**Conception Orientée Objet et Design Patterns 2**

# Objectifs et enjeux

## Enjeux

Être capable de concevoir et développer en Java des programmes souples, extensibles et faciles à maintenir. Cela suppose de respecter les principes suivants afin de garantir une forte cohésion et un faible couplage :

* Responsabilité unique : une classe ne doit avoir qu’une seule raison de changer.
* Ouverture fermeture : une classe doit être ouverte aux extensions mais fermées aux modifications.
* Substitution de LISKOV : une méthode utilisant une référence vers une classe de base doit pouvoir référencer des objets de ses classes dérivées sans les connaitre (polymorphisme).
* Inversion de dépendance : une classe doit dépendre d’abstraction et non pas de classes concrètes.
* Ségrégation des interfaces : toute classe implémentant une interface doit implémenter chacune de ces fonctions.

## Moyens

L’objectif pédagogique de ce projet est de mettre en œuvre, conjointement dans une même application, plusieurs patterns en appliquant les bonnes pratiques de conception objet :

* Encapsuler ce qui varie.
* Préférer la composition à l’héritage.
* Programmer des interfaces (au sens abstraction) et non des implémentations :
  + En Java une abstraction est soit une interface soit une classe abstraite.
  + En Java, une implémentation est une classe instanciable.
* Coupler faiblement les objets qui interagissent.

## Points techniques abordés

* Programmation graphique et évènementielle : packages javax.swing et java.awt
* Framework de collections : package java.util
* Introspection de classes : packages java.lang et java.lang.reflect
* Tests unitaires : framework JUnit.

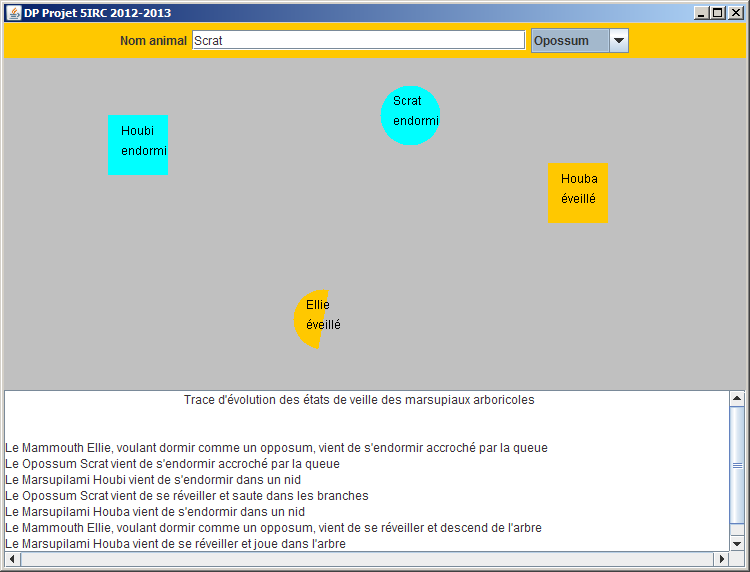
# Travail à faire

## Objectif du projet

Le but du programme est essentiellement pédagogique (désolée…) mais pourrait servir à implémenter quelques briques de base utilisées dans un jeu vidéo.

Le programme consiste à travailler sur une population d’animaux, en particulier sur des marsupiaux arboricoles, dont les principales caractéristiques sont de dormir et de vivre dans les arbres. Ils pourront être imités (mammouth se prenant pour un opossum), être considérés comme un tout ou une partie d’un tout et être observés. Une vue en mode console ou graphique de chaque individu se mettra à jour à chaque fois qu’ils changeront d’état (éveillé ou endormi).

Dans l’application graphique, les marsupiaux arboricoles pourront être crées par un clic gauche, supprimés par un clic droit, déplacés par un drag&drop (sans prendre une place déjà occupé) ; un double clic gauche leur permettra de changer d’état. Le choix de l’animal à créer se fera par un menu. Un trace des différents changements d’état sera à prévoir.



**Figure 1 : exemple de réalisation graphique**

La conception et le développement se feront en 4 étapes, les 3 premières se faisant en mode console.

Pour chaque étape, le diagramme UML et le code Java sont à réaliser. Les patterns mis en œuvre dans chaque classe doivent être identifiés. Les tests doivent permettre de vérifier le bon fonctionnement du programme à chaque sous étape.

