Les design patterns

Design Pattern

- Objectifs
  - Comprendre les bases de la philosophie des « formes de conception »
  - Connaître le vocabulaire spécifique
  - Connaître quelques « patterns »
  - Concevoir objet différemment

Design Pattern

En architecture (Christopher Alexander)

- Description d'un problème rémanent et de sa solution
- Solution pouvant être utilisée des millions de fois sans être deux fois identique
- Forme de conception, pattern, modèle, patron de conception
- Mur, porte, fenêtre <--> objet, interface, classe

Il existe aussi des « anti-patterns »

Ce que c ’est

- Description d ’une solution classique à un problème récurent
- Décrit une partie de la solution…
- Avec des relations avec le système et les autres parties...
- C ’est une technique d ’architecture logicielle

Ce que ce n ’est pas

- Une brique
  - Un pattern dépend de son environnement
- Une règle
  - Un pattern ne peut pas s ’appliquer mécaniquement
- Une méthode
  - Ne guide pas une prise de décision ; un pattern est *la* décision prise
- Nouveau
  - Lao-Tzu (-550) travaillait déjà sur les patterns...

« Computer scientists think they have discovered the world »
Anonymous

Avantages

- Un vocabulaire commun
- Capitalisation de l ’expérience

- Un niveau d ’abstraction plus élevé qui permet d ’élaborer des constructions logicielles de meilleure qualité
- Réduire la complexité
- Guide/catalogue de solutions

Inconvénients

- Effort de synthèse ; reconnaître, abstraire…

- Apprentissage, expérience

- Les patterns « se dissolvent » en étant utilisés

- Nombreux…
  - lesquels sont identiques ?
  - De niveaux différents … des patterns s ’appuient sur d ’autres...

Description d'une forme : langage de pattern

- nom : augmente le vocabulaire, réifie une idée de solution, permet de mieux communiquer.
- problème : quand appliquer la forme, le contexte...
- solution : les éléments de la solution, leurs relations, responsabilités, collaborations. Pas de manière précise, mais suggestives...
- conséquences : résultats et compromis issus de l'application de la forme

Exemple

• Nom	Salle d 'attente
• Problème	On doit attendre
• Solution	Toujours relaxante et pas confinée
• Conséquences	Attente active ou passive ? Durée de l 'attente ? Distraction ?
• Exemples	Aéroport, dentiste, ...

Forme et langage

- Une forme est indépendante du langage

  (plutôt orienté-objet, mais pas exclusivement, cf. Patterns dans Minix)

- Mais certaines constructions syntaxiques ou propriétés du langage rendent inutile ou "naturelle" l'utilisation de telle ou telle forme

  (ex : multi-methode simplifie les visiteurs)

Interactions Formes-langages

- Influence des langages sur les Patterns
  - des langages implantent des formes de bas niveau
  - quelques formes utilisent des concepts spécifiques à un langage
  - quelques formes ne sont pas indépendantes des langages
  - certains langages forcent à tordre des formes compliquées lors de l ’implantation
- Influence des Patterns sur les langages
  - Les Formes capitalisent l ’état de réflexion courant sur les pratiques de programmation.

Gamma, Helm, Johnson, Vlissides "Design Patterns"

- Nom et classification
- Intention
- Autres noms connus
- Motivation (scénario)
- Applicabilité
- Structure (OMT)
- Participants (classes…)
- Collaborations
- Conséquences
- Implantation
- Exemple de code
- Usages connus
- Formes associées

Gang4

Classification

	Créateurs	Structuraux	Comportementaux
Class	Factory Method	Adapter(class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter(objet) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Respons. Command Iterator Mediator Memento Observer State Strategy Visitor

Application des formes lors de la conception

- Trouver les bons objets
- Bien choisir la granularité des objets
- Spécifier les interfaces des objets
- Spécifier l'implantation des objets
- Mieux réutiliser
  - héritage vs composition
  - délégation
- Compiled-Time vs Run-Time Structures
- Concevoir pour l'évolution

Mais d'abord, le catalogue !

