Génie Logiciel Avancé
Cours 8 : Conception d'un système d'objets

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

vendredi 19 mars 2010

Architecture d'un système à objets

- Les différents styles architecturaux du dernier cours donnent des pistes pour organiser un système suivant multiples niveaux d'abstractions, multiples modèles de calcul et multiples composantes de façon à minimiser les **interdépendances** et maximiser la **cohésion**.
- Les objets (et les modules) sont des briques élémentaires permettant de réaliser ces architectures puisque :
  - Ils aident à maximiser la cohésion grâce à l'encapsulation de leur état.
  - Les classes abstraites (et l'héritage) permettent de minimiser les interdépendances en contrôlant le niveau de détails des services visibles par les clients d'un objet.

⇒ Le sujet d'aujourd'hui :

Comment agencer ces briques ?

Deux difficultés

1. La conception de composants réutilisables est une activité délicate.
2. La programmation objet a des problèmes intrinsèques.

Les patrons de conception : des « recettes »

- Un programmeur ou un concepteur expérimenté réutilise des procédés d'organisation qui ont fonctionné dans le passé.
- Ces solutions sont indépendantes (en général) du langage objet utilisé.
- Votre expérience :
  - XHTML + CSS : séparer un contenu de la description de sa mise en forme.

  Idée à retenir :
    - un composant traitant les données (modèle),
    - un composant de visualisation (vue).
  - Compilateur : composition de traductions bien spécifiées.

  Idée à retenir :
    - la fameuse *separation of concerns* de Dijkstra.
    - "Un composant bien conçu se focalise sur un unique problème."

~Menu~

Création

Fabrique abstraite (Abstract Factory)
Monteur (Builder)
Fabrique (Factory Method)
Prototype (Prototype)
Singleton (Singleton)

Structure

Adaptateur (Adapter)
Pont (Bridge)
Objet composite (Composite)
Décorateur (Decorator)
Façade (Facade)
Poids-mouche ou poids-plume (Flyweight)
Proxy (Proxy)

Comportement

Chaîne de responsabilité (Chain of responsibility)
Commande (Command)
Interpréteur (Interpreter)
Itérateur (Iterator)
Médiateur (Mediator)
Memento (Memento)
Observateur (Observer)
État (State)
Stratégie (Strategy)
Patron de méthode (Template Method)
Visiteur (Visitor)

- Chefs : Gamma, Helm, Johnson, Vlissides (GoF).
- Livre de recettes (référence) :
  Design Patterns :
  Elements of Reusable Object-Oriented Software.
  Addison-Wesley

  (source : Wikipedia)

Classification des patrons de conception

- GoF ont explicité trois grandes classes de patrons dans leur livre, chacune spécialisée dans :
  - la création d'objets ;
  - la structure des relations entre objets ;
  - le comportement des objets.
- D'autres catégories ont été repertoriées pour la destruction des objets, la concurrence, la persistance . . .

Patrons de conception de « Création et Destruction »

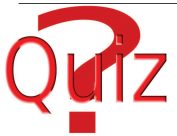
Fabrique (Factory Method) – Description

- Exemple de situation :

  Je veux fournir un composant réutilisable implémentant un dictionnaire. Ce dictionnaire doit être implémenté à l'aide de la structure de données la plus adaptée possible compte tenu du nombre d'entrées qu'il va contenir (par exemple, une table de hachage tant que sa taille n'excède pas quelques mégas, une base de données sinon). Les prochaines versions de mon composant intègreront de nouvelles structures de données. J'utilise donc une classe abstraite Dictionnary et une sous-classe par structure de données.

- Problème : Comment bien choisir la structure de données sans rendre publique l'existence des sous-classes ?
- Solution : On fournit une classe *DictionaryFactory* qui répond au message *CreateDictionary (n)* où *n* est le nombre potentiel d'entrées dans le dictionnaire. L'implémentation de cette classe choisit la bonne sous-classe et en retourne une instance vue comme un *Dictionary*. On ne divulgue donc que *DictionaryFactory* et *Dictionary*.

Fabrique (Factory Method) – en UML



Dessiner le diagramme de classes correspondant !

