comment s'intègrent-ils à la version 1.0 ?

Q 10 . Donnez l'en-tête de la classe `RegisteredLetter`.

Un test unitaire pour l'envoi de l'accusé de réception pourrait ressembler à :

```
public class RegisteredLetterTest ... {
  @Test
  public void receiverSendsAcknowledgment() {
    assertEquals(0, receiver.numberOfLetterSent);
    createLetter().doAction();
    assertEquals(1, receiver.numberOfLetterSent);
  }
}
```

La variable `numberOfLetterSent` ne *doit pas* être ajoutée à la classe `Inhabitant`, cette variable n'étant utile qu'aux tests.

Q 11 . Implémentez les parties manquantes de la classe `RegisteredLetterTest` pour faire passer le test ci-dessus. En plus des tests spécifiques à la classe `RegisteredLetter`, les tests du type `Letter` doivent pouvoir être exécutés sur les instances de `RegisteredLetter`.

Version 3.0 On ajoute maintenant la prise en compte de *courriers urgents* (*urgent letter*). Tout courrier peut être transformé en courrier urgent (y compris les recommandés). L'action est inchangée et le coût est doublé.

Q 12 . Comment prendre en compte cette modification ? Faites une proposition qui s'intègre à la version 2.0.

Q 13 . Peut on utiliser le système de types pour faire interdire, à la compilation, la création d'un courrier urgent qui contiendrait un autre courrier urgent ? Et pour les urgents qui contiennent un recommandé qui contient un urgent ?

Q 14 . À quoi peut bien servir un test comme celui ci-dessous ? À quoi correspond l'annotation `@SuppressWarnings` ?

```
public class UrgentLetterTest ... {

  @Test
  public void whatAmITesting() {
    SimpleLetter simpleLetter = new SimpleLetter(sender, receiver);
    UrgentLetter<SimpleLetter> letter = new UrgentLetter<SimpleLetter>(simpleLetter);

    @SuppressWarnings("unused")
    Text text = letter.getContent().getContent();
  }
}
```

Annexe

Votre code doit générer La trace d'exécution ci-dessous qui suit le cahier des charges suivant :

- Création d'une ville,
- Création de *100 habitants* dans cette ville.
- Chaque jour, pendant k jours ($k = 6$ dans la trace ci-dessous), un *nombre aléatoire* d'habitants envoie un courrier (d'un *type aléatoire*) à un autre *habitant aléatoire* de la ville. Éventuellement, ces courriers peuvent engendrer des réponses (accusés de réception, lettres de remerciement, etc.) qui sont donc distribuées le jour suivant. L'application simule la collecte et la distribution du courrier tant qu'il reste des courriers à distribuer. Cette simulation se déroule donc éventuellement sur plus de k jours (comme c'est le cas sur la trace fournie).

Note : Le caractère "urgent" des courriers n'est pas pris en compte dans cette simulation : urgent ou non, un courrier arrive à destination le lendemain du jour où il est posté ; simplement un courrier urgent coûte le double.

```
Creating Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch city
Creating 100 inhabitants
Mailing letters for 6 days
************************************************
Day 1
-> inhabitant-17 mails an urgent letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-1 for a cost of 2 euros
   - 2 euros are debited from inhabitant-17 account whose balance is now 4998 euros
-> inhabitant-35 mails a simple letter whose content is a text content (bla bla) to inhabitant-10 for a cost of 1 euro
   - 1 euro is debited from inhabitant-35 account whose balance is now 4999 euros
-> inhabitant-46 mails a promissory note letter whose content is a money content (18) to inhabitant-9 for a cost of 1 euro
   - 1 euro is debited from inhabitant-46 account whose balance is now 4999 euros
-> inhabitant-87 mails a promissory note letter whose content is a money content (94) to inhabitant-50 for a cost of 1 euro
   - 1 euro is debited from inhabitant-87 account whose balance is now 4999 euros
-> inhabitant-35 mails a simple letter whose content is a text content (bla bla) to inhabitant-25 for a cost of 1 euro
   - 1 euro is debited from inhabitant-35 account whose balance is now 4998 euros
************************************************
Day 2
<- inhabitant-1 receives an urgent letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-17
```

```
<- inhabitant-10 receives a simple letter whose content is a text content (bla bla) from inhabitant-35
<- inhabitant-9 receives a promissory note letter whose content is a money content (18) from inhabitant-46
   - 18 euros are debited from inhabitant-46 account whose balance is now 4981 euros
   + inhabitant-9 account is credited with 18 euros; its balance is now 5018 euros
-> inhabitant-9 mails a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (18)) to inhabitant-9 for a cost of 1 euro
   - 1 euro is debited from inhabitant-9 account whose balance is now 5017 euros
<- inhabitant-50 receives a promissory note letter whose content is a money content (94) from inhabitant-87
   - 94 euros are debited from inhabitant-87 account whose balance is now 4905 euros
   + inhabitant-50 account is credited with 94 euros; its balance is now 5094 euros
-> inhabitant-50 mails a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (94)) to inhabitant-50 for a cost of 1 euro
   - 1 euro is debited from inhabitant-50 account whose balance is now 5093 euros
<- inhabitant-25 receives a simple letter whose content is a text content (bla bla) from inhabitant-35
-> inhabitant-87 mails a promissory note letter whose content is a money content (731) to inhabitant-28 for a cost of 8 euros
   - 8 euros are debited from inhabitant-87 account whose balance is now 4897 euros
-> inhabitant-62 mails a registered letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-55 for a cost of 16 euros
   - 16 euros are debited from inhabitant-62 account whose balance is now 4984 euros
-> inhabitant-33 mails an urgent letter whose content is a registered letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-41 for a cost of 32 euros
   - 32 euros are debited from inhabitant-33 account whose balance is now 4968 euros
**************************************************
Day 3
<- inhabitant-9 receives a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (18)) from inhabitant-46
<- inhabitant-50 receives a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (94)) from inhabitant-87
<- inhabitant-28 receives a promissory note letter whose content is a money content (731) from inhabitant-87
   - 731 euros are debited from inhabitant-87 account whose balance is now 4166 euros
   + inhabitant-28 account is credited with 731 euros; its balance is now 5731 euros
-> inhabitant-28 mails a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (731)) to inhabitant-28 for a cost of 1 eur
   - 1 euro is debited from inhabitant-28 account whose balance is now 5730 euros
<- inhabitant-55 receives a registered letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-62
-> inhabitant-55 mails an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content is a
   - 1 euro is debited from inhabitant-55 account whose balance is now 4999 euros
<- inhabitant-41 receives an urgent letter whose content is a registered letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-33
-> inhabitant-41 mails an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content is a
   - 1 euro is debited from inhabitant-41 account whose balance is now 4999 euros
-> inhabitant-24 mails a simple letter whose content is a text content (bla bla) to inhabitant-6 for a cost of 1 euro
   - 1 euro is debited from inhabitant-24 account whose balance is now 4999 euros
-> inhabitant-66 mails a simple letter whose content is a text content (bla bla) to inhabitant-89 for a cost of 1 euro
   - 1 euro is debited from inhabitant-66 account whose balance is now 4999 euros
-> inhabitant-35 mails an urgent letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-53 for a cost of 2 euros
   - 2 euros are debited from inhabitant-35 account whose balance is now 4996 euros
**************************************************
Day 4
<- inhabitant-28 receives a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (731)) from inhabitant-87
<- inhabitant-55 receives an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content i
<- inhabitant-41 receives an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content i
<- inhabitant-6 receives a simple letter whose content is a text content (bla bla) from inhabitant-24
<- inhabitant-89 receives a simple letter whose content is a text content (bla bla) from inhabitant-66
<- inhabitant-53 receives an urgent letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-35
-> inhabitant-39 mails a simple letter whose content is a text content (bla bla) to inhabitant-55 for a cost of 1 euro
   - 1 euro is debited from inhabitant-39 account whose balance is now 4999 euros
-> inhabitant-22 mails an urgent letter whose content is a registered letter whose content is a promissory note letter whose content is a money content (714) to inhabitant-56 for a cost of 46 euros
   - 46 euros are debited from inhabitant-22 account whose balance is now 4954 euros
-> inhabitant-47 mails an urgent letter whose content is a promissory note letter whose content is a money content (128) to inhabitant-11 for a cost of 4 euros
   - 4 euros are debited from inhabitant-47 account whose balance is now 4996 euros
**************************************************
Day 5
<- inhabitant-55 receives a simple letter whose content is a text content (bla bla) from inhabitant-39
<- inhabitant-56 receives an urgent letter whose content is a registered letter whose content is a promissory note letter whose content is a money content (714) from inhabitant-22
   - 714 euros are debited from inhabitant-22 account whose balance is now 4240 euros
   + inhabitant-56 account is credited with 714 euros; its balance is now 5714 euros
-> inhabitant-56 mails a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (714)) to inhabitant-56 for a cost of 1 eur
   - 1 euro is debited from inhabitant-56 account whose balance is now 5713 euros
-> inhabitant-56 mails an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a promissory note letter whose cor
   - 1 euro is debited from inhabitant-56 account whose balance is now 5712 euros
<- inhabitant-11 receives an urgent letter whose content is a promissory note letter whose content is a money content (128) from inhabitant-47
   - 128 euros are debited from inhabitant-47 account whose balance is now 4868 euros
   + inhabitant-11 account is credited with 128 euros; its balance is now 5128 euros
-> inhabitant-11 mails a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (128)) to inhabitant-11 for a cost of 1 eur
   - 1 euro is debited from inhabitant-11 account whose balance is now 5127 euros
-> inhabitant-44 mails a simple letter whose content is a text content (bla bla) to inhabitant-36 for a cost of 1 euro
   - 1 euro is debited from inhabitant-44 account whose balance is now 4999 euros
-> inhabitant-76 mails an urgent letter whose content is a registered letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-11 for a cost of 32 euros
   - 32 euros are debited from inhabitant-76 account whose balance is now 4968 euros
-> inhabitant-57 mails a promissory note letter whose content is a money content (296) to inhabitant-53 for a cost of 3 euros
   - 3 euros are debited from inhabitant-57 account whose balance is now 4997 euros
**************************************************
Day 6
<- inhabitant-56 receives a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (714)) from inhabitant-22
<- inhabitant-56 receives an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a promissory note letter whose
<- inhabitant-11 receives a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (128)) from inhabitant-47
<- inhabitant-36 receives a simple letter whose content is a text content (bla bla) from inhabitant-44
<- inhabitant-11 receives an urgent letter whose content is a registered letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-76
-> inhabitant-11 mails an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content is a
   - 1 euro is debited from inhabitant-11 account whose balance is now 5126 euros
<- inhabitant-53 receives a promissory note letter whose content is a money content (296) from inhabitant-57
   - 296 euros are debited from inhabitant-57 account whose balance is now 4701 euros
   + inhabitant-53 account is credited with 296 euros; its balance is now 5296 euros
-> inhabitant-53 mails a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (296)) to inhabitant-53 for a cost of 1 eur
   - 1 euro is debited from inhabitant-53 account whose balance is now 5295 euros
-> inhabitant-63 mails a registered letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-42 for a cost of 16 euros
   - 16 euros are debited from inhabitant-63 account whose balance is now 4984 euros
-> inhabitant-60 mails a registered letter whose content is a simple letter whose content is a text content (bla bla) to inhabitant-2 for a cost of 16 euros
   - 16 euros are debited from inhabitant-60 account whose balance is now 4984 euros
**************************************************
Day 7
<- inhabitant-11 receives an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content i
<- inhabitant-53 receives a thanks letter which is a simple letter whose content is a text content (thanks for a promissory note letter whose content is a money content (296)) from inhabitant-57
<- inhabitant-42 receives a registered letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-63
-> inhabitant-42 mails an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content is a
   - 1 euro is debited from inhabitant-42 account whose balance is now 4999 euros
<- inhabitant-2 receives a registered letter whose content is a simple letter whose content is a text content (bla bla) from inhabitant-60
-> inhabitant-2 mails an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content is a
   - 1 euro is debited from inhabitant-2 account whose balance is now 4999 euros
**************************************************
Day 8
<- inhabitant-42 receives an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content i
<- inhabitant-2 receives an aknowledgment of receipt which is a simple letter whose content is a text content (aknowledgment of receipt for a registered letter whose content is a simple letter whose content is
[...]
```

TD 8 — Louables et contraintes

Au terme de ce TD, vous devriez être capable
- d'utiliser le patron de conception *strategy* ;
- d'utiliser le patron de conception *composite* ;
- d'utiliser la réflexion pour garder à l'exécution des informations sur les paramètres de types génériques.