- Créational Patterns
- Structural Patterns
- Behavioural Patterns

Remarque

- La référence "Design Patterns" décrit les formes par des diagrammes OMT...
  - Comme dans la suite

MAIS

- L'héritage et les objets ne sont pas nécessaires

Creational Patterns

Formes de création :

- Abstraire le processus d'instanciation.
- Rendre indépendant de la façon dont les objets sont créés, composés, assemblés, représentés.
- Encapsuler la connaissance de la classe concrète qui instancie.
- Cacher ce qui est créé, qui crée, comment et quand.

Principes

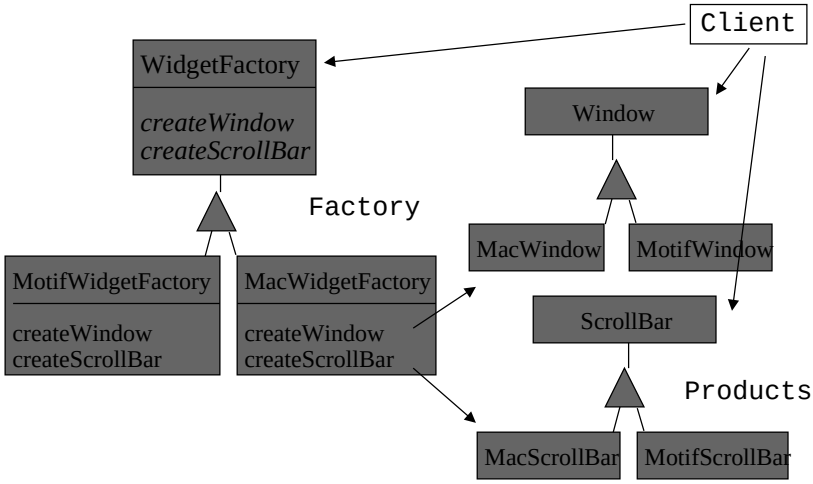
- AbstractFactory ; on passe un paramètre à la création qui définit ce qu'on va créer
- Builder ; on passe en paramètre un objet qui sait construire l'objet à partir d'une description
- FactoryMethod ; la classe sollicité appelle des méthode abstraites ...il suffit de sous-classer
- Prototype ; des prototypes variés existent qui sont copiés et assemblés
- Singleton ; unique instance

Utilisation

On utilise l'AbstractFactory lorsque :

- un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
- un système repose sur un produit d'une famille de produits
- une famille de produits doivent être utilisés ensemble, pour renforcer cette contrainte
- on veut définir une interface unique à une famille de produits concrets

Abstract Factory

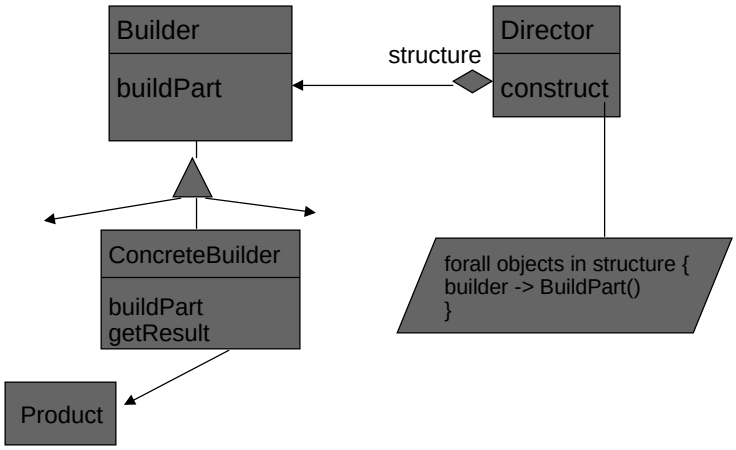


Utilisation

On utilise le Builder lorsque :

