Génie Logiciel Avancé
Cours 5 — Conception d'un système d'objets

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2014-2015

URL http://upsilon.cc/zack/teaching/1415/gla/
Copyright © 2011-2013 Stefano Zacchiroli
© 2010 Yann Régis-Gianas
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

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.

Exemple

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).

Exemple

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."

Références

~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)

Gamma, Helm, Johnson, Vlissides
Design Patterns : Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1995.

auteurs AKA “*Gang of Four*” (GoF)

Classification des patrons de conception

GoF ont explicité trois grandes classes de patrons dans leur livre, chacune spécialisée dans :

- création d'objets (*creational patterns*)
- structure des relations entre objets (*structural patterns*)
- comportement des objets (*behavioral patterns*)

Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
composed using
Command
adding responsibilities to objects
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
Interpreter
adding operations
Chain of Responsibility
sharing strategies
Strategy
sharing states
sharing terminal symbols
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
configure factory dynamically
Factory Method
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Héritage ou composition ?

- on parle de *white-box reuse* quand on réutilise du code à travers l'héritage
  - est dangereux, car les sous-classes ont droit à briser l'encapsulation
  - n'est pas toujours possible (p. ex. code propriétaire dont la documentation interne n'est pas disponible)
- on parle de *black-box reuse* quand on réutilise du code à travers la composition d'objet (association/composition/agrégation)
  - est dynamique, donc le compilateur et le langage de programmation n'aident pas à imposer des contraints de bonne forme
    - est plus difficile raisonner sur un logiciel
  - les objets parlent entre eux seulement à travers interfaces, donc l'encapsulation est sauvegardé

Principe général

Favor object composition over class inheritance.

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

Fabrique (*Factory Method*) — description

Exemple 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 (p. ex. une table de hachage tant que sa taille n'excède pas quelques Mo, une base de données sinon). Les prochaines versions de mon composant intégreront de nouvelles structures de données.

Problème Comment rendre la choix de la structure de données indépendant de l'acte d'instanciation ?

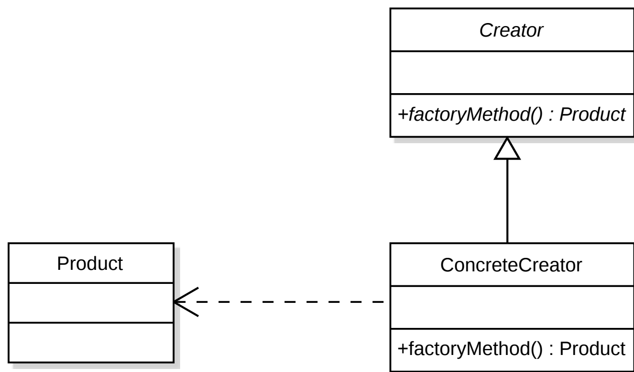
Fabrique (*Factory Method*) — description

Exemple 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 (p. ex. une table de hachage tant que sa taille n'excède pas quelques Mo, une base de données sinon). Les prochaines versions de mon composant intégreront de nouvelles structures de données.

Problème Comment rendre la choix de la structure de données indépendant de l'acte d'instanciation ?

Solution On fournit une classe *DictionaryFactory* qui répond au message *CreateDictionary (n)*—le *factory method*—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*) — UML



http://en.wikipedia.org/wiki/File:FactoryMethod.svg

On peut donc créer *Product* sans lier le code client à un choix spécifique de classe qui l'implémente.

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 chooses 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*) — discussion