## 1ère étape – 4h

* Tous les animaux ont un nom. Les marsupiaux arboricoles doivent pouvoir changer d’état. En fonction du type de marsupial, un libellé (court et long) relatif a leur état doit pouvoir être exploité (« dort accroché par la queue, par les bras, dans un nid », etc.) :
  + **Prévoir les interfaces et classes nécessaires et tester.**
* Rappel : une interface permet de définir un comportement. Ce comportement est implémenté par une classe ou une hiérarchie de classes.
* Penser réutilisation et factoriser tous les éléments communs dans les classes de base.
* D’autres animaux voudraient adopter le comportement des marsupiaux arboricoles :
  + **Adapter leur(s) classe(s) afin que ce soit possible et tester sur un exemple (mammouth ☺).**
* Les applications clientes voudraient pouvoir considérer une collection de marsupiaux arboricole (y compris des animaux adaptés) comme un seul individu :
  + **Déployer le DP Composite en ce sens et tester.**
* Dans le souci de programmer des interfaces et non des implémentations :
  + **Programmer une ou plusieurs fabriques abstraites (et concrètes) pour créer des marsupiaux arboricoles (adaptés ou non) et tester.** Les fabriques peuvent être des Singleton (pas indispensable dans ce contexte mais pédagogique…).

## 2ème étape – 8h

Lorsqu’un marsupial change d’état (éveil/sommeil) une vue devrait immédiatement se mettre à jour et s’afficher (1 en mode console, 1 en mode graphique plus tard) :

* **Décorer les marsupiaux arboricoles en vue de les rendre observables.** 
  + Il s’agit d’étendre leur comportement. Ce sont ces marsupiaux décorés qui seront manipulés dans la suite du projet.
  + **Déléguer la gestion des observateurs à un objet Strategy**. Ce ne doit pas être de la responsabilité des marsupiaux décorés de gérer les observateurs.
* **Créer un observateur (qui affiche l’état du marsupial sur la console).**
* **Définir les fabriques nécessaires.**
* **Tester.** Cela doit fonctionner aussi bien sur les marsupiaux arboricoles que sur leurs adaptations que sur les composites.

## 3ème étape – 8h

On imagine un Back Office qui stockerait/gérerait les différentes classes et fabriques et serait capable de construire un fichier XML, par exemple, contenant les noms des classes (des marsupiaux arboricoles) et des fabriques (+ éventuellement des noms des méthodes des fabriques). **On ne gèrera pas ce Back Office**.

Il s’agit donc d’être capable de construire des objets connaissant le nom (de type String) de la classe à instancier (constructeur avec ou sans paramètre) ou de la méthode (d’une fabrique dont on connait le nom) à invoquer (avec ou sans paramètres).

* **Écrire une classe d’introspection (non instanciable) qui permet de :**
  + Construire un objet connaissant le nom de la classe (String).
  + Construire un objet connaissant le nom de la classe (String) et la liste des arguments (Object []).
  + Invoquer une méthode connaissant l’objet (Object), le nom de la méthode (String), la liste des arguments (Object []).
  + Éventuellement, si les noms des méthodes des fabriques ne sont pas connus, invoquer une méthode connaissant le nom de la fabrique(String) ou la fabrique (Object), le nom de la classe de l’objet à fabriquer(String) et la liste des arguments (Object []).
* **Tester**. Cela doit fonctionner aussi bien sur les marsupiaux arboricoles que sur leurs adaptations que sur les composites. Inutile de gérer des fichiers, les noms des classes et fabriques peuvent être codés en dur dans le pgm de tests.

## 4ème étape – 8h

Le but du programme est de réaliser quelques briques de base d’un jeu vidéo (en mode graphique).

L’idée étant que après le choix d’un marsupial (ou adaptation) il soit positionné sur un fond par un clic gauche, supprimé par un clic droit, change de statut de veille par un double clic, soit déplacé par un drag&drop. Attention, un objet ne pourra être positionné à un emplacement déjà occupé (Cf. Trucs et astuces).

Les changements de statut (éveillé/endormi) sont observés de manières à ce qu’ils apparaissent ostensiblement sur l’objet sélectionné (au choix : couleur, forme, image, etc.). Ils devront également être tracés en détail au bas de l’écran.

### Contraintes

* L’IHM est au choix de chacun.
* La gestion de l’affichage et des opérations liées à la souris doivent pouvoir fonctionner de manière **générique** quels que soient les objets affichés (marsupiaux arboricoles, guerriers, voitures, robots, etc.).
* **Prévoir un contrôleur (DP Mediator)** qui gère les échanges entre les objets. Tous les objets connaissent le contrôleur et peuvent invoquer ces méthodes en revanche, ils ne communiquent pas directement entre eux (faible couplage).
* Les noms des fabriques, classes et éventuellement méthodes sont connues (String) et les méthodes de la **classe d’introspection doivent être invoquées.**
* **Dans le but de favoriser le couplage faible, penser DP Strategy et/ou Decorator. Ne pas négliger les Template Method. Les Observer sont nativement indispensables.** Les noms des patterns doivent apparaitre ostensiblement dans le diagramme UML et dans le code (en commentaire).