- l'algorithme pour créer un objet doit être indépendant des parties qui le compose et de la façon de les assembler
- le processus de construction permet différentes représentations de l'objet construit

Builder

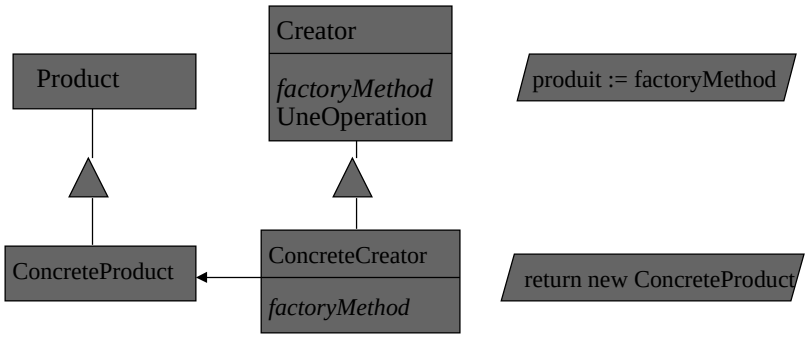


Utilisation

On utilise le FactoryMethod lorsque :

- une classe ne peut anticiper la classe de l'objet qu'elle doit construire
- une classe délègue la responsabilité de la création à ses sous-classes, tout en concentrant l'interface dans une classe unique

FactoryMethod

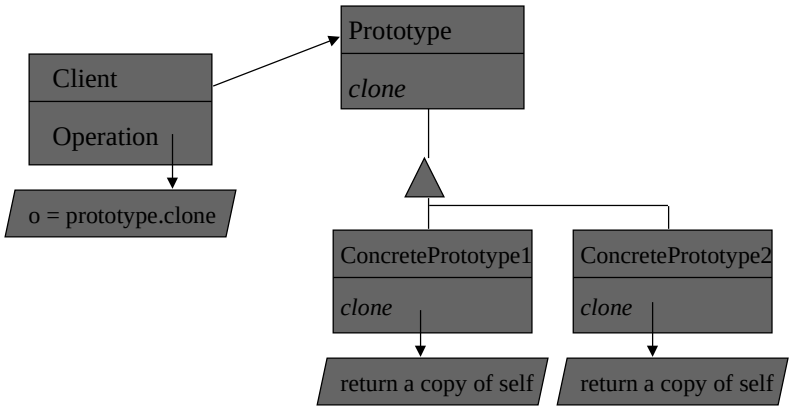


Utilisation

On utilise le Prototype lorsque :

- un système doit être indépendant de la façon dont ses produits sont créés, assemblés, représentés
- quand la classe n'est connue qu'à l'exécution
- pour éviter une hiérarchie de Factory parallèle à une hiérarchie de produits

Prototype

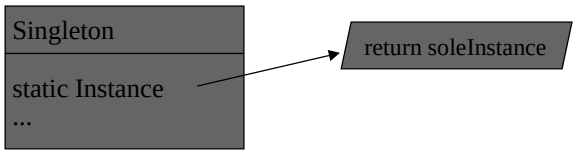


Utilisation

On utilise le Singleton lorsque :

- il n'y a qu'une unique instance d'une classe et qu'elle doit être accessible de manière connue
- une instance unique peut être sous-classée et que les clients peuvent référencer cette extension sans avoir à modifier leur code

Singleton



Structural Patterns

Formes de structure :

- Comment les objets sont assemblés
- Les patterns sont complémentaires les uns des autres

Principes

- Adapter ; rendre un objet conformant à un autre
- Bridge ; pour lier une abstraction à une implantation
- Composite ; basé sur des objets primitifs et composants
- Decorator ; ajoute des services à un objet
- Facade ; cache une structure complexe
- Flyweight ; petits objets destinés à être partagés
- Proxy ; un objet en masque un autre