À préparer avant le TD

Q 1 . Lire la fiche sur le patron de conception *decorator* et indiquer comment ce patron s'applique dans le TD précédent.

À faire pendant le TD

On s'intéresse à des biens (*good*) qu'il est possible de louer (*rent*). Pour pouvoir louer, le client (*customer*) doit satisfaire la contrainte de location. Un prix de location à la journée (*daily price*), un entier fixé à la construction, est associé à chaque bien.
Parmi les biens louables (*rentable*) on trouve :
- les *vidéos* (*video*) qui peuvent être louées sans contrainte sauf quand elles sont réservées aux adultes ;
- les *appartements* (*apartment*) qui peuvent être loués avec justification de revenus (*income*) ;
- les *salles des fêtes* (*community hall*) qui peuvent être louées avec des arrhes (*deposit*) ;
- les *véhicules* qui peuvent être loués avec le permis (*driving license*).

Les contraintes de location qui s'appliquent aux biens louables peuvent être de natures différentes :
- *aucune contrainte* : tout le monde peut louer ;
- *contrainte d'âge* : il faut avoir un âge minimum fixé par la contrainte pour louer le bien ;
- *contrainte de permis* : il faut avoir le permis de conduire pour louer le bien ;
- *contrainte d'arrhes* : il faut que le client accepte de laisser un certain montant au moment de la réservation ;
- *contrainte de revenus* : il faut que le client justifie d'un niveau de revenus minimum ;
- etc.

Q 2 . Faire un diagramme de classes UML décrivant une modélisation de ce domaine.

Q 3 . Implémentez les classes nécessaires aux vidéos et véhicules ainsi qu'à leurs contraintes.

Certaines locations nécessitent que plusieurs contraintes soient validées : par exemple, louer un véhicule utilitaire nécessite d'avoir le permis *et* un dépôt de garantie.

Q 4 . Adaptez votre architecture et donnez le code correspondant à cet ajout.

On appelle loueur (*renter*) une société qui a pour objet la location de biens. On trouve des loueurs spécialisés (*specialized renter*) tels que les vidéo clubs (*video club*) et les agences immobilières (*estate agency*).

Q 5 . Faites une proposition d'architecture : fournissez un diagramme de classes et donnez la signature d'une méthode `rent(Customer)` qui retourne un bien pour la personne en paramètre quand c'est possible, `null` sinon. Votre proposition doit faire passer le test suivant :

```
@Test
public void createVideoClub() {
  SpecializedRenter<Video> videoClub = new SpecializedRenter<Video>(...);
  Video video = videoClub.rent(new Customer());
  assertNotNull(video);
}
```

TD 9 — Plugins

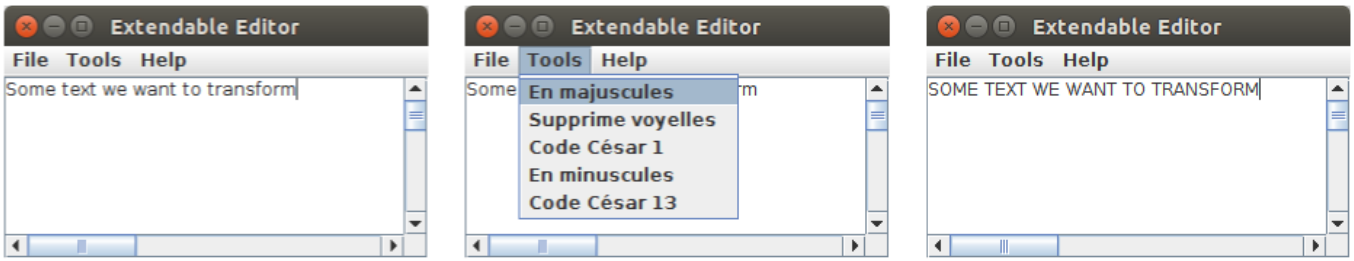
Au terme de ce TD, vous devriez être capable
- d'utiliser le patron de conception *observer* ;
- de concevoir et d'implémenter des programmes simples réagissant à des évènements (ici, fonctions du temps et du système de fichiers) : on parle de programmation *réactive* ;
- de concevoir et d'implémenter des programmes dont le comportement est étendu à l'exécution.

À préparer avant le TD

Q 1 . Lire la fiche sur le patron de conception *strategy* et indiquer comment ce patron s'applique dans le TD précédent.

Q 2 . Lire la fiche sur le patron de conception *composite* et indiquer comment ce patron s'applique dans le TD précédent.

L'objectif de ce TD est la mise en place d'une application qui s'adapte dynamiquement en fonction de *plugins* installés dans un répertoire. L'application se compose d'une zone de texte et d'une barre de menu. Le menu `Tools` propose un certain nombre d'opérations de manipulation du texte.



Cette application a la particularité de pouvoir être enrichie dynamiquement — c'est-à-dire pendant son exécution — par de nouvelles opérations de manipulation de texte. Chaque opération est un *plugin*. L'installation de plugin se fait par ajout d'un fichier `.class` dans un dossier particulier, dossier qui est parcouru toutes les secondes par l'application à la recherche de nouveaux plugins. C'est de cette façon que fonctionne le dossier `dropins` d'Eclipse.

Q 3 . Lisez la page décrivant le dossier `dropins` sur `http://help.eclipse.org`.

Q 4 . Lisez la documentation de `java.io.FilenameFilter` et de `java.io.File.list(FilenameFilter)`.

Q 5 . Écrivez un programme qui crée puis démarre un timer (`javax.swing.Timer`) dont la fonction est d'afficher sur la sortie standard l'heure (`java.util.Calendar.getInstance().getTime()`) toutes les secondes. N'oubliez pas de démarrer le timer. Pour éviter que l'application ne se termine aussitôt, placez un superbe `while (true) ;` à la fin de votre `main()`.

À faire pendant le TD

Q 6 . Écrivez une classe `PluginFinder` dont les instances sont paramétrées par un répertoire donné à la création et qui dispose d'une méthode qui retourne tous les fichiers de ce répertoire dont l'extension est `.class`. Cette classe utilisera la classe `PluginFilter` (implémentant `FilenameFilter`) qui se chargera de faire la sélection des fichiers.

Nous allons maintenant nous intéresser à la mise en place d'un mécanisme événementiel pour gérer la détection de l'apparition de fichiers dans un répertoire choisi. Il s'agira d'émettre un évènement à chaque fois qu'un *nouveau* fichier (d'un format bien particulier, voir plus bas) est ajouté dans le répertoire spécifié. L'objet émetteur de l'évènement, une instance de `PluginFinder`, examinera régulièrement le contenu du répertoire concerné à l'aide d'un *timer*.
L'instance de la classe gérant le menu `Tools` devra être inscrite aux évènements envoyés par le `PluginFinder`.

Q 7 . Dessinez un diagramme de classes UML représentant cette architecture.

Q 8 . Finissez de coder la classe `PluginFinder`.

Q 9 . Pour ne pas avoir à coder un menu en TD, codez une classe `PluginAddedLogger` et dont la seule réaction à l'événement d'ajout de fichier est d'afficher un message.

Ajoutons les contraintes qui font d'un fichier avec l'extension `.class` un vrai plugin. La classe sous-jacente à ce fichier devra :
- implémenter l'interface `plugins.Plugin` (éventuellement indirectement grâce à `Class.isAssignableFrom()`, voir appendice) :

```
public interface Plugin {
  public String transform(String s) ;
  public String getLabel() ;
}
```

— appartenir au paquetage `plugins` ;
— fournir un constructeur sans paramètre.

Q 10 . Écrivez les tests pour `PluginFilter`. Chaque contrainte ci-dessus doit être vérifiée indépendamment.

Q 11 . Adaptez le code de `PluginFilter` à ces nouvelles contraintes.

Appendice

Exemple d'application

Un exemple d'application à réaliser est disponible [1] : décompressez le dans votre espace de travail. Vous devez maintenant avoir un répertoire `plugins/` contenant les répertoires `classes/` et `dropins/`. Placez vous dans le répertoire `plugins/` et exécutez la commande :

```
java -classpath classes:dropins plugins.editor.Main &
```

La fenêtre qui apparaît contient une barre de menu. Le menu `Tools` ne propose rien pour l'instant.
Vous pouvez enrichir cette application en lui ajoutant dynamiquement de nouveaux *plugins*. Les plugins proposés se trouvent dans le dossier `classes/plugins/`. Copiez un par un les fichiers `ToLowercase.class` et `ToUppercase.class` situés dans ce répertoire vers le répertoire `dropins/`. Observez ce qui se passe entre chaque copie dans le menu `Tools` et testez chacun des plugins proposés.
Vous pouvez placer dans le dossier `dropings/` un autre fichier (y compris d'extension `.class`) quelconque et rien ne se passe au niveau de l'application.
Il est également possible de supprimer un plugin : il vous suffit pour cela de retirer le fichier `.class` correspondant du répertoire `dropins/`.