### Trucs et astuces

* Support cours et correction exercices du module DP du semestre 8 sur le e-campus.
* Documentation de référence Java : <http://docs.oracle.com/javase/1.5.0/docs/api/>
* Il peut être pratique pour vérifier que l’emplacement où l’on souhaite positionner un objet ne soit pas déjà occupé, de bénéficier des méthodes d’un objet de type java.awt.Rectangle ou java.awt.Polygon (Cf. <http://duj.developpez.com/tutoriels/java/dessin/intro/>).

En particulier par les méthodes (entre autre) :

* + getLocation() : retourne le java.awt.Point en haut à gauche du java.awt.Rectangle.
  + contains(Point ou Rectangle) : vérifie si le java.awt.Rectangle contient le java.awt.Point (ou Rectangle) spécifié.
  + intersect(Rectangle) : vérifie si le java.awt.Rectangle (ou Polygon) courant a au moins 1 point d’intersection avec le java.awt.Rectangle spécifié en paramètre.
* Les méthodes de l’écouteur MouseListener sont les suivantes :
  + mouseEntered() : entrée dans un composant,
  + mouseClicked() : clic,
  + mousePressed() : début du clic,
  + mouseReleased() : fin du clic,
  + mouseExited() : sortie d’un composant.
* Les méthodes de l’écouteur MouseMotionListener sont les suivantes :
  + mouseDragged() : déplacement bouton enfoncé,
  + mouseMoved() : déplacement dans un composant.

## Pour les plus rapides

Prévoir des procédures de tests des classes écrites en utilisant les méthodes du Framework JUnit.

# Modalités pratiques

## Organisation

* Ce travail se réalise en binôme ; les 10 binômes sont imposés (Cf. liste §3.5).
* Un travail **original** (entendre individuel…) est attendu de chaque binôme. Il n’est cependant pas interdit de travailler en collaboration avec d’autres binômes, **le préciser** dans ce cas mais il est interdit de s’échanger du code. Il convient également de **citer ses sources** (Internet ou autre).
* Chaque étape doit être réalisée dans les heures impartis (voir dans le sujet). Si ce n’est pas suffisant, il conviendra d’étudier avec l’enseignante à la fin de chaque période ce qu’il convient de faire : travail supplémentaire en dehors des heures dédiées ou révision à la baisse des ambitions.
* Les 8 séances sont encadrées.

## Suivi

* Les modèles de conception (des étapes 1, 2 et 4) doivent être validés par l’enseignante avant de commencer le codage (sauf en cas d’attente trop longue ☺).
* Les différents échanges avec l’enseignante et l’état d’avancement de chaque binôme seront reportés au fil des séances sur un tableau de bord. **Chaque membre** du binôme sera invité à s’exprimer et à justifier les choix de conception et de programmation.
* En cas de travail en dehors des séances encadrées, toute modification fondamentale, non dûment justifiée, par rapport à la version précédente ou présentant des similitudes avec le travail d’un autre binôme, sera considérée comme une **tentative de fraude** et sera traitée selon les dispositions prévues dans le règlement des études.

## Rendus

* Chaque fin d’étape (théorique) fera l’objet d’un livrable : diagramme UML (éventuellement sous forme papier) et sources Java. Un dépôt de fichier sera ouvert à cet effet sur le e-campus.

## Évaluation

* DS 50%, Projet 50%.
* Dans la note de projet seront appréciés :
  + La maitrise (conceptuelle) des patterns.
  + L’aisance en Java.
  + La cohérence des conceptions avec les réalisations (à chaque étape).
  + La pertinence des réponses aux questions posées (à l’oral en cours de séance).
  + L’exhaustivité des procédures de test.
  + L’originalité de la solution finale.
  + Le tout étant pondéré par la progression des compétences de chacun.
* Une note individuelle pourra être donnée aux différents membres d’un même binôme en cas de différence de niveau flagrante.
* Le DS sera sous forme papier/crayon avec documents.

## Liste des binômes

* BERTHELOOT – MARCHELLI
* CADHIHAC – DOLLINGER
* CHASTAGNIER – ROUVIER
* CO – GAUJAL
* FERROLIET – GRAIL
* HOAREAU – RENOULT
* MANIEZ – MOYSSAN
* MARENCO – PRIGNON
* MARTIN – PIEGAY
* MULLIER – TARCHOUNI