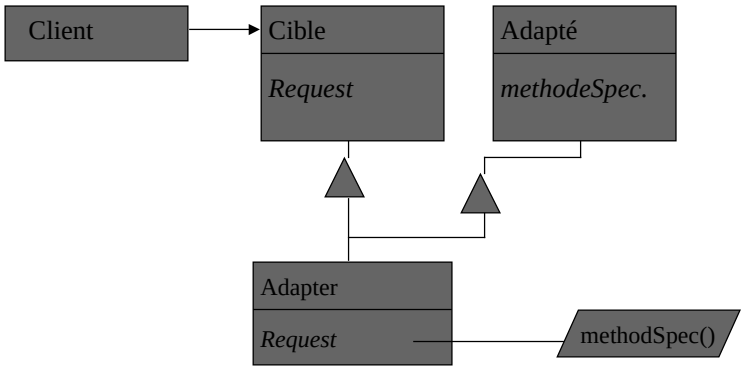
Utilisation

On utilise l'Adapter lorsque on veut utiliser :

- une classe existante dont l'interface ne convient pas
- plusieurs sous-classes mais il est est coûteux de redéfinir l'interface de chaque sous-classe en les sous-classant. Un adapter peut adapter l'interface au niveau du parent.

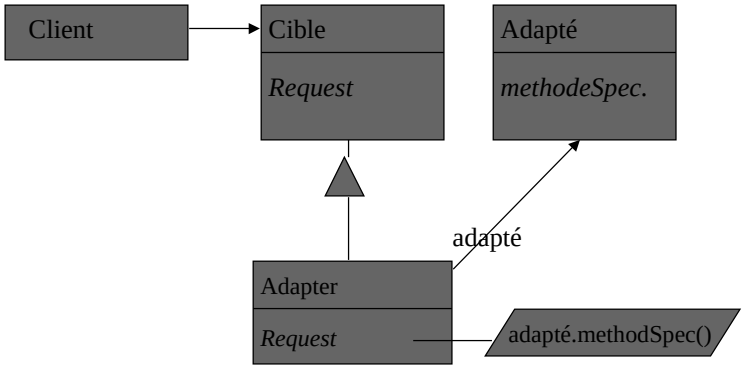
Adapter

Version héritage multiple



Adapter

Version composition

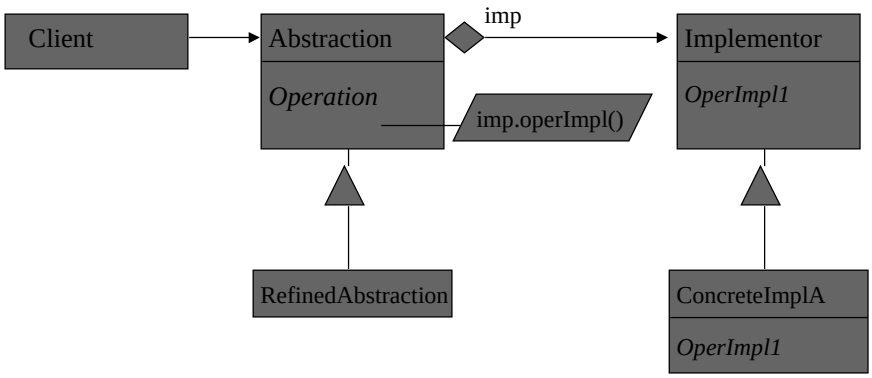


Utilisation

On utilise Bridge lorsque :

- on veut éviter un lien permanent entre l'abstraction et l'implantation (ex: l'implantation est choisie à l'exécution)
- l'abstraction et l'implantation sont toutes les deux susceptibles d'être raffinées
- les modifications subies par l'implantation ou l'abstraction ne doivent pas avoir d'impacts sur le client (pas de recompilation)