Documentation

```
public Timer.Timer(int delay, ActionListener listener)

  Creates a Timer and initializes both the initial delay and
  between-event delay to delay milliseconds. If delay is less than or
  equal to zero, the timer fires as soon as it is started. If listener
  is not null, it's registered as an action listener on the timer.

  Parameters:
    delay - milliseconds for the initial and between-event delay
    listener - an initial listener; can be null

public void Timer.start()

  Starts the Timer, causing it to start sending action events to its
  listeners.

public File[] File.listFiles(FilenameFilter filter)

  Returns an array of abstract pathnames denoting the files and
  directories in the directory denoted by this abstract pathname that
  satisfy the specified filter. The behavior of this method is the same
  as that of the listFiles() method, except that the pathnames in the
  returned array must satisfy the filter. If the given filter is null
  then all pathnames are accepted. Otherwise, a pathname satisfies the
  filter if and only if the value true results when the
  FilenameFilter.accept(File, String) method of the filter is invoked on
  this abstract pathname and the name of a file or directory in the
  directory that it denotes.

  Parameters:
    filter - A filename filter
  Returns:
    An array of abstract pathnames denoting the files and directories in
    the directory denoted by this abstract pathname. The array will be
    empty if the directory is empty. Returns null if this abstract
    pathname does not denote a directory, or if an I/O error occurs.
  Throws:
    SecurityException - If a security manager exists and its
    SecurityManager.checkRead(String) method denies read access to the
    directory
```

1. http://www.fil.univ-lille1.fr/~cassou/coo/documents/application-plugins.zip

```
  Since:
    1.2

public boolean Class.isAssignableFrom(Class<?> cls)

  Determines if the class or interface represented by this Class object
  is either the same as, or is a superclass or superinterface of, the
  class or interface represented by the specified Class parameter. It
  returns true if so; otherwise it returns false. If this Class object
  represents a primitive type, this method returns true if the specified
  Class parameter is exactly this Class object; otherwise it returns
  false.

  Specifically, this method tests whether the type represented by the
  specified Class parameter can be converted to the type represented by
  this Class object via an identity conversion or via a widening
  reference conversion. See The Java Language Specification, sections
  5.1.1 and 5.1.4 , for details.

  Parameters:
      cls - the Class object to be checked
  Returns:
      the boolean value indicating whether objects of the type cls can be
      assigned to objects of this class
  Throws:
      NullPointerException - if the specified Class parameter is null.
  Since:
      JDK1.1

public static Class<?> Class.forName(String className) throws ClassNotFoundException

  Returns the Class object associated with the class or interface with
  the given string name. Invoking this method is equivalent to:

      Class.forName(className, true, currentLoader)

  where currentLoader denotes the defining class loader of the current
  class.

  For example, the following code fragment returns the runtime Class
  descriptor for the class named java.lang.Thread:

      Class t = Class.forName("java.lang.Thread")

  A call to forName("X") causes the class named X to be initialized.

  Parameters:
    className - the fully qualified name of the desired class.
  Returns:
    the Class object for the class with the specified name.
  Throws:
    LinkageError - if the linkage fails
    ExceptionInInitializerError - if the initialization provoked by
      this method fails
    ClassNotFoundException - if the class cannot be located

public T Class.newInstance() throws InstantiationException, IllegalAccessException

  Creates a new instance of the class represented by this Class object.
  The class is instantiated as if by a new expression with an empty
  argument list. The class is initialized if it has not already been
  initialized.

  Note that this method propagates any exception thrown by the nullary
  constructor, including a checked exception. Use of this method
  effectively bypasses the compile-time exception checking that would
  otherwise be performed by the compiler. The Constructor.newInstance
  method avoids this problem by wrapping any exception thrown by the
  constructor in a (checked) InvocationTargetException.

  Returns:
    a newly allocated instance of the class represented by this object.
  Throws:
    IllegalAccessException - if the class or its nullary constructor is not accessible.
    InstantiationException - if this Class represents an abstract
      class, an interface, an array class, a primitive type, or void; or if
      the class has no nullary constructor; or if the instantiation fails
      for some other reason.
    ExceptionInInitializerError - if the initialization provoked by this method fails.
    SecurityException - If a security manager, s, is present and any
```

```
of the following conditions is met:

- invocation of s.checkMemberAccess(this, Member.PUBLIC) denies
    creation of new instances of this class
- the caller's class loader is not the same as or an ancestor of
    the class loader for the current class and invocation of
    s.checkPackageAccess() denies access to the package of this class
```

TD 10 — Questionnaire

Au terme de ce TD, vous devriez être capable
- d'utiliser les mécanismes de base de l'API de réflexion Java ;
- d'écrire des tests pour du code d'interface utilisateur.

À préparer avant le TD

Q 1 . Lire la fiche sur le patron de conception *observer* et indiquer comment ce patron s'applique dans le TD précédent.

Q 2 . Lire au moins les sections suivantes sur la réflexion Java : `http://docs.oracle.com/javase/tutorial/reflect` :
- *Trail : The Reflection API*
- *Classes, Retrieving Class Objects*
- *Classes, Troubleshooting*
- *Members, Invoking Methods*
- *Members, Creating New Class Instances*

Un questionnaire est défini par un ensemble de questions. Chaque question est caractérisée par un texte (la question elle-même), la solution et un nombre de points. Les solutions aux questions peuvent être de différentes natures : numériques (*numerical*), textuelles (*textual*) ou de type oui/non (*boolean*). On représente les questionnaires par des instances d'une classe `Questionnaire` qui dispose d'une méthode `runAll()` qui consiste à (en commençant par la première question) :

1. afficher le texte de la question et la nature de la solution (le *prompt*) ;
2. attendre la réponse de l'utilisateur en n'acceptant la saisie que lorsqu'elle est *conforme à la nature de la solution* (par exemple un nombre si la solution est numérique),
3. on a alors deux situations : si la réponse donnée est correcte, l'indiquer et augmenter le score de l'utilisateur ; si la réponse n'est pas correcte, afficher la solution.
4. passer à la question suivante s'il en reste, sinon afficher le score.

Voici un exemple de trace possible pour cette méthode.

```
Quel est le nom de l'auteur du Seigneur des Anneaux ?
(textual) Tolkien
correct (1 point)
Frodon est un Hobbit.
(yes/no) yes
correct (1 point)
Combien de membres composent la Compagnie de l'Anneau ?
(numerical) neuf
(numerical) 9
correct (2 points)
Gandalf est un humain.
(yes/no) oui
(yes/no) yes
wrong, good answer was no
En quelle annee est paru le Seigneur des Anneaux ?
(numerical) 1960
wrong, good answer was 1954

You have 4 points.
```

Q 3 . Que teste le code suivant ?

```
public class QuestionnaireTest {
  [...]
  @Test
  public void whatAmITesting() {
    int nbPoints = 15;

    question.isSolution = true;
    question.nbPoints = nbPoints;

    assertEquals(0, questionnaire.getScore());
    questionnaire.runQuestion(question);
    assertEquals(nbPoints, questionnaire.getScore());
  }
}
```

Q 4 . Écrivez un test unitaire pour vérifier la contrainte *"attendre la réponse de l'utilisateur en n'acceptant la saisie que lorsqu'elle est* conforme à la nature de la solution*"*.

À faire pendant le TD

On choisit d'adopter le point de vue que ce qui change d'une question à une autre c'est le type de réponse. On a donc un seul type pour les questions, la classe `Question`, mais plusieurs types de réponses (e.g., `NumericalAnswer`, `BooleanAnswer`, `TextualAnswer`).

Q 5 . En respectant ce point de vue, donnez un diagramme de classe UML pour les types nécessaires à la gestion de tels questionnaires.

On souhaite pouvoir initialiser les questionnaires à partir d'informations contenues dans des fichiers texte.
La structure du fichier est définie par des suites de blocs de 3 lignes (voir exemple en annexe) : la première ligne contient le texte de la question, la seconde ligne contient la solution et la troisième le nombre de points associés à la question (un entier). On se propose de créer une classe `QuestionnaireFactory` qui permet la création d'un questionnaire à partir d'un fichier sous cette forme. Voici un extrait de cette classe :

```
public class QuestionnaireFactory {
  public Questionnaire readFile(BufferedReader reader) throws IOException {
    List<IQuestion> questions = new ArrayList<IQuestion>();

    String line;
    while ((line = reader.readLine()) != null) {
      String questionLine = line;
      String solutionLine = reader.readLine();
      String pointLine = reader.readLine();

      if (solutionLine == null || pointLine == null) {
        throw new IOException("Invalid input file");
      }

      questions.add(readBlock(questionLine, solutionLine, pointLine));
    }

    return new Questionnaire(questions);
  }
  public IQuestion readBlock(String textLine, String solutionLine,
      String pointLine) throws IOException {
    Integer points;

    try {
      points = new Integer(pointLine);
    } catch (NumberFormatException e) {
      throw new IOException(e);
    }

    Question question = new Question(textLine,
        answerFactory.buildAnswer(solutionLine), points);
    return question;
  }

}
```

Pour pouvoir créer une question à partir d'un fichier, il est nécessaire de pouvoir créer les objets de type réponse associés à chaque question. On décide de déléguer ce travail à une classe `AnswerFactory` qui disposera d'une méthode de fabrique `buildAnswer(String)` pour créer les différents objets réponses à partir de la 2ème ligne de chaque bloc du fichier.

Q 6 . Proposez un code pour la méthode `AnswerFactory.buildAnswer()`.

On ajoute maintenant un nouveau type de réponse qui accepte plusieurs entrées de l'utilisateur. Les points sont attribués si l'utilisateur fournit l'une de ces entrées. Le nombre d'entrées possibles est annoncé. Voici un exemple :

```
Donnez le nom de l'un des hobbits de la Compagnie de l'Anneau.
(4 possible solutions) Pippin
correct (1 point)
```

Q 7 . Sans coder ce nouveau type de réponse, indiquez quels sont les changements nécessaires dans votre code. Votre architecture respecte elle le principe ouvert fermé ? Si ça n'est pas le cas, proposez une solution pour que le principe soit le plus respecté possible.

Q 8 . Pour respecter le principe ouvert fermé complètement, il est nécessaire d'adapter la structure du fichier texte de définition de questionnaires. Proposer et implémenter une telle adaptation et indiquer en quoi votre architecture respecte complètement le principe ouvert fermé.

Appendice

Exemple de fichier questionnaire

Les questions sont par blocs de 3 lignes, la première contient le texte de la question, la seconde la solution et la troisième le nombre de points associés à la question (un entier). Le dernier bloc correspond aux questions à la fin du TD.

```
Quel est le nom de l'auteur du Seigneur des Anneaux ?
Tolkien
1
Frodon est un Hobbit.
vrai
1
Combien de membres composent la Compagnie de l'Anneau ?
9
2
Gandalf est un humain.
faux
3
En quelle annee est paru le Seigneur des Anneaux ?
1954
3
Donnez le nom de l'un des hobbits de la Compagnie de l'Anneau.
Frodon ; Pippin ; Merry ; Sam
1
```

Documentation

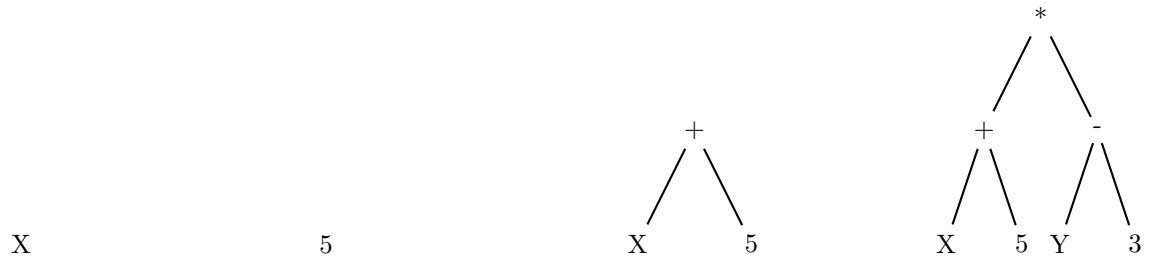
```
public Integer.Integer(String s) throws NumberFormatException

  Constructs a newly allocated Integer object that represents the int
  value indicated by the String parameter. The string is converted to an
  int value in exactly the manner used by the parseInt method for radix
  10.

  Parameters:
    s - the String to be converted to an Integer.
  Throws:
    NumberFormatException - if the String does not contain a parsable
    integer.
```

TD 11 — Expressions

Au terme de ce TD, vous devriez être capable
- d'appréhender la conception et l'implémentation de langages de programmation ;
- de simuler l'utilisation de fonctions de première classe avec des interfaces ;
- d'utiliser le patron de conception *visitor*.