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 (*i.e.* 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*).

```
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 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) ?

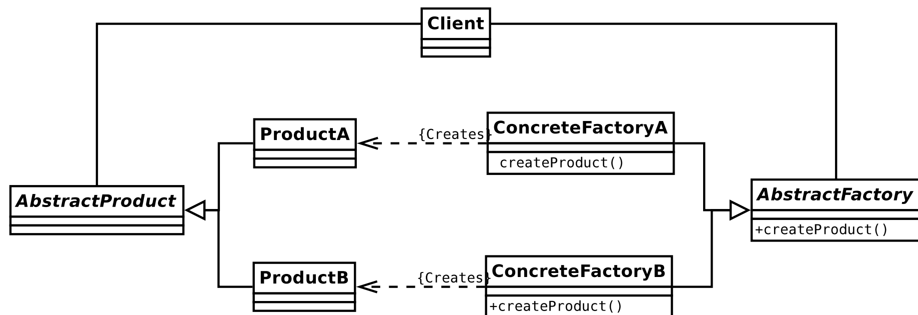
Fabrique abstraite (*Abstract Factory*) — description

Exemple 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) ?

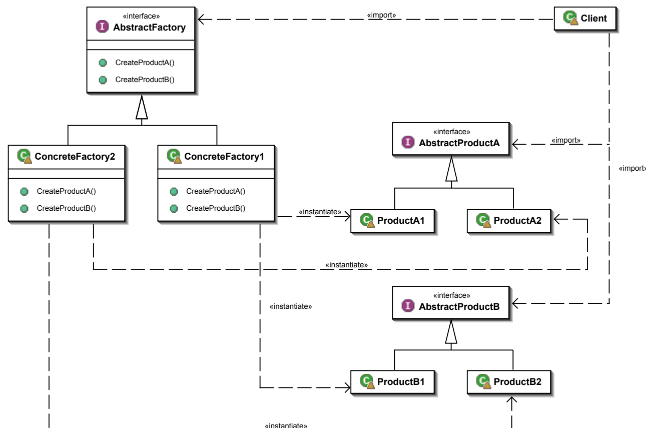
Solution Appliquer le motif de Fabrique à la Fabrique elle-même ! La classe *DictionaryFactory* devient abstraite. On crée une classe *PersistantDictionaryFactory* et une classe *NonPersistantDictionaryFactory*. On peut ensuite définir une classe *DictionaryFactoryCreator* qui choisit d'instancier la bonne fabrique en fonction d'un booléen.

Fabrique abstraite (*Abstract Factory*) — UML



http://thisblog.runsfreesoftware.com/?q=Abstract+Factory+Design+Pattern+UML+Class+Diagram

Fabrique abstraite (*Abstract Factory*) — UML (cont.)



http://upload.wikimedia.org/wikipedia/commons/9/9d/Abstract_factory_UML.svg

En plus, nous pouvons utiliser une fabrique abstraite pour créer *familles* d'objet, avec des garanties de cohérence entre eux (p.ex. widgets des toolkits graphiques différents).

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

Exemple [1] Après leur migration à la version persistante de mon composant, les utilisateurs se plaignent de la présence de fuites de mémoire.

Problème Lors que 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.

1. ce patron s'applique seulement en absence de ramasse-miettes

Pointeur intelligent (*Smart Pointer*) — description

Exemple [1] Après leur migration à la version persistante de mon composant, les utilisateurs se plaignent de la présence de fuites de mémoire.

Problème Lors que 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).

1. ce patron s'applique seulement en absence de ramasse-miettes

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 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 ?

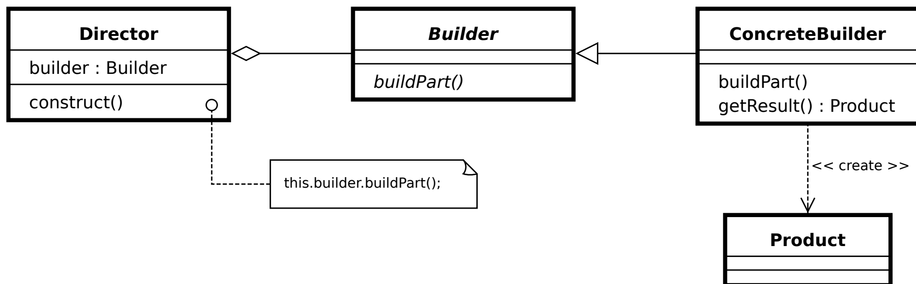
Monteur (*Builder*) — description

Exemple 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.

Monteur (*Builder*) — UML



http://en.wikipedia.org/wiki/File:Builder_UML_class_diagram.svg

On peut donc utiliser différents implémentations concrètes pour les différentes phases (abstraites) de la construction du produit.

Prototype (*Prototype*) — description

Exemple
: 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 ?

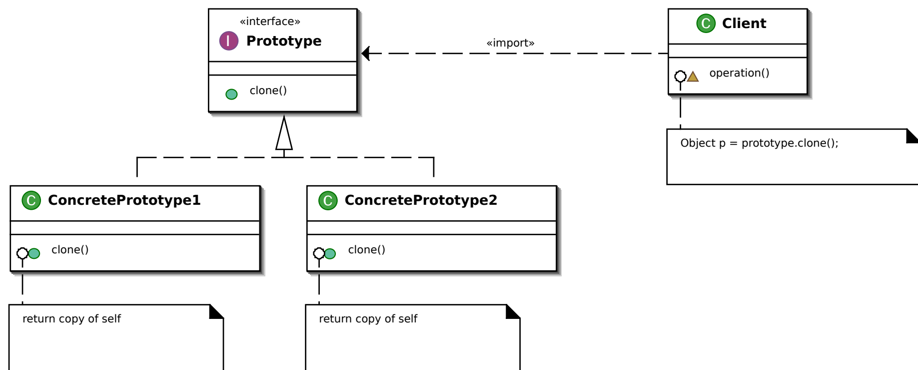
Prototype (*Prototype*) — description

Exemple 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*) — UML



http://upload.wikimedia.org/wikipedia/commons/1/14/Prototype_UML.svg

Prototype (*Prototype*) — en Scala

```
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 (cont.)

```
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).

Exemple

```
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

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 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 exécution donnée, il ne doit exister qu'une unique instance de la classe *Environment*.

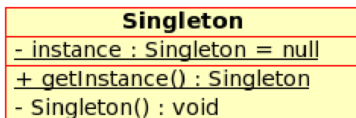
Singleton (*Singleton*) — description

Exemple 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 exécution 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*) — UML



http://upload.wikimedia.org/wikipedia/commons/b/bd/Singleton.png

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*).
- Problèmes de : concurrence, unit testing, boucles d'instanciation, héritage, etc.
- 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://web.archive.org/web/20080915000000*/http://www.codingwithoutcomments.com/2008/10/08/singleton-i-love-you-but-youre-bringing-me-down/`

Réserve d'objets (*Object Pool*) — description

Exemple Un serveur peut traiter un nombre N de clients simultanés. Chaque client se voit allouer des ressources dont l'allocation est longue (une connexion 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.

Référence : `http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf`

Réserve d'objets (*Object Pool*) — description

Exemple Un serveur peut traiter un nombre N de clients simultanés. Chaque client se voit allouer des ressources dont l'allocation est longue (une connexion 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.

Référence : `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 ?

?

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

Adaptateur (*Adapter*) — description

Exemple 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.

Adaptateur (*Adapter*) — description

Exemple 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

```
// image manipulation library
public interface ColorImage2D {
    public int   get_width  ();
    public int   get_height ();
    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) {}
}

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

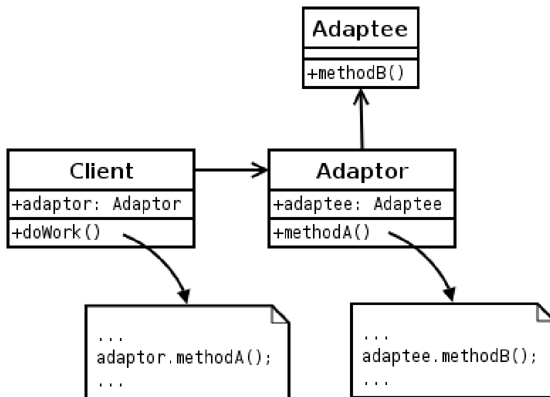
Adaptateur (*Adapter*) — en Java