Bridge

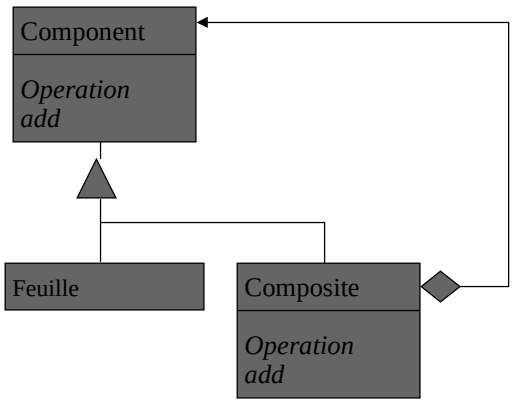


Utilisation

On utilise Composite lorsque on veut :

- représenter une hiérarchie d'objets
- ignorer la différence entre un composant simple et un composant en contenant d'autres. (interface uniforme)

Composite

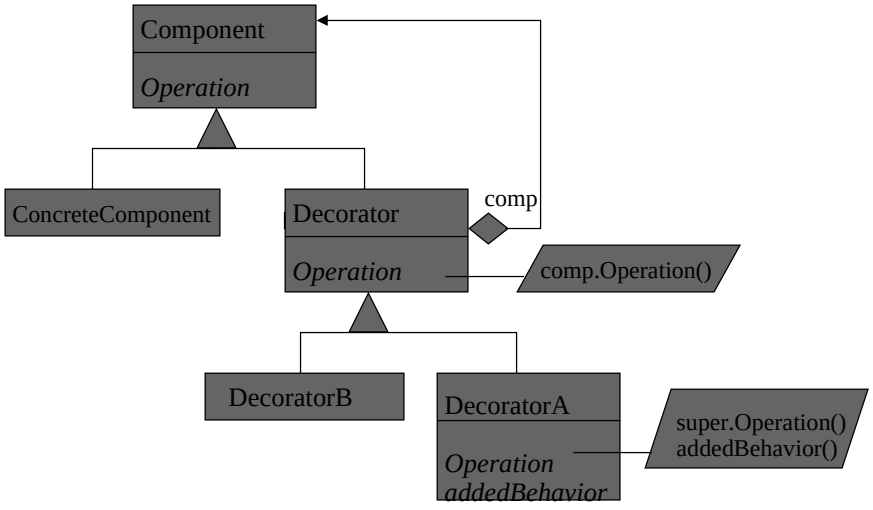


Utilisation

On utilise Decorator lorsque :

- il faut ajouter des responsabilités dynamiquement et de manière transparente
- il existe des responsabilités dont on peut se passer
- des extensions sont indépendantes et qu'il serait impraticable d'implanter par sous-classage

Decorator

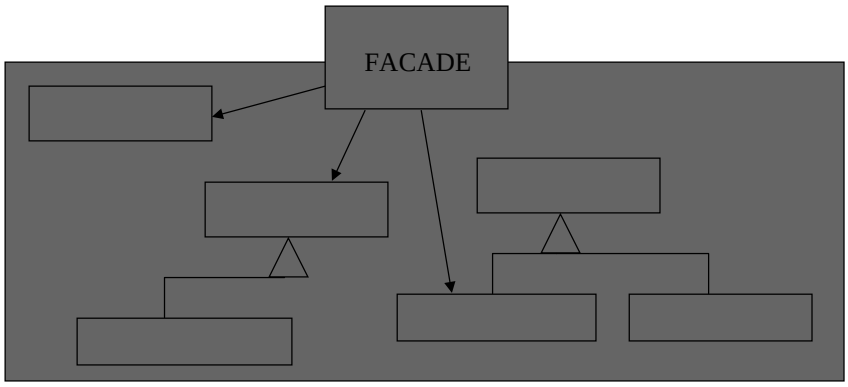


Utilisation

On utilise Facade lorsque on veut :

- fournir une interface simple à un système complexe
- introduire une interface pour découpler les relations entre deux systèmes complexes
- construire le système en couche