Fabrique (Factory Method) – Interface en C++

```
// Dictionary from string to string.
class Dictionary {
public:
  struct NotFound {};

  virtual Dictionary*
  addEntry (const std::string& key, const std::string& value) = 0;

  virtual const std::string&
  operator[](const std::string& key) const throw (NotFound) = 0;
};
// Dictionary factory is choosing the best implementation
// given the potential number of entries.
class DictionaryFactory {
public:
  // You are responsible for the life of the resulting dictionary.
  Dictionary* create (long int potential_entry_number);
};
```

Fabrique (Factory Method) – Implémentation en C++

```
class RedBlackTreeDictionary : public Dictionary {
  virtual Dictionary*
  addEntry (const std::string& key, const std::string& value) { ... }

  virtual const std::string&
  operator[](const std::string& key) const throw (NotFound) { ... }
};

class DataBaseDictionary : public Dictionary {
  virtual Dictionary*
  addEntry (const std::string& key, const std::string& value) { ... }

  virtual const std::string&
  operator[](const std::string& key) const throw (NotFound) { ... }
};

Dictionary* DictionaryFactory::create (long int potential_entry_number) {
  if (potential_entry_number ≤ 100000000L)
    return (new RedBlackTreeDictionary ());
  else
    return (new DataBaseDictionary ());
}
```

Fabrique (Factory Method)

- Une fabrique peut aussi servir à différencier les constructeurs de classe à l'aide d'un nom.
- En effet, il n'est pas toujours évident de dissocier les constructeurs seulement à l'aide de leur signature (*ie* du types des arguments).
- Dans notre exemple, on peut vouloir construire un dictionnaire à l'aide d'un entier correspondant :
  - au nombre moyen d'entrées ;
  - ou au nombre minimal d'entrées ;
  - ou au nombre maximal d'entrées.
- Les constructeurs auraient tous la même signature !
- Dans ce cas, on peut plutôt donner un nom explicite aux messages de la fabrique pour dénoter le cas de construction (*createFromMean*, *createFromMin*, *createFromMax*).

Fabrique (Factory Method) – Vision fonctionnelle

- Introduire une indirection pour la construction des données est un procédé d'encapsulation très courant dans tous les paradigmes de programmation.
- Lorsque le langage de programmation fournit des types abstraits, on implémente des « constructeurs intelligents » (*smart constructor*).
- En O'Caml :

```
module Dictionary : sig
  type t (* Abstract type of dictionaries. *)
  val mk_dictionary : int → t (* Smart constructor. *)
  (* Functionalities *)
  val add_entry : string → string → t → t
  val get_entry : string → t → t
end =
struct
  type t =
    | RedBlackTreeDictionary of RedBlackTreeDictionary.t
    | DataBaseDictionary of DataBaseDictionary.t

  let mk_dictionary nb_potential_entries =
    if nb_potential_entries ≤ 10000000L then
      RedBlackTreeDictionary RedBlackTreeDictionary.empty
    else
      DataBaseDictionary DataBaseDictionary.empty
end
```

Fabrique abstraite (Abstract Factory) – Description

- Exemple de situation :

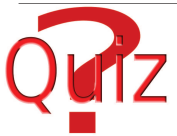
  Je veux distribuer une nouvelle version de mon composant : les dictionnaires peuvent maintenant être persistants (la méthode `addEntry` renvoie une nouvelle instance du dictionnaire sur le tas et ne détruit pas la précédente).

- Problème : Comment minimiser la quantité de code à modifier pour assurer la migration vers la nouvelle version (sans pour autant divulguer les sous-classes mise en jeu) ?
- Solution : Appliquer le motif de Fabrique à la Fabrique elle-même !
  La classe *DictionaryFactory* devient abstraite. On crée une classe *PersistantDictionaryFactory* et une classe *NonPersistantDictionaryFactory*. On peut ensuite définir une classe *DictionaryFactoryCreator* qui choisit d'instancier la bonne fabrique en fonction d'un booléen.

Abstraction de l'allocation et gestion explicite de la mémoire

```
typedef std::list< std::pair<std::string, std::string> > entry_list;

Dictionnary*
add_list (Dictionnary* dict, const entry_list& entries) {
  Dictionnary* accu = dict;
  for (entry_list::iterator i = entries.begin (); i != entries.end (); ++i)
    accu = accu→add_entries (i→first, i→second);
  return (accu);
}
```



Quelle est l'empreinte mémoire de l'exécution de ce programme ?

Pointeur intelligent (Smart Pointer) – Description

(ce patron s'applique seulement si le langage d'implémentation de propose pas de ramasse-miettes)

- Exemple de situation :

  Après leur migration à la version persistante de mon composant, les utilisateurs se plaignent de la présence de fuites de mémoire.

- Problème : Lors le mécanisme d'allocation est abstrait, on ne peut pas demander au client de désallouer l'instance. En d'autres termes, il faut aussi implémenter un mécanisme abstrait de désallocation.
- Solution : Les objets implémentent le comportement de pointeurs munis d'un mécanisme de destruction automatique lorsque l'on ne les utilise plus. Il existe plusieurs implémentations de ces pointeurs « intelligents » :
  - les auto-pointeurs (présents dans la bibliothèque standard de C++) ;
  - les compteurs de références (présents dans la bibliothèque Boost).