```
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  () { return width; }
    public int   get_height () { 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); }
}
```

Adaptateur (*Adapter*) — UML



http://en.wikipedia.org/wiki/File:ObjectAdapter.png

Figure: *object adapter* (avec composition)

Adaptateur (*Adapter*) — UML (cont.)

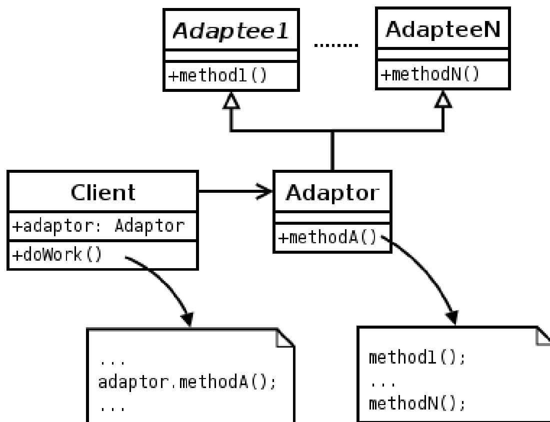


Figure: *class adapter* (avec héritage)

Pont (*Bridge*) — description

Exemple 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.

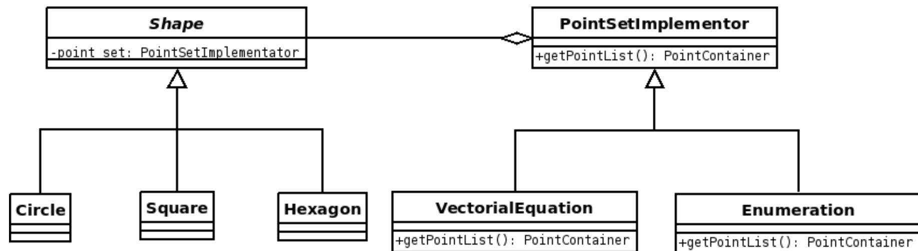
Pont (*Bridge*) — description

Exemple
: 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*, . . .).

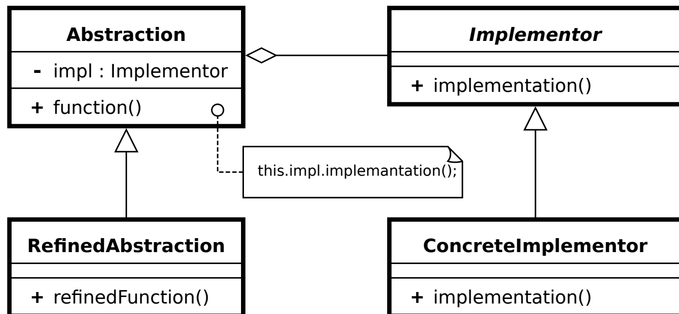
Problème
: Pour éviter la multiplication des classes et modulariser le système, il est nécessaire de décorréler la hiérarchie des abstractions (les formes) de leurs implémentations (les ensembles de points) car elles peuvent varier de manière indépendante.

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 agrégée à toute forme.

Pont (*Bridge*) — UML



Pont (*Bridge*) — UML, cas général



http://en.wikipedia.org/wiki/File:Bridge_UML_class_diagram.svg

Pont (*Bridge*) — discussion

- les deux hiérarchies peuvent maintenant évoluer indépendemment
  - l'alternative est de mélanger dans une même hiérarchie des relations d'héritage liées à l'implémentation avec des relations liées à l'abstraction
- la structure ressemble Adapter, mais ici elle est nécessaire pour permettre l'indépendance des évolutions ; avec Adapteur elle est utilisée pour adapter un *objet* existant

Objet composite (*Composite*) — description

Exemple 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 ?

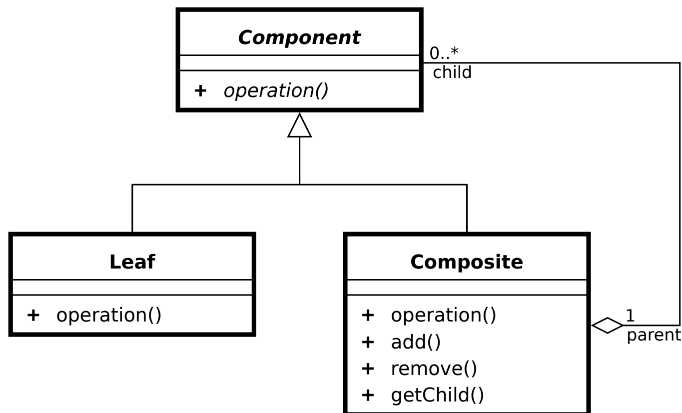
Objet composite (*Composite*) — description

Exemple 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 noeud 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*) — UML



http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_(fixed).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. . .

Objet composite *sûr* — en OCaml