À préparer avant le TD

On s'intéresse à la modélisation d'expressions numériques symboliques. Une *expression* est soit *atomique* soit *binaire*. Dans les expressions atomiques on distingue les *littéraux* (valeurs constantes, *literal value*) et les *variables* (noms donnés à des zones mémoires, *variable*). Les expressions binaires sont les opérateurs arithmétiques standards : `+`, `-`, `*` et `/`. Voici 4 exemples d'expressions représentées sous forme d'arbre. De gauche à droite on trouve une expression variable, une expression littérale et deux expressions binaires :



Dans ce problème nous nous limiterons aux littéraux à valeur entière. Les variables sont caractérisées par un nom. Les variables peuvent se voir attribuer une valeur entière. Une table, nommée *environnement* (*environment*), fait le lien entre une variable et sa valeur : un environnement doit avoir au moins les méthodes suivantes :
- une méthode pour associer une valeur à une variable (`setValue(Variable, Value)`) ;
- une méthode pour connaître la valeur d'une variable (`getValue(Variable)`) (lève l'exception `UnboundVariable` si la variable en paramètre n'a pas de valeur associée dans cet environnement) ;
- une méthode permettant de retrouver une variable à partir de son nom (`getVariable(String)`).

Q 1 . Indiquer comment implémenter les environnements, sans en donner le code ni les tests.

À faire pendant le TD

Les opérations qu'il est possible d'invoquer sur une expression sont les suivantes :
- `print()` : retourne une chaîne de caractères décrivant cette expression en utilisant une notation infixe totalement parenthésée. Ainsi pour les expressions ci-dessus on obtient respectivement `"X"`, `"5"`, `"(X+5)"` et `"((X+5)*(Y-3))"`.
- `eval(Environment)` : évalue cette expression en fonction de l'environnement. Le résultat est une valeur.

Q 2 . Proposer, un diagramme de classes UML représentant les types permettant la représentation des expressions.

Q 3 . Donner des tests pour `print()` et `eval()`.

Q 4 . Donner le code de la classe permettant la représentation d'expressions binaires pour les sommes.

La solution mise en place actuellement ne permet qu'un type d'affichage des expressions : l'affichage infixe totalement parenthésé. Il existe cependant d'autres types d'affichage pour les expressions : postfixe, préfixe, etc.

Q 5 . Faites une proposition de conception permettant de proposer plusieurs types d'affichage pour les expressions.

Q 6 . En imaginant qu'il puisse y avoir plusieurs façons d'évaluer une expression (par exemple en considérant qu'une variable prends la valeur `0` par défaut), réutilisez la conception précédente pour évaluer les expressions. Implémentez l'évaluation des expressions d'addition et de multiplication. Comment éviter la duplication de code ?

Q 7 . Que se passe-t-il lorsqu'on ajoute de nouveaux types d'expressions, par exemple `if` ?

Tests automatiques avec JUnit (rapidement)

On appelle *test automatique* un code qui vérifie qu'une partie d'un programme fonctionne comme attendu, et ceci sans intervention du développeur. C'est ce que vous réaliserez en écrivant des tests pour chaque méthode.
Ce document présente quelques points de détails techniques concernant Java et un framework de tests automatiques en particulier. Ce document vous aidera à réaliser vos projets. Cependant, les aspects plus généraux (et plus importants) que vous devez retenir seront discutés en cours et en TD.
Dans ce document, nous utiliserons le framework de tests JUnit [1] qui fait référence pour l'écriture et l'exécution des tests automatiques en Java. Plus particulièrement nous utiliserons JUnit4.

1 Classes de tests

Le principe est de placer les tests dans une classe dédiée.
Il faut ajouter en entête de cette classe les lignes d'import nécessaires, liées à JUnit :

```
import org.junit.*;
import static org.junit.Assert.*;
```

Et vous devrez certainement ajouter les `import` vers les classes de votre application utilisées dans les tests.
Précisons, si nécessaire, que les classes de test, comme leur nom l'indique, sont des classes Java. Elles peuvent donc définir des attributs, des méthodes (autres que les méthodes de tests), etc.

1.1 Paquetage et dossier source

Pour faciliter l'écriture de tests pour votre programme, il est recommandé de mettre les tests de la classe `A` dans une classe `ATest` du même paquetage. Ceci vous permettra de pouvoir accéder aux attributs et méthodes protégé(e)s (`protected`) de la classe `A` depuis la classe `ATest`. Par exemple, si `A` est dans le paquetage `dungeon.rooms`, la classe `ATest` devra se trouver dans `dungeon.rooms` aussi.
Cependant, certains déconseillent de mélanger dans un même dossier les tests et le programme testé. Une bonne séparation facilite la lecture du code et le déploiement (puisqu'on déploie en général le programme sans ses tests). Pour ce faire, vous devrez utiliser la notion de *dossier source* (*source folder*). Un dossier source est un dossier dans lequel les fichiers `.java` seront cherché par le compilateur `javac`. Il est tout à fait possible d'avoir plusieurs dossiers sources pour un même projet. Typiquement, un dossier `src/` et un dossier `test/`. Pour revenir à l'exemple précédent, vous devriez avoir :

- un fichier `src/dungeon/rooms/A.java` et
- un fichier `test/dungeon/rooms/ATest.java`.

2 Méthodes de tests

Dans une classe de test on crée autant de méthodes que de tests que l'on souhaite écrire. Il y aura souvent plusieurs méthodes de tests pour une même méthode d'une classe testée. Même si cela amène à écrire un peu plus de code, car plusieurs méthodes de tests, il est conseillé d'écrire une méthode par aspect testé (principe "unitaire" des tests). Par exemple, si vous avez une méthode contenant un `if`, vous pouvez écrire deux méthodes 2 tests, une par valeur possible de la condition du `if` : dans ce cas, un test se chargera de vérifier le cas nominal (le premier bloc du `if`), l'autre test se chargera de vérifier le cas alternatif (le bloc `else`).
Ces méthodes doivent obligatoirement :

- être précédées de l'annotation `@Test`,
- être publiques (`public`),
- ne rien retourner (`void`) et
- ne rien prendre en paramètre.

Le nom de ces méthodes est libre, mais il est recommandé d'y inclure le scénario testé et la réponse attendue. [2]
Par exemple :

```
@Test
public void shouldLockOutUserAfterThreeInvalidLoginAttempts() {
  [...]
}
```

Ces méthodes contiennent le code exécuté pour le test, en particulier les *assertions de test* qui doivent être vérifiées pour que le test passe :

- `assertTrue(v)` vérifie que la valeur fournie en paramètre vaut `true`,
- `assertFalse(v)` vérifie que la valeur fournie en paramètre vaut `false`,
- `assertEquals(expected, actual)` vérifie l'égalité de deux valeurs passées en paramètre, en utilisant `equals()`,

1. http://junit.org/
2. http://googletesting.blogspot.fr/2014/10/testing-on-toilet-writing-descriptive.html

— `assertNull(ref)` (respectivement `assertNotNull(ref)`) vérifie que la référence fournie en paramètre est (resp. n'est pas) `null`
— `assertSame(expected, actual)` vérifie en utilisant `==` que les deux références fournies en paramètre correspondent au même objet (`assertNotSame(expected, actual)` existe également)
— `fail()` échoue toujours

Pour `assertEquals()` et `assertSame()`, il faut faire attention à distinguer les 2 paramètres (`expected` et `actual`). L'ordre est important car les messages d'erreurs et les outils de comparaison en tiennent compte. Le premier paramètre est la valeur que l'on veut obtenir, le second est celle que l'on teste.

> **Le travail le plus compliqué (et c'est une tâche réellement difficile) est de bien construire les tests réalisés afin qu'ils permettent d'avoir le *maximum de confiance* sur le fonctionnement de la méthode.**

3 Tester les exceptions

Pour tester la levée d'une exception on peut utiliser une annotation particulière. La méthode de tests doit être précédée par :

```
@Test(expected=MyException.class)
public void mytest() {
  [...]
}
```

Le test sera réussi si et seulement si une exception du type mentionné est levée lors de l'exécution de la méthode. Dans le cas contraire il échoue. Il est conseillé de créer une méthode de test particulière pour les exceptions.
L'exemple ci-dessus est équivalent au code suivant mais on préférera utiliser le paramètre `expected` qui est plus concis :

```
@Test
public void mytest() {
  try {
    [...]
  } catch(MyException e) {
    // Everything is ok, that was expected
    return;
  }
  fail("MyException should have be thrown but was not");
}
```

4 Méthodes complémentaires

L'initialisation des objets nécessaires aux tests se répète le plus souvent d'un test à l'autre. Afin de faciliter l'écriture des méthodes de tests il est possible de définir et désigner des méthodes qui seront exécutées avant *chacune* des méthodes de tests. Il s'agit à nouveau de méthodes dont la signature sera `public void` *`nomMethode`*`()`, mais cette fois elles sont précédées de l'annotation `@Before`. Il peut y avoir plusieurs méthodes ainsi annotées, toutes sont exécutées avant chacune des méthodes de tests.

Junit fournit énormément de fonctions non discutées ici. Référez vous au site internet pour de plus amples informations.

5 Compilation

Une classe de tests se compile comme toute classe Java. Simplement il faut indiquer dans le `classpath` le chemin vers le jar de JUnit (appelé `junit-4.x.jar` où `x` désigne le numéro de version) ainsi que les autres dépendances éventuelles du code testé.

Introduction à Eclipse (rapidement)

Le logiciel ECLIPSE est un environnement de développement intégré (IDE). Libre, il peut être téléchargé sur le site `www.eclipse.org`.
Il n'est pas question dans ce document de présenter exhaustivement les fonctionnalités de l'outil ECLIPSE. Seules les principales fonctionnalités seront présentées ici, charge à vous de découvrir les autres possibilités offertes afin d'accroître votre efficacité. L'intérêt d'un outil comme Eclipse, quand il est bien utilisé, est de permettre au programmeur de se concentrer sur l'essentiel de son travail et de se dégager de détails techniques peu importants.