Pointeur intelligent (Smart Pointer)

- Dans notre exemple, la méthode *addEntry* renvoie désormais un pointeur intelligent vers un dictionnaire.

  boost::shared_ptr<Dictionary> addEntry (...) ;

- Il est malheureusement nécessaire de modifier le code client car la signature de la méthode *addEntry* a changé.

Monteur (Builder) – Description

- Exemple de situation :

  Pour obtenir une partie d'échec pleinement initialisée, je dois avoir un plateau, attendre que deux joueurs soient prêts à jouer, s'accorder sur le type de partie voulue, et enfin déterminer la couleur de chacun des joueurs.
  Mon programme permet de jouer sur différentes formes de plateau, les joueurs peuvent se connecter localement ou via un serveur, il y a 30 types de partie proposés . . .

- Problème : Comment décorréler le protocole de création d'une partie de la nature de ces parties ?
- Solution : Un objet *GameDirector* fait office de fabrique implémentant le protocole en appelant les fabriques des différents composants de la partie dans le bon ordre.

Prototype (Prototype) – Description

- Exemple de situation : On veut créer des modèles de partie d'échec (par exemple, une partie d'échec en Blitz sur un plateau en 3 dimensions avec deux joueurs sur le réseau ou encore une partie d'échec sans limitation de temps sur un plateau en ASCII avec un joueur local jouant contre un programme, . . .).
- Problème : Doit-on faire une classe de fabrique par combinaison ?
- Solution : Pour éviter la multiplication des sous-classes, on représente une combinaison particulière des paramètres à l'aide d'une instance d'une classe *GamePrototype* qu'on est capable de cloner.

Prototype (Prototype) – En Scala (1/2)

```
abstract class Board { def cloneMe () : Board }
class Board2d extends Board { override def cloneMe () = this }
class Board3d extends Board { override def cloneMe () = this }

abstract class Kind { def cloneMe () : Kind }
class Blitz extends Kind { override def cloneMe () = this }
class Standard extends Kind { override def cloneMe () = this }

abstract class Player { def cloneMe () : Player }
class LocalPlayer extends Player { override def cloneMe () = this }
class AIPlayer extends Player { override def cloneMe () = this }

class Game (board : Board, kind : Kind, player_1 : Player, player_2 : Player) {
  var _board : Board = board
  var _kind : Kind = kind
  var _player_1 : Player = player_1
  var _player_2 : Player = player_2

  def play () = ...
}
```

Prototype (Prototype) – En Scala (2/2)

```
class GamePrototypeFactory (board : Board, kind : Kind, player_1 : Player, player_2 : Player) {
  var _board : Board = board
  var _kind : Kind = kind
  var _player_1 : Player = player_1
  var _player_2 : Player = player_2

  def makeGame () = new Game (_board.cloneMe (), _kind.cloneMe (),
                              _player_1.cloneMe (), _player_2.cloneMe ())
}

val standard_game_factory =
  new GamePrototypeFactory (new Board2d, new Standard, new LocalPlayer, new LocalPlayer)

val alone_game_factory =
 new GamePrototypeFactory (new Board2d, new Standard, new LocalPlayer, new AIPlayer)

val blitz_game_factory =
 new GamePrototypeFactory (new Board2d, new Blitz, new LocalPlayer, new AIPlayer)
```

Prototype (Prototype) – Dans un langage fonctionnel

- Dans un cadre fonctionnel, on peut composer les constructeurs pour obtenir de nouveaux constructeurs (ce sont des objets de première classe).
- En O'Caml, si on a :

```
val mk_board_2d : unit → Board.t
val mk_board_3d : unit → Board.t
val mk_local_player : unit → Player.t
val mk_ai_player : unit → Player.t
val mk_blitz : unit → Kind.t
val mk_standard : unit → Kind.t
```

alors on peut écrire :

```
let mk_game mk_board mk_kind mk_player_1 mk_player_2 =
  (mk_board (), mk_kind (), mk_player_1 (), mk_player_2 ())

let mk_standard_game () =
  mk_game mk_board_2d mk_local_player mk_local_player mk_standard

let mk_alone_game () =
  mk_game mk_board_2d mk_local_player mk_ai_player mk_standard

let mk_blitz_game () =
  mk_game mk_board_2d mk_local_player mk_ai_player mk_blitz
```

Singleton (Singleton) – Description

- Exemple de situation :

  Un programme s'exécute dans un environnement qui a une existence physique particulière (un système d'exploitation, un numéro de processus, ...). On veut le représenter par un objet qui est une instance d'une classe `Environment`.