```
class virtual shape = object    (* interface *)
  method virtual draw : unit -> unit
  method virtual as_composite : composite option
end
and circle = object              (* leaf *)
  inherit shape
  method draw () = (* ... *)
  method as_composite = None
end
and composite = object (self)  (* container *)
  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
(* ... why not using an algebraic data type alltogether? *)
```

Décorateur (*Decorator*) — description

Exemple 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 ?

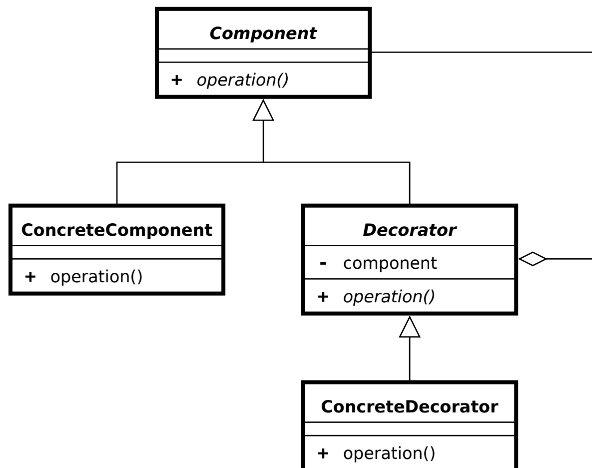
Décorateur (*Decorator*) — description

Exemple 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 agrège un composant particulier et implémente l'interface d'un composant en *délégant tous les messages correspondants à son instance agrège*. En héritant d'un Décorateur, on peut rajouter un comportement particulier, comme par exemple, la gestion des hyperliens.

Décorateur (*Decorator*) — UML



http://en.wikipedia.org/wiki/File:Decorator_UML_class_diagram.svg

Décorateur (*Decorator*) — discussion

- les implémentations de Decorator::operation() typiquement appellent component−>operation() (en ajoutant avant ou après des autres comportements)
- on peut décorer le même objet avec plusieurs décorateurs (au moins au niveau de types. . .)
- on peut changer des “décorations” dynamiquement, car le type du décorateur(s) est compatible avec le type du composants
  - beaucoup plus flexible que l’approche à la décoration basé sur l’héritage

Façade (*Facade*)

Exemple 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 ?

Façade (*Facade*)

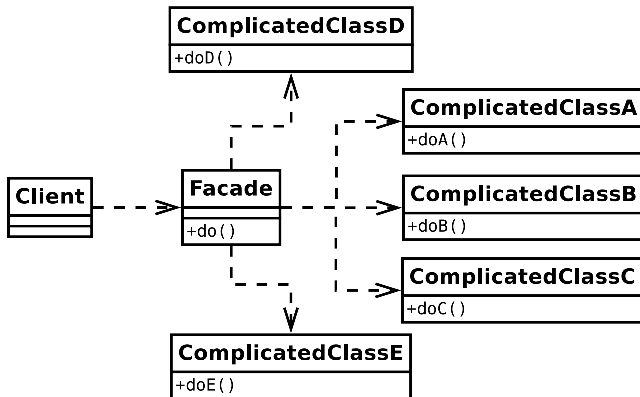
Exemple Pour traiter les données d'un programme de gestion de bibliothèque multimédia, je désire utiliser l'API de Postgresql, un gestionnaire de base de données très élaboré. Mon utilisation de la base de données est essentiellement un dictionnaire associant des noms de fichier et des descriptions à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

Problème Comment fournir une interface simple en utilisant un système complexe ?

Solution Le patron façade implémente seulement les méthodes utiles à mon dictionnaire en utilisant l'API de Postgresql. Il s'agit donc d'implémenter une couche d'abstraction.

- Dans le cas général, façade fourni une abstraction unique (simplifiée) sur plusieurs sous-système, sans empêcher de les utiliser directement (*make simple tasks simple, complex tasks possible*).

Façade (*Facade*) — UML, cas général



http://thisblog.runsfreesoftware.com/?q=Facade+Design+Pattern+UML+Class+Diagram

Poids-mouche (*Flyweight*) — description

Exemple
: 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 ?

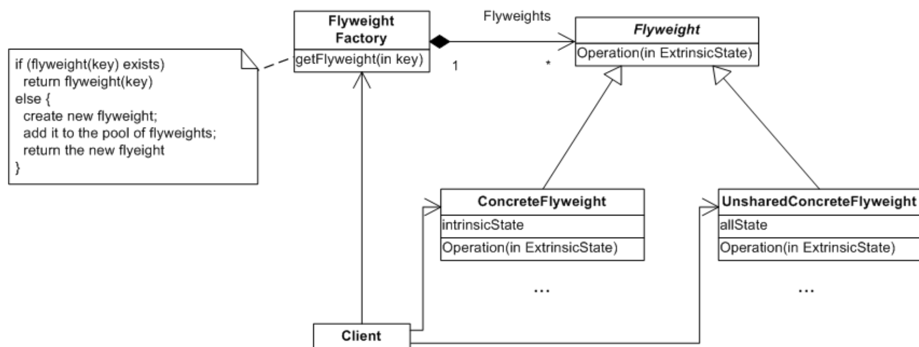
Poids-mouche (*Flyweight*) — description

Exemple 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 On coupe l'état des *occurrences* d'identifiants en état interne (*intrinsic state*) et externe (*extrinsic state*). Les identifiants sont créés par une fabrique. Cette dernière stocke une table associant un descripteur à un identifiant et a son état interne. Si l'identifiant n'est pas dans la table, le calcul concret est effectué, sinon on renvoie l'élément déjà calculé. L'état externe est passé à toutes utilisations.

Poids-mouche (*Flyweight*) — UML



http://www.lepus.org.uk/ref/companion/Flyweight.xml

Poids-mouche (*Flyweight*) — en OCaml

Dans les langages fonctionnels, on peut implémenter une partie de *flyweight* de façon générale pour toute fonction avec la memoization.

```
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 (c.f. *weak hashtables*).

Proxy (*Proxy*)

Exemple
: 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.

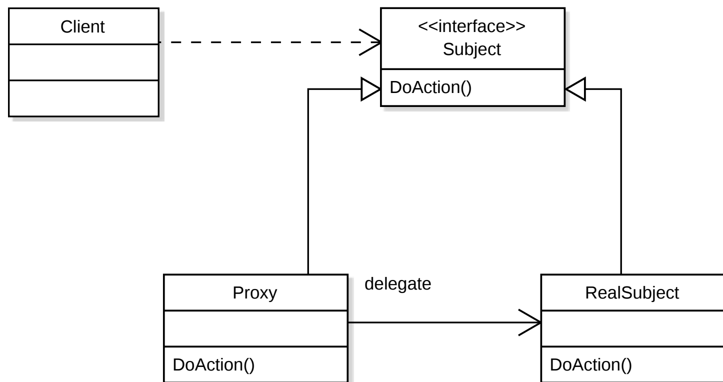
Proxy (*Proxy*)

Exemple 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.

Proxy (*Proxy*) — UML



http://en.wikipedia.org/wiki/File:Proxy_pattern_diagram.svg

Proxy (*Proxy*) — applicabilité

On peut imaginer beaucoup des types de proxy différents :

proxy virtuel
: proxy paresseux, comme dans l'exemple du GPS

remote proxy
: représentants "locales" d'objet accessible à distance (p. ex. RPC, RMI, Web Services, SOAP, etc.)

proxy de protection
: si un proxy est la seule moyenne pour accéder le sujet réel, le proxy peut être utilisé pour donne de garantie de contrôle d'accès

smart pointer
: (déjà vu) sont un cas particulier de proxy

memoization proxy
: ...

...
: ...

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

Chaîne de responsabilité (*Chain of responsibility*)

Exemple On veut intégrer un mécanisme de greffons (*plugins*) a un traitement de texte pour pouvoir rajouter dynamiquement la gestion de certains types de composants. (Ou encore : on veut ajouter un mécanisme d'aide en ligne du composant plus spécifique au composant plus générique.)

Problème Comment écrire une fois pour toute la fonction qui choisit le bon greffon à utiliser par observation du type du composant à traiter ?

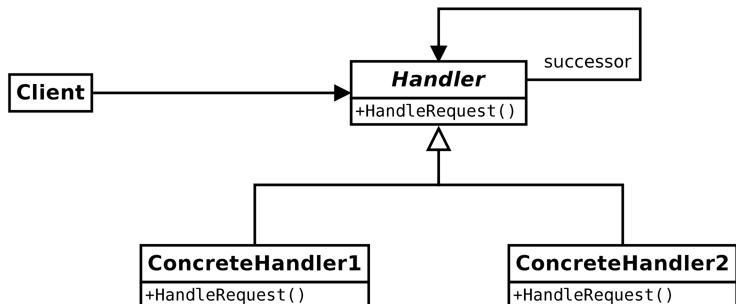
Chaîne de responsabilité (*Chain of responsibility*)

Exemple On veut intégrer un mécanisme de greffons (*plugins*) a un traitement de texte pour pouvoir rajouter dynamiquement la gestion de certains types de composants. (Ou encore : on veut ajouter un mécanisme d'aide en ligne du composant plus spécifique au composant plus générique.)