1 Premier lancement

Au premier lancement d'ECLIPSE vous aurez une fenêtre "Workspace Launcher" qui apparaît, elle propose un emplacement par défaut pour le *Workspace*. Le *Workspace* est un répertoire où ECLIPSE stocke un certain nombre d'informations et de fichiers qui lui sont utiles.
Vous pouvez ou non modifier l'emplacement proposé puis cochez la case située en dessous et intitulée Use this as default and do not ask again. Vous pouvez ensuite valider par OK. Cette fenêtre ne devrait plus apparaître lors des lancements ultérieurs d'Eclipse.
Toujours uniquement pour le premier lancement, si vous arrivez sur une fenêtre "Welcome". Fermez cette fenêtre. Vous arrivez alors sur l'espace de travail. Si elles apparaissent, fermez la fenêtre "Tasks list" (en bas à droite) et "Font and Colors" (sur la droite) qui ne nous seront pas utiles.

2 Créer un projet

ECLIPSE fonctionne par projet. Pour créer un projet faites File→New→Java Project..., ou dans la fenêtre Package Explorer (sur la gauche) faites clic droit puis New, etc. Choisissez un nom de projet.
Dans la partie Project layout (dans la seconde moitié de la fenêtre) vous pouvez demander à distinguer les dossiers pour les fichiers sources (`.java`) et binaires (`.class`). C'est ce qu'il faut faire. Si ce n'est pas déjà fait, sélectionnez donc Create separate source and output folders, puis cliquez sur Configure Defaults... et dans les Folders nommez les dossiers (par exemple src et classes). Cliquez Ok.
De retour dans la fenêtre "New Java Project", en décochant Use default location ("en haut" dans la fenêtre) vous avez la possibilité de ne pas placer les fichiers du projet dans un dossier workspace géré par ECLIPSE mais de préciser le répertoire de travail. Prenez cette seconde option et indiquez un répertoire dans votre espace de travail où seront rangés les fichiers de ce projet (vous pouvez créer un nouveau dossier via ECLIPSE via Browse...).
Faites Next, vous pouvez alors vérifier dans l'onglet Source que le dossier src a bien été ajouté. [1] Vous pouvez aussi ajouter un dossier `test` à vos sources pour y placer vos classes de tests.
Dans l'onglet Libraries vous devriez voir apparaître la référence à la bibliothèque du jdk utilisée (ici la 1.7 priori). [2]
Ces données peuvent être modifiées par la suite en accédant aux Properties d'un projet (clic droit sur le nom du projet).
Enfin cliquez Finish. Si vous avez un message vous proposant de passer en mode java perspective, acceptez.
Le projet est créé et vous le voyez apparaître, ainsi qu'un dossier src dans la zone de gauche de l'écran.

3 Créer un paquetage

Pour créer un paquetage, placez-vous sur l'icône du dossier src, cliquez droit puis New et Package.

4 Création et code

Lors du développement, vous remarquerez des symboles apparaissant dans la marge. Le plus souvent, ce symbole indiquera une erreur. Le code que vous saisissez est analysé au fur et à mesure et en cas d'erreur la source d'erreur (estimée) est soulignée de rouge dans le code. En plaçant le curseur au-dessus de ce signe le message d'erreur est affiché.
La petite ampoule jaune dans la marge mentionne qu'une proposition d'aide de correction est disponible, cliquez sur l'ampoule pour les afficher.

5 Exploration de code

Dans un fichier de classe, en se plaçant sur le nom de la classe ou un nom de méthode, un clic droit puis le choix Quick Type Hierarchy fait apparaître pour une classe la hiérarchie des classes (super et sous-classes) et pour une méthode les éventuelles surcharges qui lui sont associées. Il est alors possible d'accéder directement aux éléments proposés.

1. Sinon il faut l'ajouter en choisissant Create new source folder.
2. C'est à cet endroit que l'on définit le `CLASSPATH` du projet, en ajoutant éventuellement des références vers d'autres bibliothèques tels que des "jars externes" par exemple.

Faire un CTRL-clic sur un envoie de message ouvre la définition de la méthode correspondante. Il en est de même si vous opérez sur un nom de classe ou d'interface.
Après un clic droit sur un élément du code (classe, méthode, attribut, etc.), le menu qui s'ouvre offre différentes possibilités. Notamment le choix References→Workspace permet de connaître tous les endroits du code ou cet élément apparaît. Ces occurrences sont affichées dans une fenêtre à part de nom Search (en bas de la fenêtre de l'IDE généralement) et sont accessibles par un clic.

6 Javadoc

Placez votre curseur au niveau de la signature d'une méthode, cliquez droit puis Source, puis Generate Element Comment. Le "template" pour la Javadoc est automatiquement inséré. Testez également avec une méthode qui a des paramètres ou une valeur de retour.
Le menu contextuel (celui obtenu par clic droit) offre beaucoup d'autres fonctionnalités utiles. Elles sont à découvrir par vous-même.

7 JUnit

Lire aussi à ce sujet la fiche sur les tests.

7.1 Création de classes de tests

Pour générer une classe de tests, il suffit d'un clic droit sur le nom de la classe à tester puis de choisir New puis JUnit Test Case. Dans la fenêtre de dialogue qui apparaît vous pouvez définir différents éléments pour la classe de test, notamment son nom, mais aussi le dossier dans lequel vous la définissez.
Pour pouvoir placer les classes dans le dossier `test`, il vous faut préalablement avoir ajouter ce dossier aux sources de votre projet. Pour cela, cliquez droit sur votre projet, choisissez Properties (dernier choix en bas de la liste), sélectionnez la rubrique Java Build Path et dans l'onglet Source, ajoutez le dossier `test` grâce à Add Folder... (vous pouvez même créer le dossier à ce moment là).
Après avoir vérifié que les informations dans cette fenêtre vous conviennent, cliquez sur Next. Vous pouvez alors choisir les méthodes de la classe pour lesquelles vous souhaitez écrire des tests. Les squelettes des méthodes de tests seront alors automatiquement générés une fois que vous aurez cliqué sur Finish.
Lors de la création de la première classe de test du projet, l'ajout de la bibliothèque `junit4` au projet est proposé, il faut accepter cet ajout.
Vous pouvez maintenant modifier cette classe pour définir le contenu de vos méthodes de test.

7.2 Exécution des tests

Pour exécuter les tests d'une classe, il suffit de cliquer droit sur le nom de la classe de test (ou du fichier), de choisir Run As puis Junit Test. Un onglet JUnit apparaît alors à côté de l'onglet Package Explorer. Vous pouvez y observer le rapport de vos tests, notamment la barre verte ou rouge selon que vos tests passent ou non. Vous y trouvez également des boutons pour manipuler vos tests.

8 Exécutez le code

Avant d'exécuter une application, il lui faut une méthode `main()`. Pour en créer une, ajouter une classe Main et dans la fenêtre de création cochez la création de la méthode `main()`, ensuite complétez le code de cette méthode. Vous pouvez maintenant cliquer droit sur le nom de la classe ou du projet et choisir Run As puis Java Application Le programme est alors exécuté. La trace apparaît dans une fenêtre console dans la partie inférieure de la fenêtre de l'IDE.

9 Refactoring

Des outils vous aident à réorganiser votre projet : déplacer des classes, modifier des noms de méthodes, etc.
Par exemple, pour renommer une méthode, cliquez droit sur son nom puis Refactor et Rename.... Changez le nom de la méthode et validez. Le code qui appelait cette méthode a été mis à jour pour faire référence au nouveau nom.
Essayez les autres refactorings.

10 Debugger

ECLIPSE dispose d'un debugger offrant de nombreuses possibilités qui permettent d'avoir une vue précise de l'exécution d'un programme en n'importe quel point de celui-ci. Nous allons en aborder ici les principales fonctionnalités.
Vous pouvez exécuter une application en mode debugger

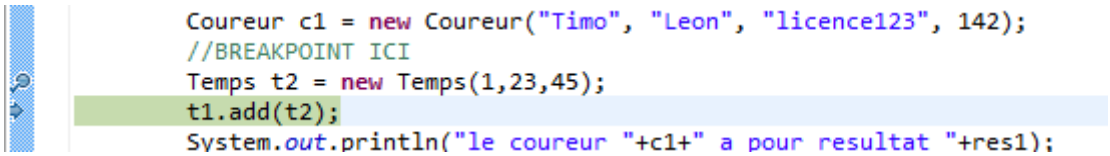


FIGURE 1 – Exécution pas-à-pas. La ligne courante est surlignée et désignée par une flèche dans la marge. On a fait un pas de traitement depuis le point d'arrêt positionné sur la ligne précédente comme on peut le constater dans la marge.

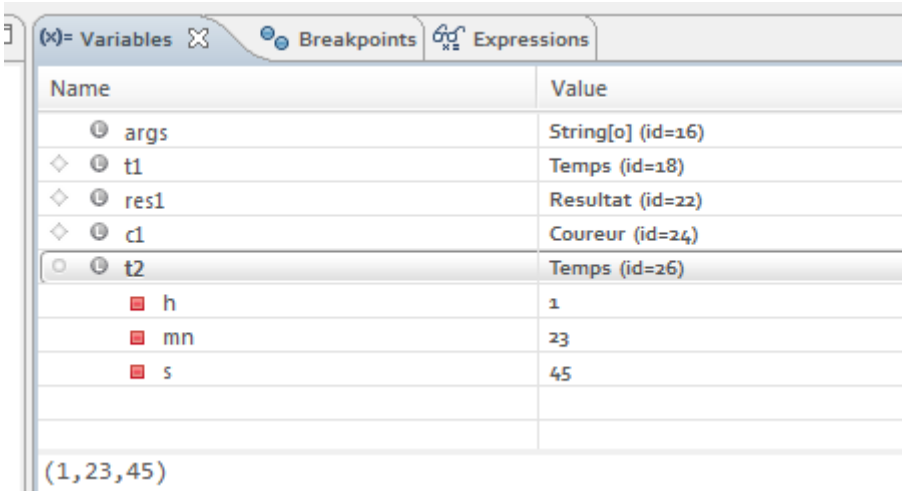


FIGURE 2 – La vue Variables qui affiche les variables définies dans le contexte du point actuel d'exécution.

— en choisissant Debug As→Java Application après un clic droit sur la classe dans la zone de paquetage
— en cliquant sur le bouton représentant une espèce de scarabée ("*bug*") dans la barre de boutons.

Une fenêtre doit vous proposer de passer dans la perspective "Debug". Il faut accepter pour avoir accès aux fonctionnalités du debugger. Vous pouvez également atteindre cette perspective via le menu Window→Open Perspective→Debug. [3] On peut retourner dans la perspective initiale par le même menu en choisissant la perpective Java.