- Problème : Pour une exécutation donnée, il ne doit exister qu'une unique instance de la classe *`Environment`*.
- Solution : On empêche l'instanciation explicite (par un constructeur) de la classe *`Environment`*. Une classe *`EnvironmentSingleton`* est fournie et offre le service *`getInstance`* qui construit une instance de la classe *`Environment`* la première fois qu'il est appelé puis renvoie cette même instance lors des invocations suivantes.

Singleton (Singleton) – En C++

```
class EnvironmentSingleton;

class Environment {
private:
   Environment () {}
   friend class EnvironmentSingleton;
};

class EnvironmentSingleton {
private:
   static Environment* _instance;

public:
   Environment& getInstance() {
      if (EnvironmentSingleton::_instance == NULL) {
         EnvironmentSingleton::_instance = new Environment ();
      }
      return (*EnvironmentSingleton::_instance);
   }
};

Environment* EnvironmentSingleton::_instance = NULL;
```

Singleton (Singleton) – Critique

- Le patron Singleton permet de définir des variables globales dont l'initialisation est contrôlée.
- On peut raffiner l'implémentation précédente pour s'assurer que la classe se comporte bien dans un environnement concurrent à l'aide de verrous (*mutex*).
- L'utilisation de variables globales peut mettre en danger l'encapsulation et l'extensibilité du système. Il faut donc utiliser ce patron avec parcimonie.
- Pour une discussion intéressante sur ce sujet :

  `http://www.codingwithoutcomments.com/2008/10/08/singleton-i-love-you-but-youre-bringing-me-down/`

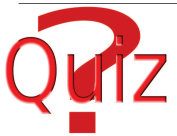
Réserve d'objets (Object pool) – Description

- Exemple de situation :

  Un serveur peut traiter un nombre N de clients simultanés. Chaque client se voit allouer des ressources dont l'allocation est longue (une connection sur une base de données externes, des périphériques systèmes, . . .).

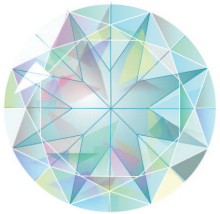
- Problème : Lorsque les ressources utilisées par chaque instance sont coûteuses et bornées, on veut contrôler le nombre d'instances d'une classe donnée.
- Solution : On peut utiliser un gestionnaire de ressources (une fabrique) fournissant un service *acquire*, donnant accès à une instance si c'est possible, et un service *release* permettant de réinjecter l'instance dans l'ensemble des instances accessibles.
- Consultez : `http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf`

Réserve d'objets (Object pool) – Réflexion



Que faire si il ne reste plus d'objets en réserve ?

Patrons de conception de “Structure”



Adaptateur (Adapter) – Description

- Exemple de situation :

  J'utilise une bibliothèque de traitement d'images (dont je ne peux pas modifier le code source). Pour fonctionner, elle attend un objet fournissant une interface d'accès en lecture et en écriture à un tableau en deux dimensions contenant des triplets d'octets. J'aimerais l'interfacer avec une bibliothèque d'entrées/sorties fournissant une abstraction sur des tableaux unidimensionnels stockés de manière persistante dans une base de données ou dans un système de fichiers.

- Problème : Comment concilier les services proposés par la bibliothèque d'entrées/sorties et l'interface attendue par la bibliothèque de traitement d'images.
- Solution : Utiliser un objet qui implémente l'interface attendue en faisant appel aux services proposés par une instance de la bibliothèque d'entrées/sorties.

Adaptateur (Adapter) – en JAVA

```
public interface ColorImage2D {
    public int get_width (int x);
    public int get_height (int x);
    public short get_red (int x, int y);
    public short get_green (int x, int y);
    public short get_blue (int x, int y);
}

public static class ImageProcessing {
    public static void blur (ColorImage2D image) {}
}
```

Bibliothèque de traitement d'images.

```
public class IOArray {
    public short get (int x) { return 0; }
    public int size () { return 0; }
}
```

Bibliothèque d'entrées/sorties.