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 stockée par l'application. Lorsqu'un greffon ne sait pas traiter un message, il le transfère au suivant.

Chaîne de responsabilité (*Chain of responsibility*) — UML



Chaîne de responsabilité (*Chain of responsibility*) — discussion

- une chaîne de responsabilité diminue le *coupling* expéditeur/destinataire et donne à plusieurs intermédiaires—plus particulièrement à une chaîne de—la possibilité de prendre en compte le message
- la flexibilité en terme des responsabilités des objets augmente et peut changer au *runtime*
- le patron ne donne pas la garantie qu'un destinataire capable de gérer un message existe
  - il faut donc prévoir un destinataire *catch all*
- les requêtes peuvent être plus compliques que dans l'exemple montré
  - p. ex. contenir des paramètres ou être des objets complexes, comme dans le patron...

Commande (*Command*)

Exemple 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 ?

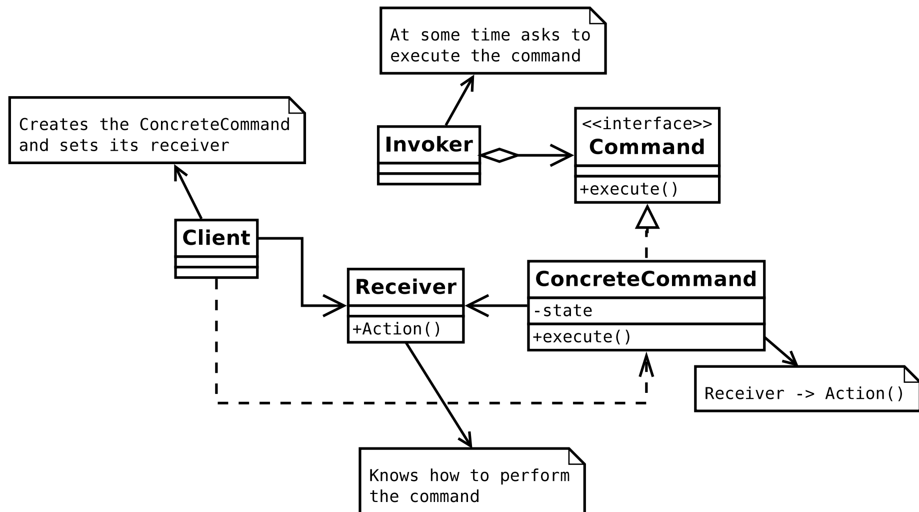
Commande (*Command*)

Exemple 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.

Commande (*Command*) — UML



basé sur `http://thisblog.runsfreesoftware.com/?q=node/75`

Commande (*Command*) — discussion

- utile pour paramétrer des objets avec des actions qu'il doivent exécuter, typique p. ex. dans les *toolkit* graphiques
- les objets qui encodent les actions peuvent avoir un cycle de vie diffèrent du cycle de vie de client, receiver, etc.
  - cas d'usage : *scheduling*
- l'état gardé par les objets command peut être utiliser pour supporter undo/redo (il faudra pour cela ajouter une méthode *unexecute*)
  - ou encore les objets command peuvent être utiliser pour implémenter des mécanismes de *journaling*
  - potentiellement avec l'aide des méthodes et attribues statiques sur leurs classes d'origine

Interpréteur (*Interpreter*) — description

Exemple Composer les filtres n'est pas suffisant, un utilisateur avancé doit pouvoir programmer ses propres filtres (visuellement ou textuellement).

Problème Comment représenter un langage (syntaxe et sémantique à la fois) à l'aide d'objets ?

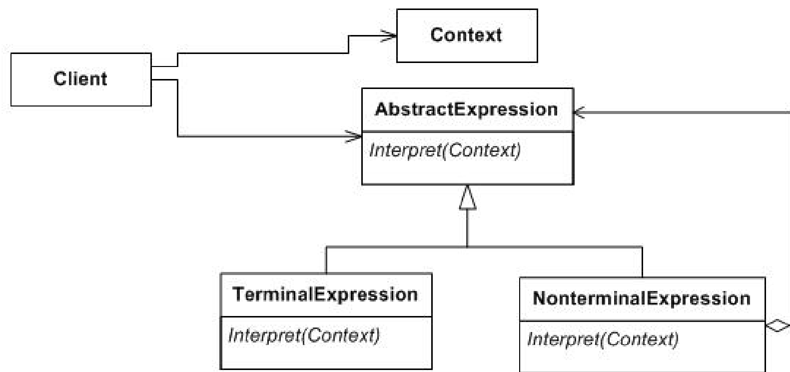
Interpréteur (*Interpreter*) — description

Exemple Composer les filtres n'est pas suffisant, un utilisateur avancé doit pouvoir programmer ses propres filtres (visuellement ou textuellement).

Problème Comment représenter un langage (syntaxe et sémantique à la fois) à l'aide d'objets ?

Solution Une classe abstraite *Expression* dont dérive les diverses constructions du langage.

Interpréteur (*Interpreter*) — UML



Interpréteur (*Interpreter*) — discussion

- il s'agit d'une généralisation du patron *Command*, pour cas plus complexes
- ce patron partage beaucoup des caractéristiques (et critiques, comme la sûreté de typage) avec le patron Composite
- on utilise le contexte pour stocker les données pas locales, nécessaire pour interpréter le langage
  - comme dans le cas du patron *Flyweight*

Itérateur (*Iterator*)

Exemple
: Revenons à notre bibliothèque de dictionnaires. Je veux 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 ?

Itérateur (*Iterator*)

Exemple Revenons à notre bibliothèque de dictionnaires. Je veux 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.