Facade

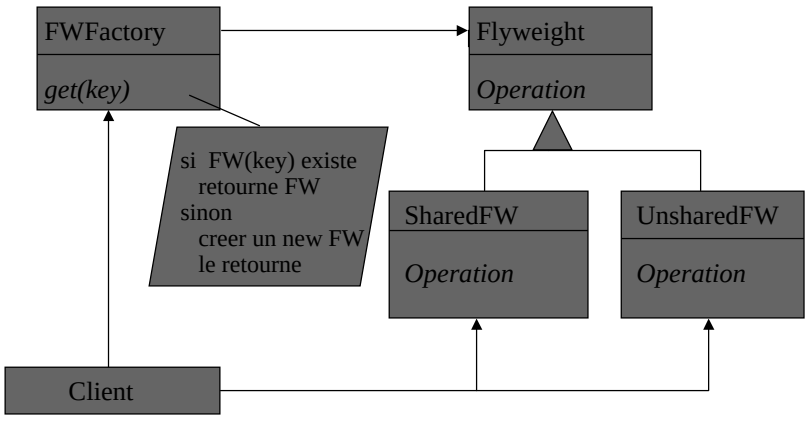


Utilisation

On utilise Flyweight lorsque :

- on utilise beaucoup d'objets, et
- les coûts de sauvegarde sont élevés, et
- l'état des objets peut être externalisé (extrinsic), et
- de nombreux groupes d'objets peuvent être remplacés par quelques objets partagés un fois que les états sont externalisés, et
- l'application ne dépend pas de l'identité des objets

Flyweight

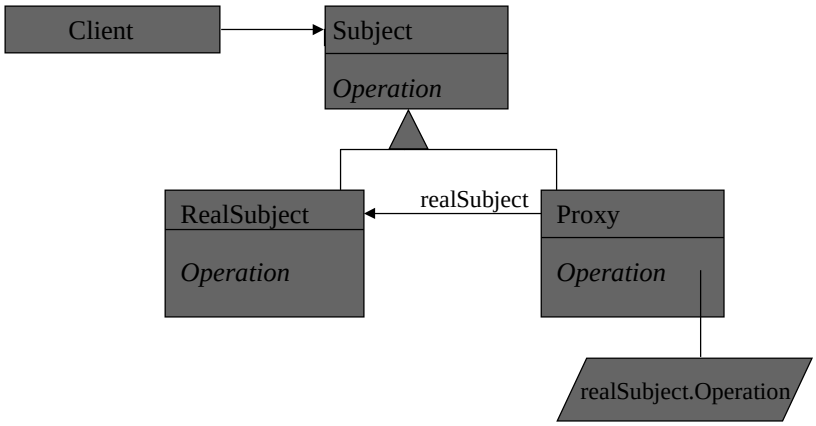


Utilisation

On utilise le Proxy lorsqu'on veut référencer un objet par un moyen plus complexe qu'un pointeur...

- remote proxy : ambassadeur
- protection proxy : contrôle d'accès
- référence intelligente
  - persistence
  - comptage de référence
  - …

Proxy



Behavioural Patterns

Formes de comportement pour décrire :

- des algorithmes
- des comportements entre objets
- des formes de communication entre objet

Principes

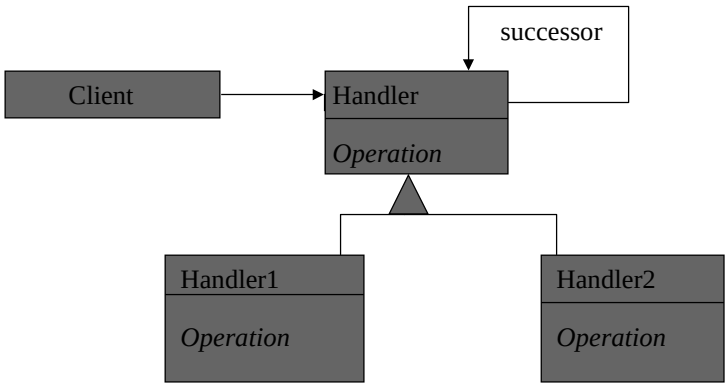
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Utilisation