Adaptateur (Adapter) – en JAVA

```
public class IOArrayToColorImage2DAdapter implements ColorImage2D {
    private int width;
    private int height;
    private IOArray array;

    IOArrayToColorImage2DAdapter (IOArray array, int width, int height) {
      assert (array.size () == width * height);
      this.width = width;
      this.height = height;
      this.array = array;
    }

    public int get_width (int x) { return width; }
    public int get_height (int x) { return height; }
    public int get_point (int x, int y) { return array.get (y * width + x); }
    public short get_red (int x, int y) { return (short) (this.get_point (x, y) & 0xff); }
    public short get_green (int x, int y) { return (short) (this.get_point (x, y) >> 8 & 0xff); }
    public short get_blue (int x, int y) { return (short) (this.get_point (x, y) >> 16 & 0xff); }
}
```

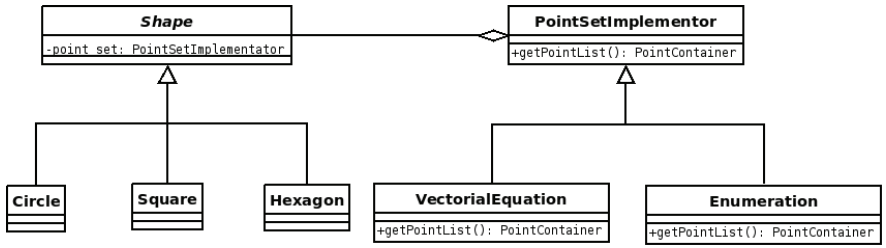
Pont (Bridge) – Description

- Exemple de situation :

  Un logiciel de dessin propose une hiérarchie de formes (carré, cercle, losange, . . .). Ces formes sont implémentées comme des ensembles de points qui peuvent être décrits de multiples façons (équation vectorielle, énumération, image bitmap, . . .).

- Problème : Pour éviter la multiplication des classes et modulariser le système, il est nécessaire de décorréler la hiérarchie des abstractions (les formes) de leurs implémentations (les ensembles de points) car elles peuvent varier de manière indépendante.
- Solution : Les implémentations sont organisées dans une hiérarchie indépendante. Une instance de la classe abstraite du sommet de cette hiérarchie (une classe nommée *Implementor*) est aggrégée à toute forme.

Pont (Bridge) – Diagramme UML



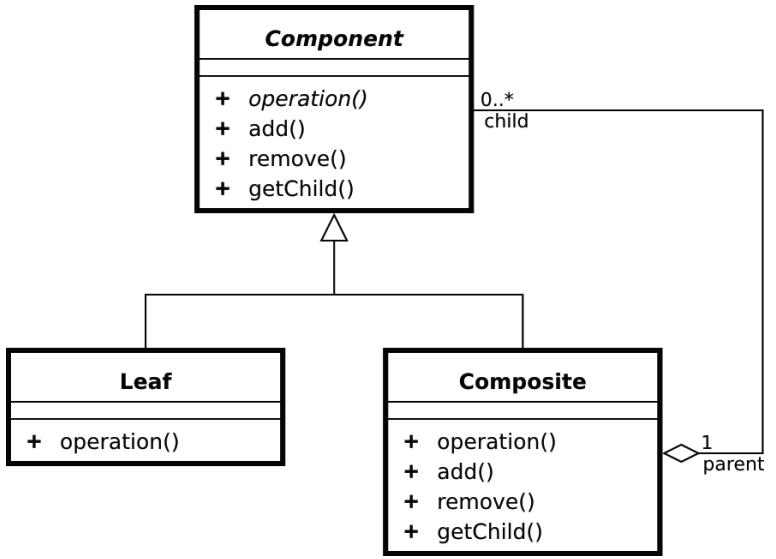
Objet composite (Composite) – Description

- Exemple de situation :

  Dans un logiciel de dessin, il est possible de créer des groupes de formes. Un groupe de formes est alors considéré comme une forme .

- Problème : Comment voir une composition de formes comme une forme ?
- Solution : Fournir un objet *Shape* qui implémente l'interface d'une forme ainsi que les services *add* et *remove*. Un objet *ShapeComposite*, héritant de *Shape*, correspond à un nœud possédant des sous-composants qui sont des formes. Lorsqu'un message est envoyé a une classe composite, elle le transfère à toutes ses composantes.

Objet composite (Composite) – Diagramme UML



(source : http://en.wikipedia.org/wiki/Image:Composite_UML_class_diagram.svg)

Objet composite (Composite) – Critique

- L'implémentation des méthodes *add* et *remove* n'a pas vraiment de sens pour les formes atomiques (cercle, carré, ...). On perd ainsi la sûreté (une exception sera lancée si le message *add* est envoyé à un cercle).
- On pourrait n'imposer ces méthodes que dans la classe *ShapeComposite*.
- On perd alors la transparence car on doit alors avoir un traitement différent pour le cas atomique et pour le cas composé.

⇒ Le GoF estime qu'il s'agit d'un compromis à décider au cas par cas ... La sûreté du logiciel doit, à mon avis, prévaloir.