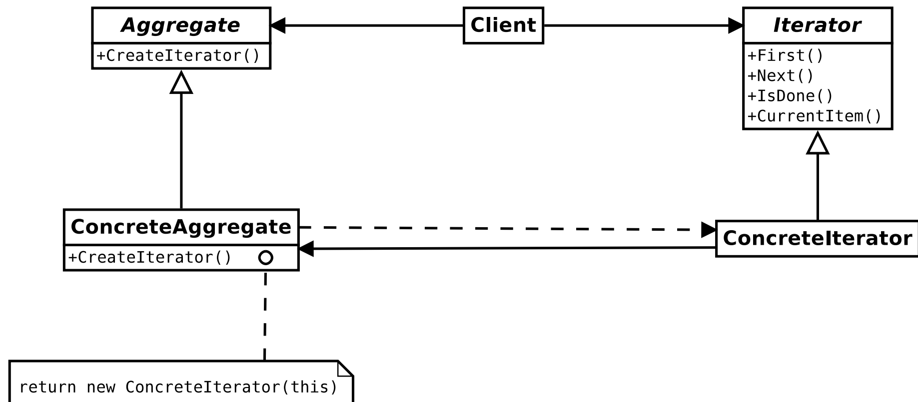
Problème Comment fournir un objet préservant l'encapsulation d'une structure de donnée tout en permettant de la parcourir ?

Solution Un itérateur fournit les méthodes (ou variantes de) :

- *CurrentItem* renvoyant l'élément courant.
- *Next* renvoyant un itérateur vers l'élément suivant ;
- *IsDone* indiquant si il existe un élément qui le suit dans la structure de donnée ;

Une structure de donnée implémente une méthode *iterator* renvoyant un nouveau itérateur sur ses éléments.

Itérateur (*Iterator*) — UML



Itérateur (*Iterator*) — discussion

- avantages :
  - l'interface de `Aggregate` reste simple
  - l'état de l'itération peut être indépendant de l'état de `Aggregate`
- très populaire dans le bibliothèque standard des langages des programmation de haut niveau
- une abondance des libertés dans la définition d'un itérateur et de sa sémantique :
  - itérateurs différents peuvent implémenter politique de visite différents (p. ex. *depth-first* vs *breadth-first*)
  - est l'itérateur robuste (par rapport aux changements du collection visitée) ?
  - l'interface change significativement cas par cas (p. ex. `First` dans notre exemple permet de recommencer l'itération)
  - qui est en contrôle de l'itération ?

    le client (comme dans notre exemple) : itérateur externe, stratégie *pull*

    l'itérateur : itérateur interne, stratégie *push* avec *callbacks*
  - ...

Médiateur (*Mediator*) — description

Exemple 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* complexe de la façon dont il communique avec les autres ?

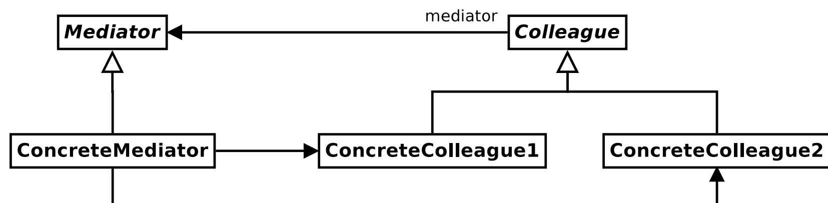
Médiateur (*Mediator*) — description

Exemple 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* complexe de la façon dont il communique avec les autres ?

Solution Ajouter un nouveau object médiateur et définir un nouveau protocole de communication avec lui. Chaque widget communique seulement avec le médiateur. Le médiateur s'occupe du routing des messages entre widgets. Comme conséquence, les widgets parlent seulement avec le médiateur et les interdépendances entre eux diminuent.

Médiateur (*Mediator*) — UML



Médiateur (*Mediator*) — critique

- Exemples typiques : MVC généralisées dans des widgets graphiques avec logiques de mise à jour très complexes
- Ce patron fourni une abstraction *protocol as an object*
- 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) des messages envoyés entre objets (sans médiation).

Memento (*Memento*)

Exemple 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 ?

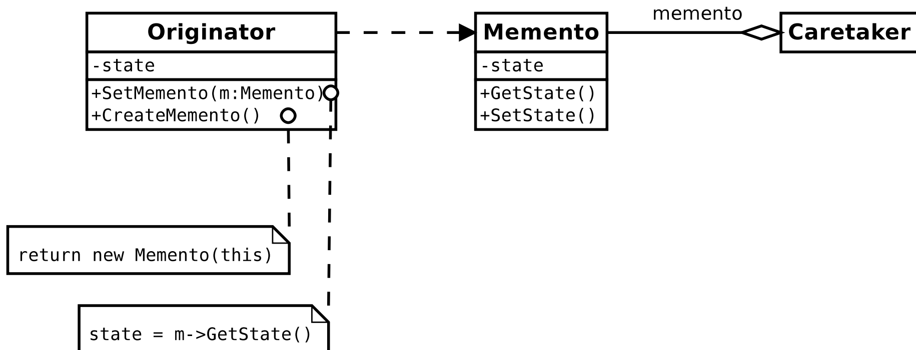
Memento (*Memento*)

Exemple 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 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.

Memento (*Memento*) — UML



Memento (*Memento*) — discussion

Quand on a besoin d'un memento ?

Memento (*Memento*) — discussion

Quand on a besoin d'un memento ?

Quand l'interface *publique* d'un objet n'est pas suffisante à extraire l'état qui nécessite sauvegarde.

Memento (*Memento*) — discussion

Quand on a besoin d'un memento ?

Quand l'interface *publique* d'un objet n'est pas suffisante à extraire l'état qui nécessite sauvegarde.

Un objet de type memento doit donc exposer *deux* interfaces des types différents :

interface restreinte : pour le CareTaker, qui ne doit *pas* pouvoir accéder à l'état de Originator

interface ouverte : pour Originator qui permet l'accès complet au memento pour restaurer l'état original

Pour obtenir cela, plusieurs *trade-off* sont à évaluer :

- *friends* en C++
- visibilité restreinte à niveau *package*
- *casting* (p. ex. vers *Object* en Java) plus *downcasting* (qui est pas sûr. . .)

Observateur (*Observer*)

Exemple 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 ?