10.1 Exécution et points d'arrêt

L'exécution en mode debugger permet d'inspecter l'état de la mémoire en cours d'exécution. Pour cela il est possible d'interrompre l'exécution du programme à un moment particulier en plaçant des *points d'arrêt* dans le code.
Pour placer un point d'arrêt dans un code, il suffit de cliquer dans la marge située sur la gauche de la zone d'édition (double clic ou clic droit puis Add Breakpoint) (voir Figure 1). Lors de l'exécution en mode debugger, le flux de traitement du programme s'interrompt dès qu'il atteint l'un de ces points en conservant l'état de la mémoire d'exécution.
Dans la perspective Debug le traitement s'arrête donc sur le premier point d'arrêt rencontré. Dans la zone supérieure droite d'ECLIPSE on trouve une zone Variables (également affichable via Window→Show View→Variables). On peut voir affiché dans cette zone (voir Figure 2) les différentes variables et attributs définis dans l'environnement d'exécution courant. Cette vue "Variables" permet d'avoir accès au détail de l'environnement d'exécution. Si une variable est de type objet, il est possible d'inspecter l'état de cet objet soit en cliquant sur le bouton à gauche du nom, soit en double-cliquant sur ce nom. Pour un objet, la colonne Value reprend le nom de la classe et l'information complémentaire (id=xxx) permet de distinguer les différentes instances de la classe : les id sont différents si et seulement si on a des objets distincts.

10.2 Contrôle de l'exécution

Une fois le traitement en pause, il est possible de maitriser la progression dans le flux d'exécution à l'aide des boutons placés en haut de la vue Debug. On peut ainsi passer d'un point d'arrêt au suivant (il est possible d'en ajouter ou supprimer un à tout moment) ou de faire une exécution pas-à-pas plus ou moins détaillée (cf. Figure 3).
Dans la zone Variables vous pouvez constater que certaines variables sont surlignées (en jaune a priori). Il s'agit de celles dont la valeur a changé depuis l'arrêt précédent.
Dans la zone Debug (voir Figure 4) vous pouvez visualiser la pile d'appels de méthodes qui a conduit au point d'exécution courant. En cliquant sur l'une de ces lignes, vous pouvez retrouver dans la zone Variables le contexte d'invocation de la méthode concernée. On peut ainsi analyser en détail les contextes qui nous ont amené jusqu'à la situation actuelle.

3. Des boutons situés à droite en haut permettent un accès direct aux différentes perspectives.

FIGURE 3 – Les boutons de contrôle de l'exécution en mode debugger et leur fonction.

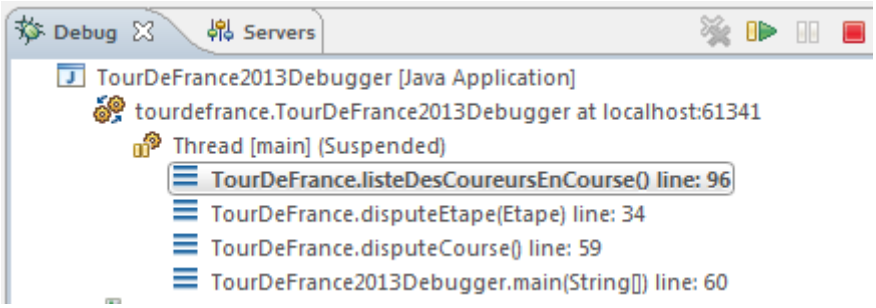


FIGURE 4 – La vue Debug et la pile des appels qui ont permis d'atteindre le point d'exécution courant.

10.3 Evaluation d'expressions

Le debugger offre également la possibilité d'évaluer n'importe quelle expression. Cela permet d'avoir une vue sur des valeurs autres que des variables. On utilise pour cela la vue Expressions, affichable aussi par Window→Show View→Expressions.
La vue Expressions est séparée verticalement en 2 parties. Sur la gauche les expressions et à droite leur valeur. Vous pouvez placer comme "expression" n'importe quelle portion de code valide.

10.4 Des breakpoints sous condition

Nous allons voir qu'il est possible d'avoir un contrôle plus fin de l'action des points d'arrêt.
Vous pouvez éditer un point d'arrêt (bouton droit sur la pastille bleue dans la marge ou dans la vue Breakpoints). Dans les propriétés du point d'arrêt, vous avez la possibilité de spécifier une condition (une expression Java) qui doit être vraie pour que le point d'arrêt soit pris en compte (voir Figure 5).

Apprenez à utiliser le debugger pour être efficace dans vos phases de mise au point et ne plus avoir à utiliser des `System.out.println()` pour visualiser l'état de votre programme.

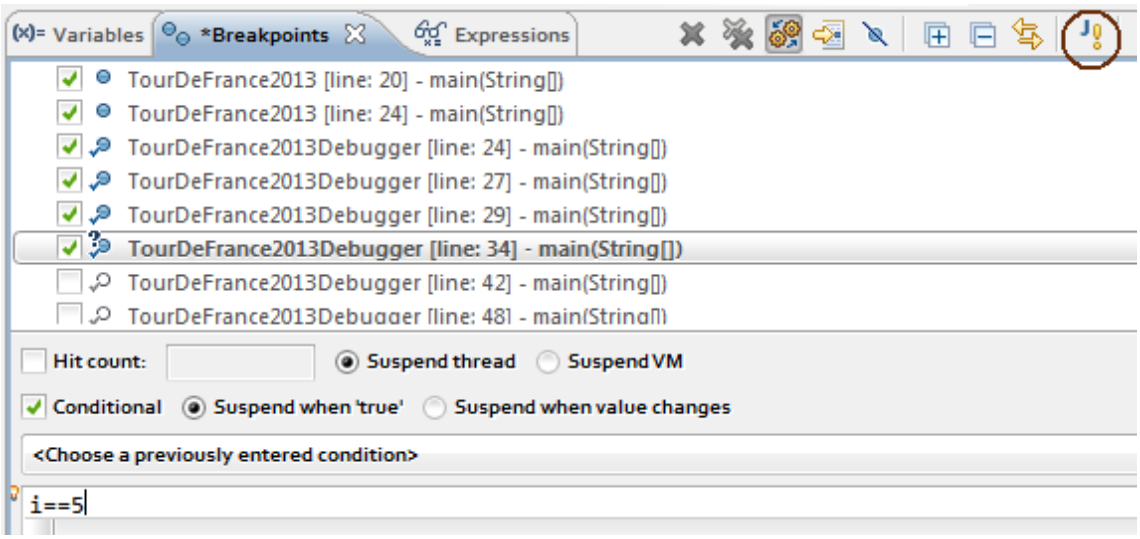


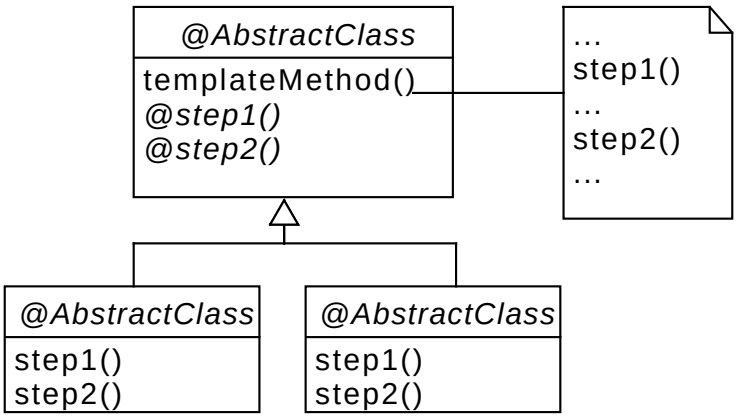
FIGURE 5 – Mise en place d'un breakpoint conditionnel et bouton de point d'arrêt sur exception.

Design Pattern : Template Method

Intent

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Structure



Éléments caractéristiques

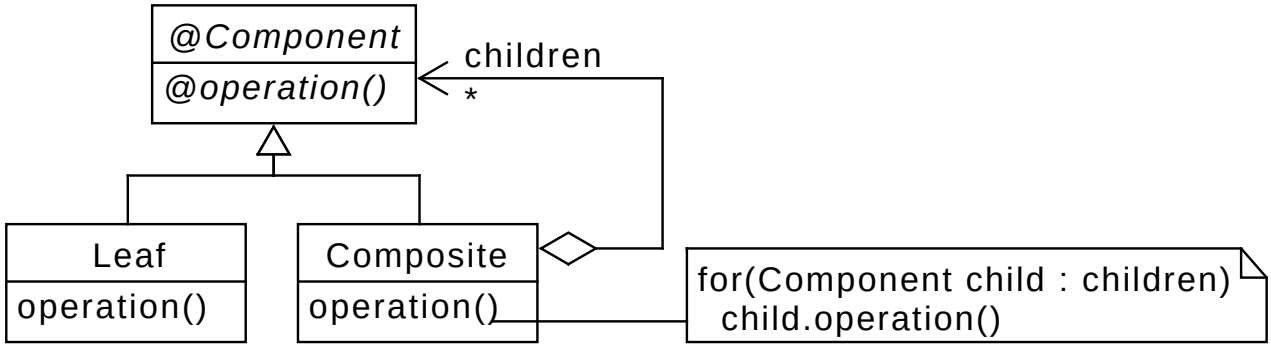
— une méthode décrivant un algorithme qui fait appel à des méthodes abstraites de la classe, ces méthodes sont définies dans les sous-classes, permettant à ces sous-classes de faire varier le comportement de l'algorithme.

Design Pattern : Composite

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure



Éléments caractéristiques

- il existe une ou plusieurs classes, sous-types de *Component*, qui ne sont pas des composites : *Leaf*.
- un *composite* est un sous-type de *Component* tel que :
  - il contient (a comme attribut) une collection d'objets du type *Component* : ce sont les "composants enfants". Ces éléments peuvent eux-mêmes être de type *Composite*. On peut ainsi construire des structures arborescentes de *Component*. Les feuilles de cette arbre sont de type *Leaf*.
  - les méthodes de `Composite` issues du type `Component` requièrent l'invocation de cette même méthode sur chaque composant enfant : c'est le cas de *operation()*.

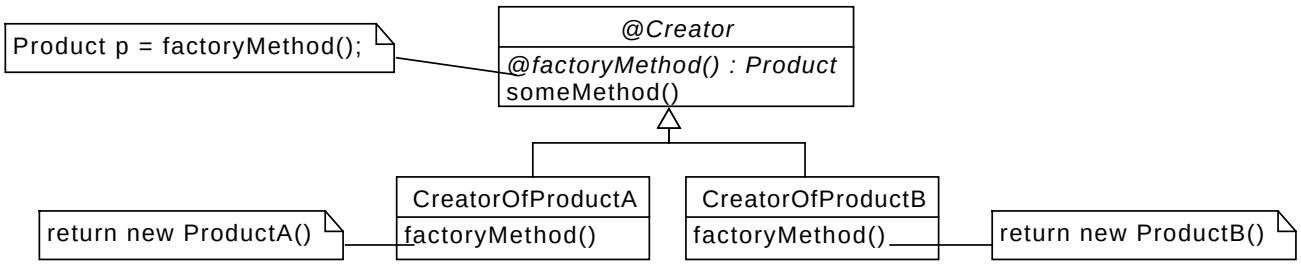
On peut considérer qu'au niveau de *operation()* du *Composite* on a, du point de vue du type *Component*, un appel récursif (propagation de l'invocation de *operation()* dans la structure arborescente récursive construite via les *Composite*). Le cas d'arrêt de cette récursivité est réalisé par l'invocation sur un objet de type *Leaf*.