Objet composite *sûr* (Safe composite) – en O'Caml

```
class virtual shape = object
  method virtual draw : unit → unit
  method virtual as_composite : composite option
end
and composite = object (self)
  inherit shape
  val mutable children : shape list = []
  method as_composite = Some (self : ¿ composite)
  method add c = children ← c : : children
  method remove c = children ← List.filter (( = ) c) children
  method draw () = List.iter (fun c → c#draw ()) children
end
and circle = object
  inherit shape
  method draw () = (* ... *)
  method as_composite = None
end
```

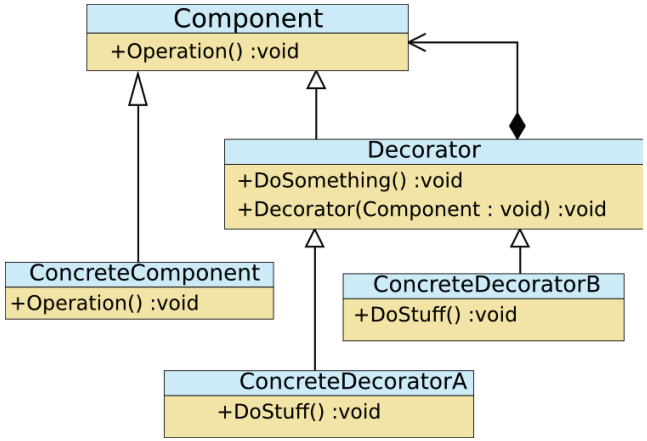
Décorateur (Decorator) – Description

- Exemple de situation :

  Dans un traitement de texte, un document est formé de différents composants (texte, image, figure, ...). L'utilisateur doit pouvoir associer une URL à chacun de ces composants. Une fois associé, on peut cliquer sur le composant pour suivre le lien.

- Problème : Comment rajouter une fonctionnalité dynamiquement à un objet ?
- Solution : Une classe Décorateur aggrège un composant particulier et implémente l'interface d'un composant en délégant tous les messages correspondants à son instance aggrégé. En héritant d'un Décorateur, on peut rajouter un comportement particulier, comme par exemple, la gestion des hyperliens.

Décorateur (Decorator) – Diagramme UML



(source : http://en.wikipedia.org/wiki/Image:Decorator_Pattern_ZP.svg)

Façade (Facade)

- Exemple de situation :

  Pour traiter les données d'un programme de gestion de bibliothèque multimédia, je désire utiliser l'API de PostgreSQL, *un gestionnaire de base de données très élaboré. Mon utilisation de la base de données est essentiellement un dictionnaire associant des noms de fichier et des descriptions à des noms symboliques.*

- Problème : Comment fournir une interface simple en utilisant un système complexe ?
- Solution : Le patron facade implémente seulement les méthodes utiles à mon dictionnaire en utilisant l'API de PostgreSQL. Il s'agit donc d'implémenter une couche d'abstraction.

Poids-mouche ou poids-plume (Flyweight) – En O'Caml

- Dans les langages fonctionnels, on peut implémenter ce procédé de façon générale pour toute fonction.

```
let memoize f =
  let table = Hashtbl.create 21 in
    fun x →
      try Hashtbl.find x
      with Not_found →
        let y = f x in
          Hashtbl.add x y ;
          y
```

- memoize f calcule une fonction qui exécute f au plus une fois pour une certaine valeur d'entrée.
- Il est possible de raffiner cette implémentation pour pouvoir « oublier » certains calculs.

Poids-mouche ou poids-plume (Flyweight) – Description

- Exemple de situation :

  Un compilateur traduit un code source en code machine. Une des premières phases consiste à savoir à quoi les identifiants utilisés dans le source font référence. Ce travail de résolution peut-être coûteux en temps et en espace (pour stocker la description de l'objet auquel on fait référence). Dans un même espace de noms, on voudrait ne pas avoir à refaire cette résolution plusieurs fois et partager les descriptions.

- Problème : Comment partager les calculs et le résultat de ces calculs de façon transparente ?
- Solution : Les identificateurs sont créés par une fabrique. Cette dernière stocke une table associant un descripteur à un identifiant. Si l'identifiant n'est pas dans la table, le calcul concret est effectué, sinon on renvoie l'élément déjà calculé.

Proxy (Proxy)

- Exemple de situation :

  Un système de navigation par GPS charge des images satellites de la planète par le réseau, c'est une opération très coûteuse. On veut pourtant voir l'ensemble de toutes ces images comme une unique image en profitant du fait que le téléchargement d'une section n'est nécessaire qu'au moment où cette section est effectivement affichée.