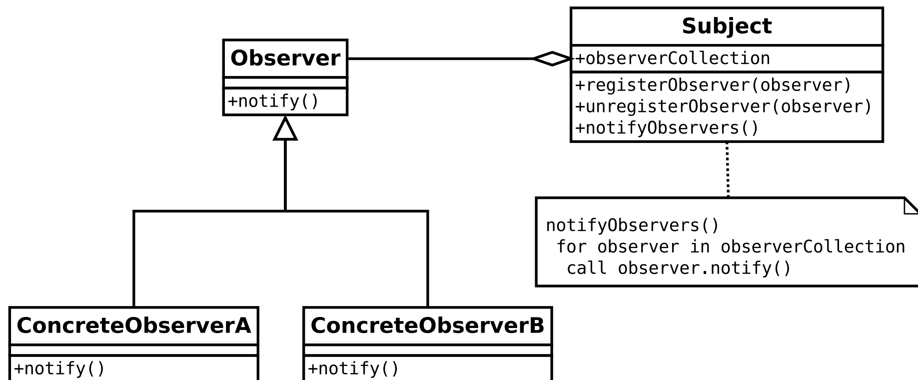
Observateur (*Observer*)

Exemple 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 *registerObserver* permettant à un observateur de s'enregistrer pour être informé. Pour cela, celui-ci doit proposer une méthode *notifyObservers*. 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.

Observateur (*Observer*) — UML



http://en.wikipedia.org/wiki/File:Observer.svg

Observateur (*Observer*) — discussion

- le diagramme UML ne montre pas comment les observateurs accèdent au subject. Typiquement ils gardent une référence explicite au subject
- ce patron decouple le subject de la nécessite de connaître les classes spécifiques des observateurs : la seule abstraction nécessaire est le type des observateurs
- le seule paradigme de communication supporté (nativement) par ce patron est la communication en *broadcast* ; les communications 1-1 ne sont pas supportées
- faites attention à (ne pas) *changer* le subject dans l'exécution de notify : ça peut générer des autres notifications. . .
- le protocole native de ce patron ne dit pas *quoi* a changé, mais seulement que quelque chose à change, donc les observateurs ne peuvent pas collecter seulement ce partie (des protocoles plus complexes sont possible)

État (*State*)

Exemple 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 ?

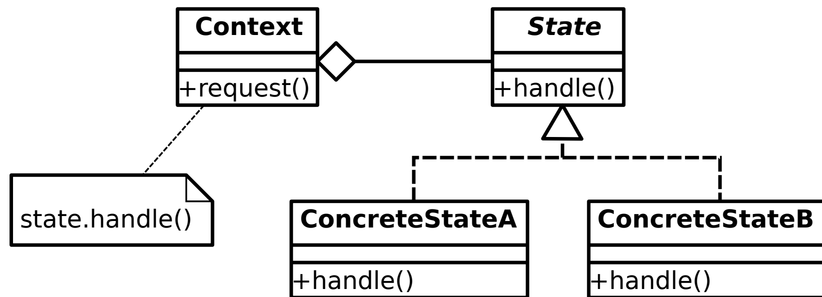
État (*State*)

Exemple 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 (son état) 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*. Le comportement dépend de l'état : 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.

État (*State*) — UML



http://en.wikipedia.org/wiki/File:State_Design_Pattern_UML_Class_Diagram.svg

État (*State*) — discussion

- tous les comportements relatives à un état sont localisés dans un seule objet, correspondant à l'état
- avec ce parton les changements d'état deviennent explicites dans le code : l'attribut state.handle doit changer
- les objets de type état peuvent être partager entre contextes différentes
  - éventuellement en s'appuyant sur la distinction entre état interne et externe (*intrinsic/extrinsic state*) (voir le patron *flyweight*)

Stratégie (*Strategy*)

Exemple Une intelligence artificielle pour un jeu d'échec 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. Est-ce à dire qu'il faille créer une sous-classe par heuristique ?

Problème Comment paramétrer une classe par un comportement ?

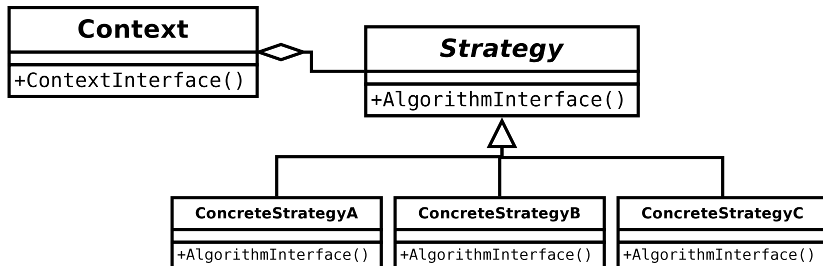
Stratégie (*Strategy*)

Exemple
: Une intelligence artificielle pour un jeu d'échec 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. 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 (ou aux autres méthodes quand nécessaire) de la classe à paramétrer. Toutes utilisations de l'heuristique passe à travers la classe comportement.

Stratégie (*Strategy*) — UML



http://thisblog.runsfreesoftware.com/?q=node/64

Stratégie (*Strategy*) — discussion

- l'interface entre *Context* et *Strategy* peut être arbitrairement complexe et peut en général échanger information dans les deux directions
  - l'interface doit permettre d'échanger tout information nécessaire à exécuter l'algorithme implémenté par *Strategy*
- des classes des algorithmes similaires peuvent être représentées avec une hiérarchie des stratégies
- ce patron offre une alternative à plusieurs if. . . then. . . else pour paramétrer un comportement
  - en générale, trop de if. . . then. . . else peut indiquer l'absence d'une hiérarchie des classes appropriée
- remarquez encore une fois qu'en présence de fonction de première classe, ce patron est trivialement implémentable.

Patron de méthode (*Template Method*)

Exemple Je souhaite implémenter un algorithme (p. ex. de tri) paramétré par un ensemble d'opération externes, dont existent plusieurs implémentation possible.
Dans le cas générale je souhaite fournir une fonctionnalité complexe qui contient plusieurs *trous*.

Problème Comment paramétrer une méthode par plusieurs fonctionnalités externes ?

Patron de méthode (*Template Method*)

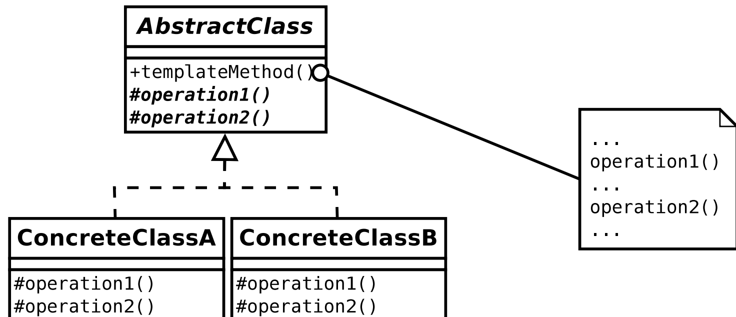
Exemple Je souhaite implémenter un algorithme (p. ex. de tri) paramétré par un ensemble d'opération externes, dont existent plusieurs implémentation possible.
Dans le cas générale je souhaite fournir une fonctionnalité complexe qui contient plusieurs *trous*.