On utilise Chain of Responsibility lorsque :

- plus d'un objet peut traiter une requète, et il n'est pas connu a priori
- l'ensemble des objets pouvant traiter une requète est construit dynamiquement

Chain of Responsibility

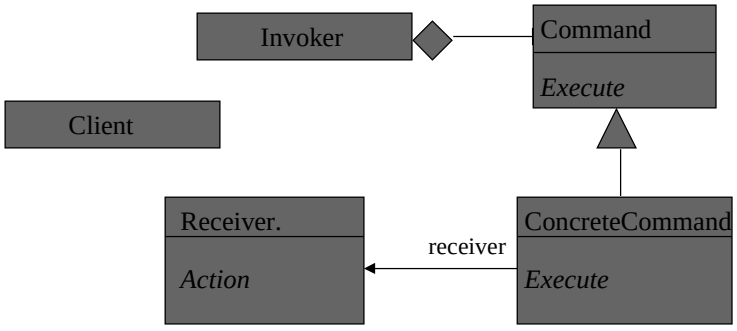


Utilisation

On utilise Command lorsque :

- spécifier, stocker et exécuter des actions à des moments différents.
- on veut pouvoir "défaire". Les commandes exécutées peuvent être stockées ainsi que les états des objets affectés...
- on veut implanter des transactions ; actions de "haut-niveau".

Command

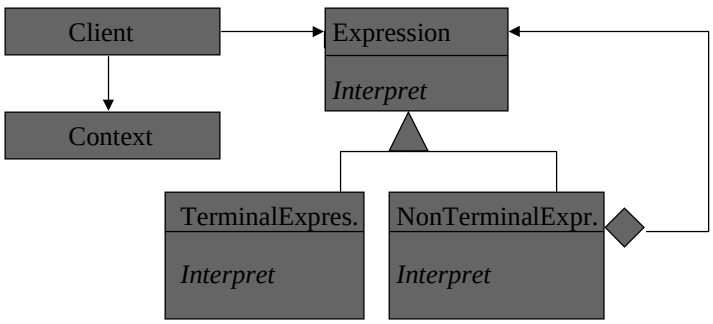


Utilisation

On utilise Interpreter lorsqu'il faut interpréter un langage et que :

- la grammaire est simple
- l'efficacité n'est pas un paramètre critique

Interpreter

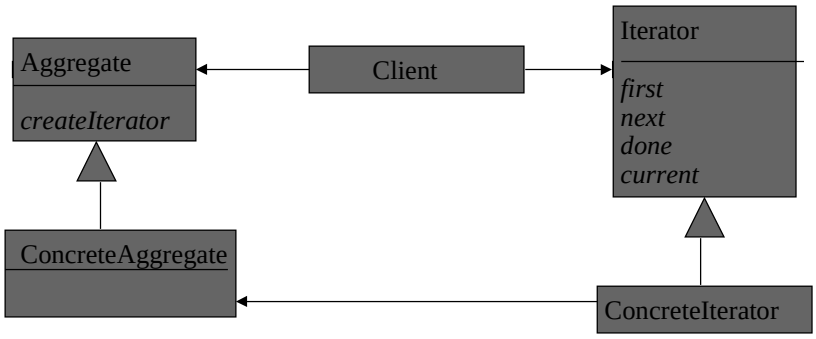


Utilisation

On utilise Iterator lorsque :

- pour accéder à un objet composé dont on ne veut pas exposer la structure interne
- pour offrir plusieurs manières de parcourir une structure composée
- pour offrir une interface uniforme pour parcourir différentes structures