- Problème : Comment implémenter l'interface d'une image sans avoir toutes les données de cette image chargées en mémoire.
- Solution : Une classe *`WorldImageProxy`* implémente l'interface d'une image en chargeant ses attributs de manière paresseuse lors d'une réception de message nécessitant des données concrètes.

Comportement

Chaîne de responsabilité (Chain of responsibility) – Description

- Exemple de situation :

  On veut intégrer un mécanisme de greffons (plugins) a un traitement de texte pour pouvoir rajouter dynamiquement la gestion de certains types de composants.

- Problème : Comment écrire une fois pour toute la fonction qui choisit le bon greffon à utiliser par observation du type du composant à traiter ?
- Solution : On se donne un identifiant unique pour les types de composants à traiter. Cet identifiant va servir de message de première classe. On définit une interface pour les greffons contenant une méthode d'interprétation de ces messages. La liste des greffons est sotckée par l'application. On itère sur cette liste : lorsqu'un greffon ne sait pas traiter un message, il le transfère au greffon suivant.

Commande (Command) – Description

- Exemple de situation :

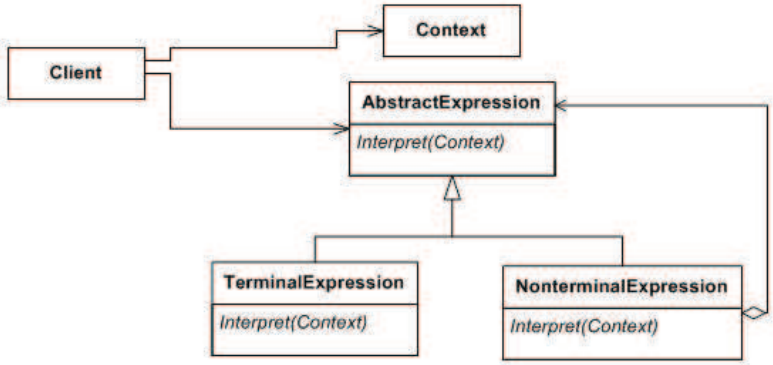
  Un logiciel de traitement d'image propose un ensemble de filtres à l'utilisateur. On veut donner la possibilité à l'utilisateur de construire ses propres filtres comme combinaison de filtres existants. On veut aussi lui donner la possibilité d'associer son filtre à un bouton de son application.

- Problème : Comment utiliser des opérations comme des données ?
- Solution : Il suffit d'implémenter un objet contenant une méthode `execute`... On peut alors associer ces fonctions à d'autres objets.

Interpréteur (Interpreter) – Description

- Exemple de situation : Composer les filtres n'est pas suffisant, un utilisateur avancé doit pouvoir programmer ses propres filtres (visuellement ou textuellement).
- Problème : Comment représenter la syntaxe d'un langage à l'aide d'objet ?
- Solution : Une classe abstraite *`Expression`* dont dérive les diverses constructions du langage.

Interpréteur (Interpreter) – Description



Itérateur (Iterator)

- Exemple de situation :

  Revenons à notre bibliothèque de dictionnaires. Je veux maintenant proposer un service permettant à un client d'inspecter l'ensemble des couples (clé, valeur) de mon dictionnaire sans divulguer la façon dont celui-ci est implémenté.

- Problème : Comment fournir un objet préservant l'encapsulation d'une structure de donnée tout en permettant de la parcourir ?
- Solution : Un itérateur fournit les méthodes :
  - `hasNext` indiquant si il existe un élément qui le suit dans la structure de donnée ;
  - `next` renvoyant un itérateur vers l'élément suivant ;
  - `getElement` renvoyant l'élément pointé par l'itérateur.

  Une structure de donnée implémente une méthode `iterator` renvoyant une itérateur sur le premier de ses éléments.

Médiateur (Mediator) – Description

- Exemple de situation :

  Une bibliothèque servant à construire des interfaces graphiques proposent en général des centaines de widgets différents. Ces widgets communiquent entre eux de façon complexe : asynchrone/synchrone, par broadcast, par groupes . . .

- Problème : Comment découpler l'implémentation d'un *widget* de la façon dont il communique avec les autres ?
- Solution : Implémenter un mécanisme de communication où les messages sont des objets de première classe et où les *widgets* ne sont pas référencés explicitement mais par des identifiants symboliques.

Médiateur (Mediator) – Critique

- La souplesse des messages implémentés par des données est claire : on peut reconfigurer, tracer, contrôler, les interactions entre les objets.
- Cependant, remplacer le mécanisme des méthodes par un tel système n'est pas forcément une bonne idée car on perd alors les garanties fournies par le typage (statique).