Problème Comment paramétrer une méthode par plusieurs fonctionnalités externes ?

Solution J'implémente ma fonctionnalité dans une classe qui appelle des méthodes abstraites. Plusieurs concretisations de la même classe vont fournir instances de la fonctionnalité abstraite ou les trous sont remplis par des méthodes concrètes.

Encore une fois, il s'agit de programmation d'ordre supérieur, on utilise un objet pour représenter la fonction pris en paramètre.

Patron de méthode (*Template Method*) — UML



basé sur http://thisblog.runsfreesoftware.com/?q=node/77

Patron de méthode (*Template Method*) — discussion

- comme dans le cas du patron stratégie, on peut organiser des familles d'implementations des trous avec une hiérarchie de classes
- ce patron supporte aussi des mécanismes de *hooks*, i.e. de points spécifiques dans l'algorithme, où l'utilisateur peux ajouter des fonctionnalités
  - dans ce cas on n'utilise pas des méthodes abstraites, mais on utilise de méthodes concrètes dont les implementations par default ne font rien. Les classes concrètes font *override* des implementations par defaults
- il est important minimiser le numéro de méthodes abstraites, pour ne pas rendre trop difficile l'utilisation du code déjà implanté
- les méthodes abstraites deviennent un *API* pour les clients, qui doit être maintenu comme d'habitude

Visiteur (*Visitor*)

Exemple 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 ?

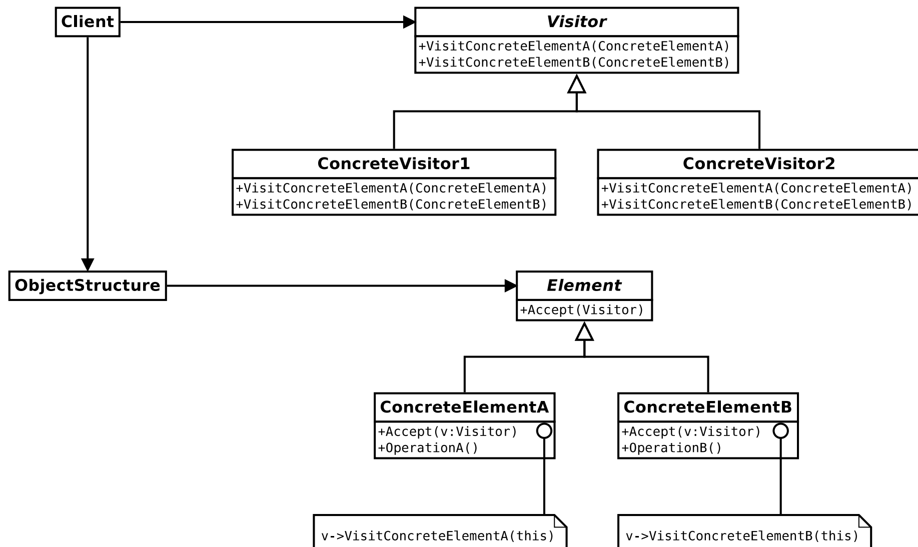
Visiteur (*Visitor*)

Exemple
: 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 de la hiérarchie implémente une méthode *visit* qui appelle la méthode du visiteur correspond à son cas.

Visiteur (*Visitor*) — UML



Visiteur (*Visitor*) — discussion

- prérequis : la hiérarchie de classe originelle est plus stable que l'*ensemble* d'opérations à effectuer sur elle
  - Ajouter des nouvelles opérations est simple et possible sans toucher la hiérarchie de classes originelle
  - Changer la hiérarchie de classe originelle devient plus difficile, car le type de visiteur doit changer avec conséquences sur tous les visiteurs existants ! (heureusement, le typage statique nous aide)
- chaque visiteur concret regroupe ensemble opérations liées entres eux et les sépare des opérations indépendantes
- différence importante avec le patron iterator : un iterateur ne peut pas iterer sur des objets de type différents (il doivent tous avoir la même interface) ; un visiteur n'as pas la même restriction (pourquoi ?)

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

Quelques anti-patrons

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

- Abstraction inverse
- Action à distance
- Ancre de bateau
- Attente active
- *Cargo cult programming*
- Coulée de lave
- Erreur de copier/coller
- *Not Invented Here*
- Interblocages et famine
- Objet divin
- Programmation spaghetti
- Réinventer la roue carrée
- Surcharge des interfaces
- ...

Très connus dans le *folklore* de langages de programmation à objets. On en voit seulement quelques exemples. Pour un index à jour, voir `http://en.wikipedia.org/wiki/Anti-pattern`.

L’objet divin

Concevoir un logiciel à objets avec des objets qui font trop de choses. C’est contraire au principe basique de *separation of concerns*.

Coulée de lave

Garder du vieux code (non optimale, plein de bugs, etc.) car l'effacer est trop coûteuse. (Application de *premature optimization is the root of all evil* au secteur financier de la génie logiciel.)

Abstraction inverse

Le client d'une interface a besoin d'une fonctionnalité implémenté par l'objet mais caché.

Résultat : le client re-implémente la même fonctionnalité à l'externe (en utilisant l'interface publique). Le code complet utilise *deux* implémentation de la même fonctionnalité à la fois : une interne et une externe.

Not Invented Here

Ne pas vouloir réutiliser une solution logiciel existante, seulement car il n'a pas était développée par les utilisateurs finals.

Cargo cult programming

Utiliser beaucoup des patrons, framework, outils, seulement car “il faut bien le faire” (est cool, on se sent plus cool à l’utiliser, etc.), sans comprendre les implications—parfois négatives—des ces patrons.

Sommaire

1. Patron de conception à objets
2. Création (*Creational Patterns*)
3. Structure (*Structural Patterns*)
4. Comportement (*Behavioral Patterns*)
5. Les anti-patrons de conception (*Anti-patterns*)
6. Synthèse

Synthèse

- réutiliser (et adapter !) des procédé d'organisation qui ont fonctionné dans le passé est une bonne démarche pour obtenir une bonne conception à objets
- un catalogue très connu des patrons de conceptions est celui du GoF, qui propose trois types des patrons :
  - *creational patterns*
  - *structural patterns*
  - *behavioral patterns*