Design Pattern : Factory Method

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Structure



Éléments caractéristiques

— une méthode dont la fonction est de créer un objet, cet objet est renvoyé comme résultat.
Cette méthode est surchargée dans les sous-classes pour modifier la classe de l'objet qu'elle crée.

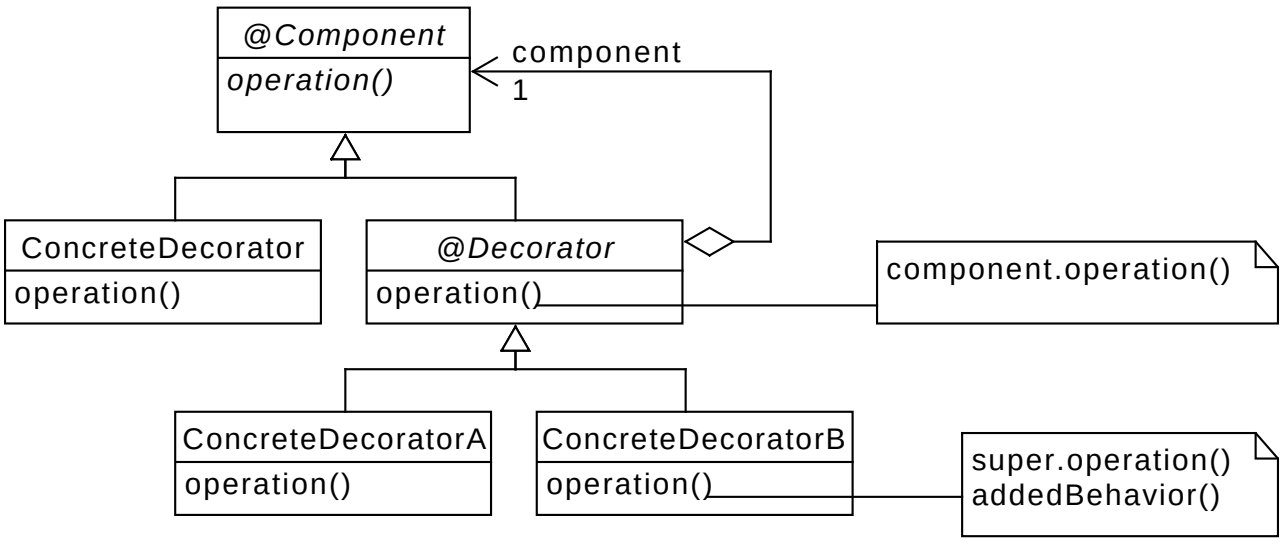
Design Pattern : Decorator

Intent

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

— utilisation de la composition en alternative à l'héritage
— évite la multiplication (voire l'explosion) des sous-classes à créer

Structure



Éléments caractéristiques

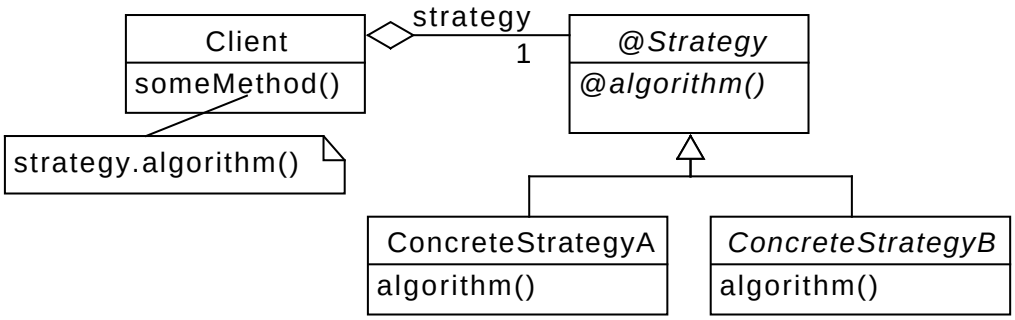
— un attribut du type de base dans le décorateur : l'élément décoré (`component`) ;
— par défaut dans les méthodes, le décorateur se contente de déléguer le travail au décoré, donc par défaut, du point de vue du comportement on ne fait pas la différence avec l'objet décoré ;
— la décoration se caractérise par un ajout d'attributs ou de méthodes (`addedBehavior()`) et en modifiant "à la marge" (subjectif) les comportements des méthodes du décoré.

Design Pattern : Strategy

Intent

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Structure



Éléments caractéristiques

— un attribut représentant la stratégie qui est d'un type abstrait

— l'invocation des méthodes de la stratégie, en faisant varier le type concret de la stratégie on change le comportement apparent de la classe cliente.

On remplace la variation du comportement que l'on peut obtenir par héritage par une variation du comportement par composition via la stratégie.

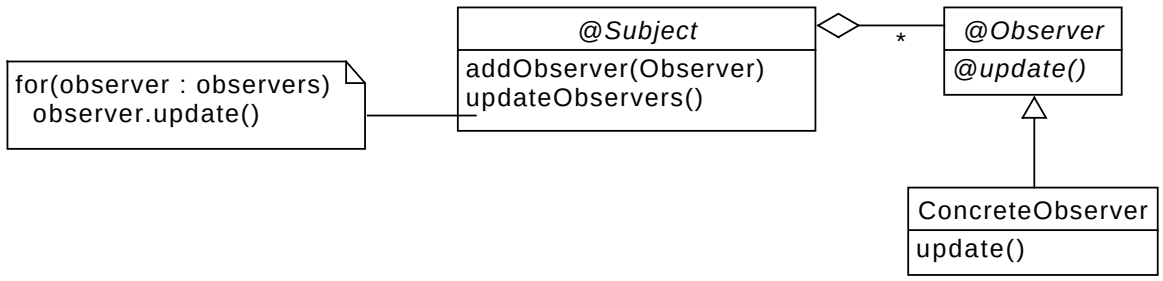
Design Pattern : Observer

Intent

> *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

Aussi appelé : *Abonneur/Abonné* ou *Event Idiom*.

Structure



Éléments caractéristiques

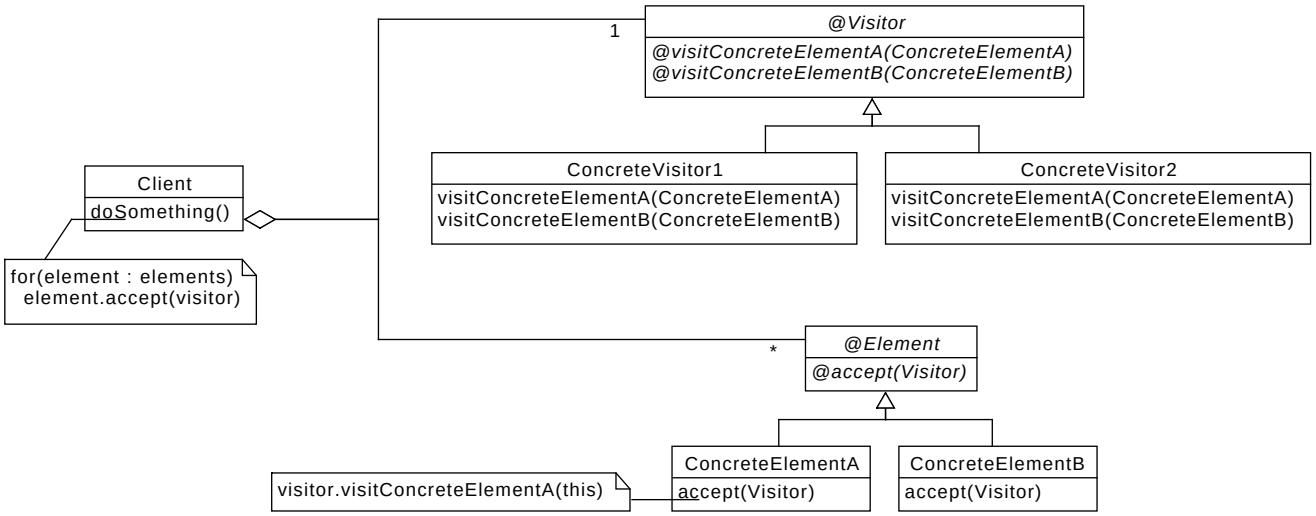
- Plusieurs objets (les `Observer`) peuvent être avertis des événements (`update()`) émis par une source (`Subject`, aussi appelée `Observable`) ;
- Le nombre et la nature des *observers* ne sont pas connus à la compilation et peuvent changer dans le temps ;
- L'émetteur de l'événement et le récepteur ne sont pas fortement liés.

Design Pattern : Visitor

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Structure



La classe *Client* représente les utilisateurs du *visiteur* : ses instances ont accès à des élements et décident comment les visiter.

Éléments caractéristiques

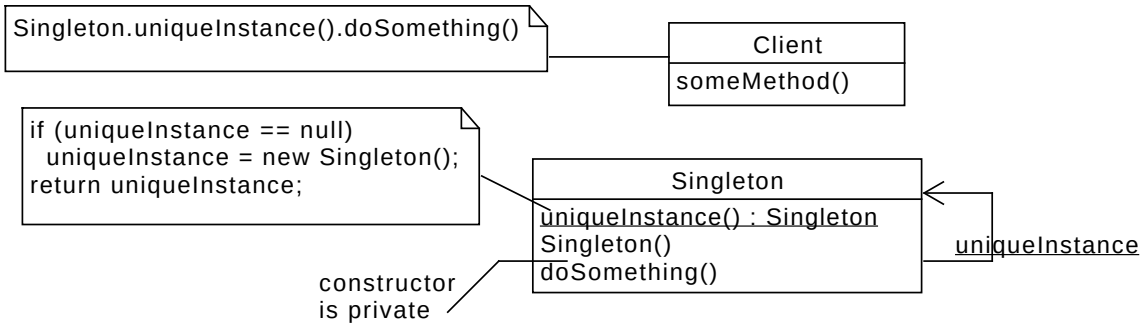
- Un ensemble de types d'éléments à exploiter : les *visitables*.
- une méthode d'"*acceptation*" qui est appelée sur les éléments afin qu'ils orientent l'appel vers la méthode du visiteur qui convient pour l'élément en question : inversion du contrôle.
  On a ainsi un même appel pour tous les objets éléments chacun d'entre eux choisissant la méthode de visite appropriée.

Design Pattern : Singleton

Intent

Ensure a class has only one instance, and provide a global point of access to it.