Memento (Memento)

- Exemple de situation :

  Dans un contexte d'accès concurrent à une ressource, il est parfois utile d'être optimiste : on lance la requête d'accès de façon asynchrone, on continue la partie du calcul qui ne dépend pas de l'accès et on termine le calcul une fois que le résultat de la requête est arrivé. Mais que faire lorsque la requête a échoué ? On doit se replacer dans l'état initial et faire comme si on n'avait pas continuer ses calculs. . .

- Problème : Comment restaurer l'état d'un objet ?
- Solution : Une classe Memento sert à représenter l'état d'un objet (en JAVA, un objet de type Object fait très bien l'affaire). Une classe CareTaker sert à stocker ses états. Elle est aggrégée par l'objet qu'on veut être capable de restaurer. Avant une section critique, l'objet se stocke sous la forme d'un Memento dans son attribut de type CareTaker et se restore si la requête a échoué.

Observateur (Observer)

- Exemple de situation :

  Imaginons l'application d'une séquence de filtres dans un logiciel de retouche d'images. Pour tenir l'utilisateur au courant de la progression du calcul, plusieurs objets s'affichent dans son interface : une barre de progression, un pourcentage dans la barre de titre, une icône sur son bureau . . . Le nombre de ces indicateurs n'est a priori pas limité. Comment s'assurer qu'ils seront tous mis au courant de l'avancement du processus ?

- Problème : Comment modéliser l'observation de l'état d'un objet par un ensemble dynamiquement défini d'autres objets ?
- Solution : L'objet observé offre une méthode `attach` permettant à un observateur de s'enregistrer pour être informé. Pour cela, celui-ci doit proposer une méthode `notify`. A chaque fois qu'il est mis à jour, l'objet observé itère sur l'ensemble de ses observateurs et les notifie de sa modification.

État (State)

- Exemple de situation :

  Revenons maintenant sur le problème des figures géométriques atomiques et composites. J'ai décidé d'implémenter le comportement suivant : lorsque le message `add` est envoyé à une figure atomique, elle devient tout simplement composite !

- Problème : Comment donner l'illusion que le type d'un objet a changé en fonction de son état ?
- Solution : Une forme stocke un attribut qui peut être une forme atomique ou une forme composite. Les formes atomiques et composites ne dérivent plus de la classe *Shape* mais d'une nouvelle classe abstraite *RawShape*. Lorsque la méthode `add` est appelée et que l'attribut est une forme atomique, on le remplace par une nouvelle forme composite la contenant ainsi que la nouvelle forme à rajouter.

Stratégie (Strategy) – Description

- Exemple de situation :

  Une intelligence artificielle s'appuie sur une heuristique, c'est-à-dire une évaluation approximative de la valeur d'une situation pour le joueur. Il existe de nombreuses heuristiques possibles pour un jeu d'échec. Est-ce à dire qu'il faille créer une sous-classe par heuristique ?

- Problème : Comment paramétrer une classe par un comportement ?
- Solution : Un comportement est implémentée par une classe. Le comportement est donné en argument au constructeur de la classe à paramétrer. Remarquez encore une fois qu'en présence de fonction de première classe, ce patron est trivialement implémentable.

Patron de méthode (Template Method) – Description

- Exemple de situation : Je souhaite implémenter un objet effectuant le tri d'un tableau d'objet paramétré par la relation de comparaison entre les objets stockés.
- Problème : Comment paramétrer une méthode par une fonction ?
- Solution : Encore une fois, il s'agit de programmation d'ordre supérieur, on utilise un objet pour représenter la fonction pris en paramètre.

Visiteur (Visitor) – Description

- Exemple de situation : Je veux traduire en une multitude de format un document composé par des instances de ma hiérarchie de formes. Pour des raisons de modularité, il est hors de question d'écrire ce code dans chacune des classes !
- Problème : Comment étendre fonctionnellement une hiérarchie existante ?
- Solution : Un visiteur implémente une méthode par cas de la hiérarchie. Chaque sous-classe implémente une méthode $visit$ qui appelle la méthode du visiteur correspond à son cas.

Les anti-patrons de conception

Quelques anti-patrons (source : Wikipedia)

Il est aussi important de savoir ce qu'il ne faut pas faire !

- Abstraction inverse
- Action à distance
- Ancre de bateau
- Attente active
- Interblocages et famine
- Erreur de copier/coller
- Plat de lasagnes
- Réinventer la roue carrée
- Coulée de lave
- Surcharge des interfaces
- L'objet divin