Structure



Éléments caractéristiques

- constructeur privé
- une instance `static final` soit `public` soit accessible par une méthode `public static`

```
public class SingletonClass {
  private SingletonClass() { }
  private static SingletonClass uniqueInstance = new SingletonClass();
  public static SingletonClass getInstance() {
    return uniqueInstance;
  }
}
```

ou

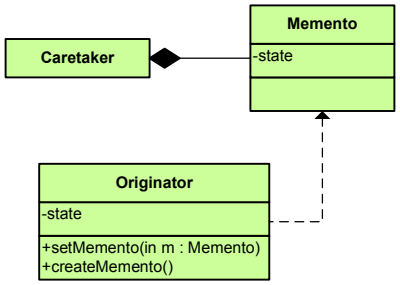
```
public class SingletonClass {
  private SingletonClass() { }
  public static final SingletonClass uniqueInstance = new SingletonClass();
}
```

C Abstract Factory
S Adapter
S Bridge
C Builder
B Chain of Responsibility
B Command
S Composite
S Decorator
S Facade
C Factory Method
S Flyweight
B Interpreter
B Iterator
B Mediator
B Memento
C Prototype
S Proxy
B Observer
C Singleton
B State
B Strategy
B Template Method
B Visitor

Memento

Type: Behavioral

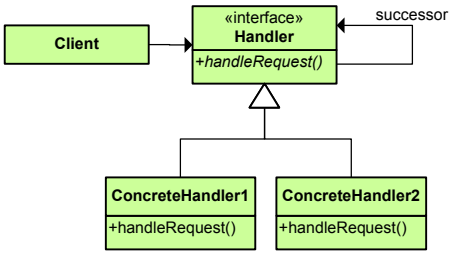
What it is:
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.



Chain of Responsibility

Type: Behavioral

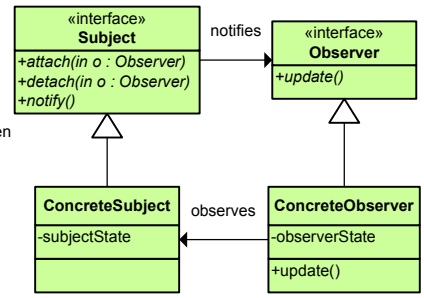
What it is:
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.



Observer

Type: Behavioral

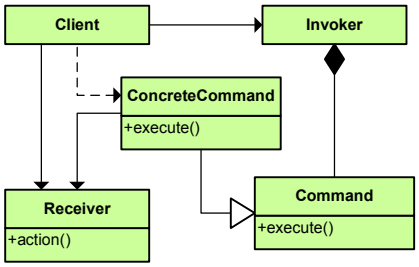
What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Command

Type: Behavioral

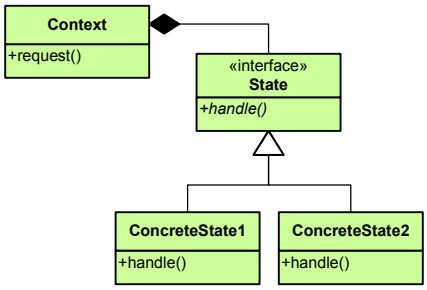
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.



State

Type: Behavioral

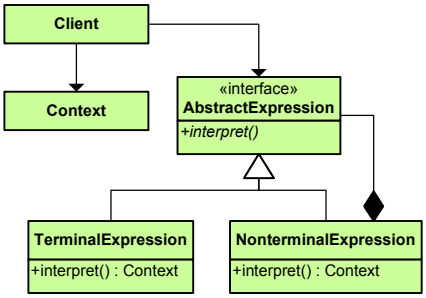
What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.



Interpreter

Type: Behavioral

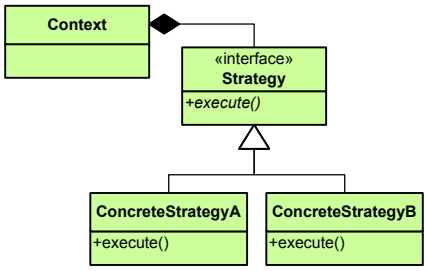
What it is:
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.



Strategy

Type: Behavioral

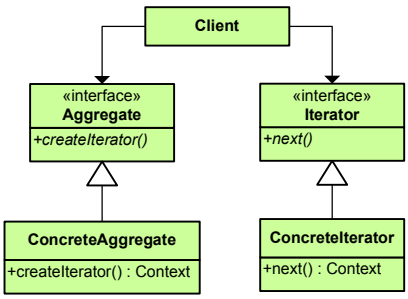
What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



Iterator

Type: Behavioral

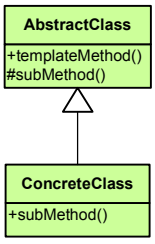
What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Template Method

Type: Behavioral

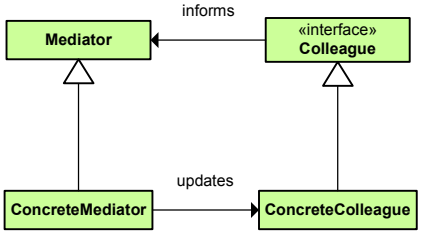
What it is:
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.



Mediator

Type: Behavioral

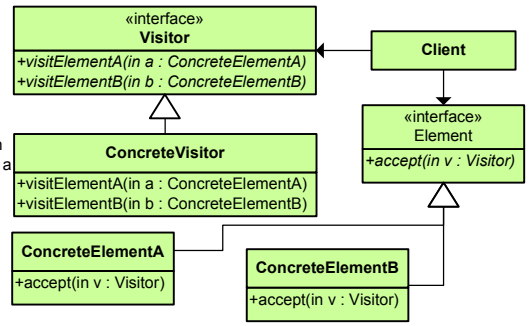
What it is:
Define an object that encapsulates how a set of objects interact. Promotes loose coupling by keeping objects from referring to each other explicitly and it lets you vary their interactions independently.



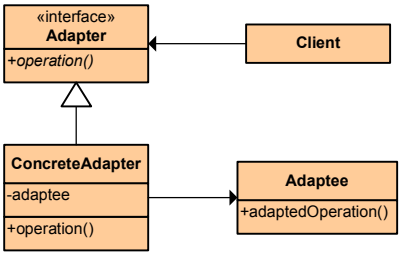
Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



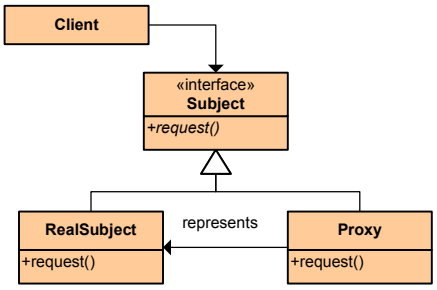
Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..



Adapter

Type: Structural

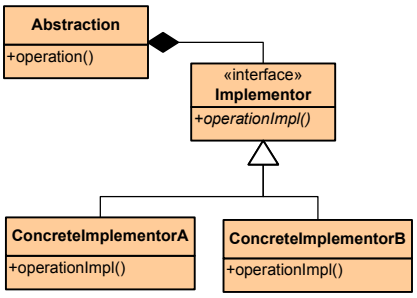
What it is:
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.



Proxy

Type: Structural

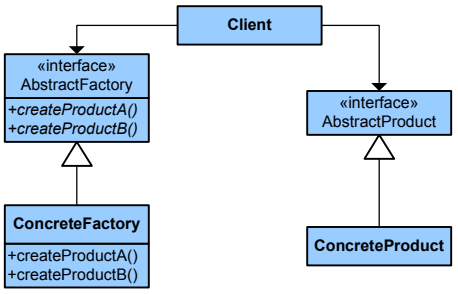
What it is:
Provide a surrogate or placeholder for another object to control access to it.



Bridge

Type: Structural

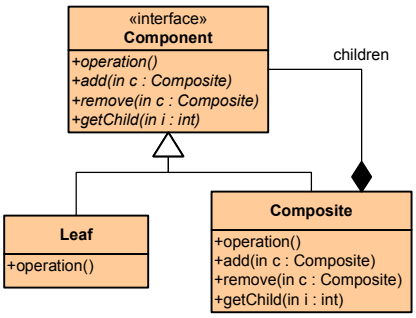
What it is:
Decouple an abstraction from its implementation so that the two can vary independently.



Abstract Factory

Type: Creational

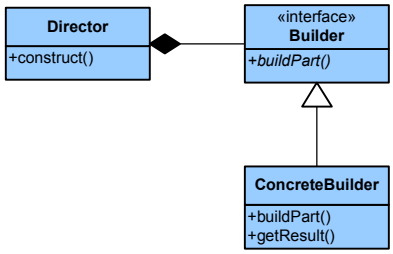
What it is:
Provides an interface for creating families of related or dependent objects without specifying their concrete class.



Composite

Type: Structural

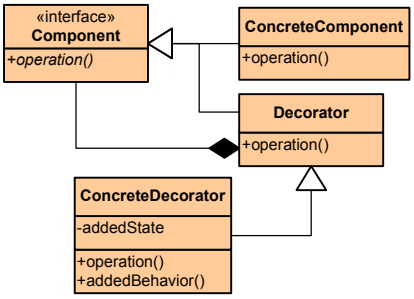
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.



Builder

Type: Creational

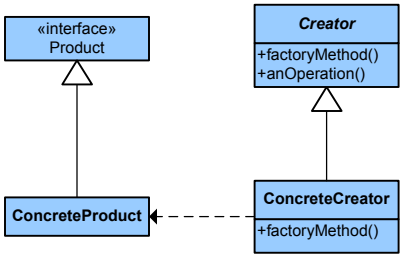
What it is:
Separate the construction of a complex object from its representing so that the same construction process can create different representations.



Decorator

Type: Structural

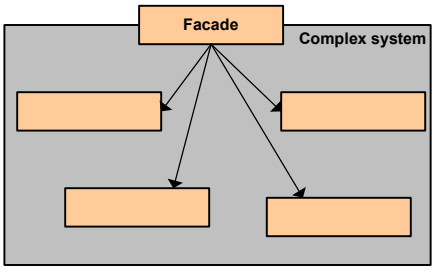
What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.



Factory Method

Type: Creational

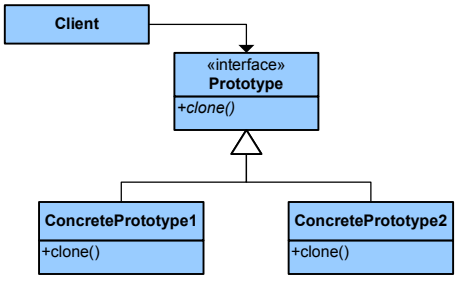
What it is:
Define an interface for creating an object, but let subclasses decide which class to instantiate. Lets a class defer instantiation to subclasses.



Facade

Type: Structural

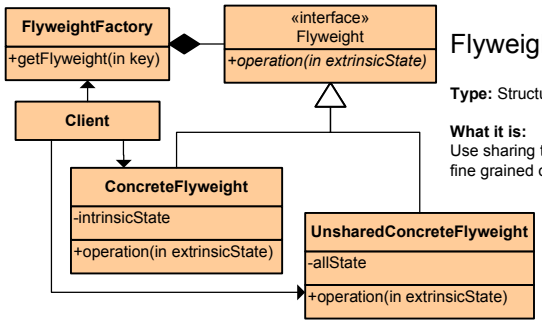
What it is:
Provide a unified interface to a set of interfaces in a subsystem. Defines a high-level interface that makes the subsystem easier to use.



Prototype

Type: Creational

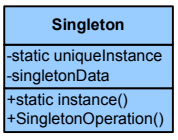
What it is:
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.



Flyweight

Type: Structural

What it is:
Use sharing to support large numbers of fine grained objects efficiently.



Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison Wesley Longman, Inc..