Iterator

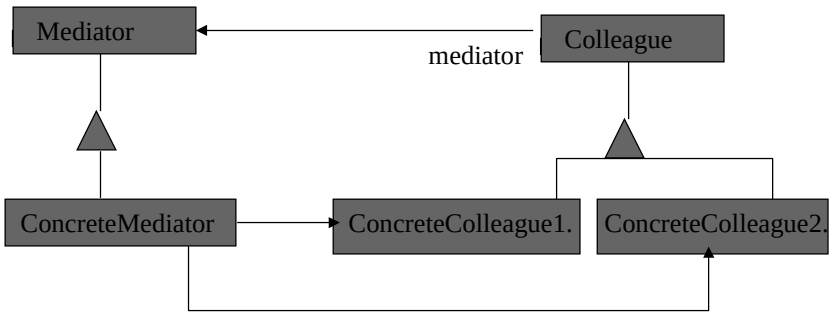


Utilisation

On utilise Mediator lorsque :

- quand de nombreux objets doivent communiquer ensemble
- la réutilisation d'un objet est délicate car il référence et communique avec de nombreux autres objets

Mediator



Utilisation

On utilise Memento lorsque :

- on veut sauvegarder tout ou partie de l'état d'un objet pour éventuellement pouvoir le restaurer, et
- une interface directe pour obtenir l'état de l'objet briserait l'encapsulation

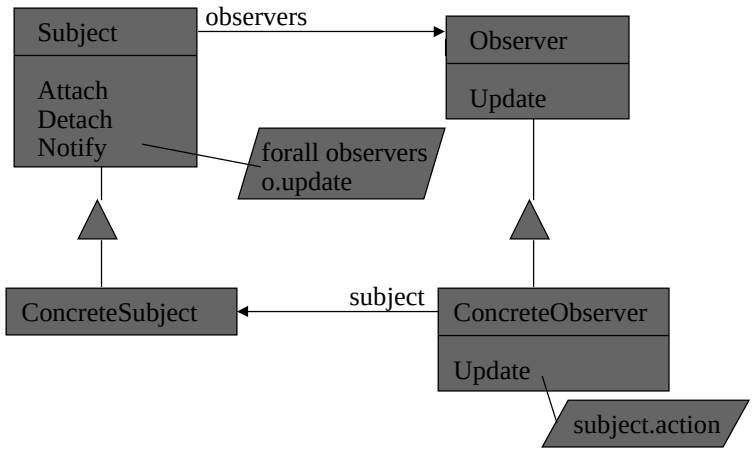
Memento

Utilisation

On utilise Observer lorsque :

- Une abstraction a plusieurs aspects, dépendant l'un de l'autre. Encapsuler ces aspects indépendament permet de les réutiliser séparément.
- Quand le changement d'un objet se répercute vers d'autres.
- Quand un objet doit prévenir d'autres objets sans pour autant les connaitre.

Observer

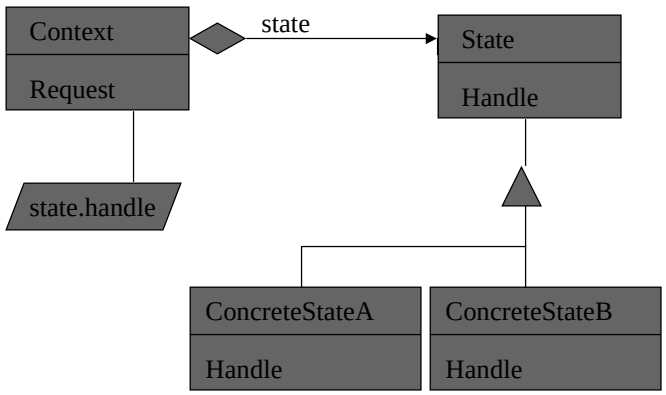


Utilisation

On utilise State lorsque :

- Le comportement d'un objet dépend de son état, qui change à l'exécution
- Les opérations sont constituées de partie conditionnelles de grande taille (case)

State

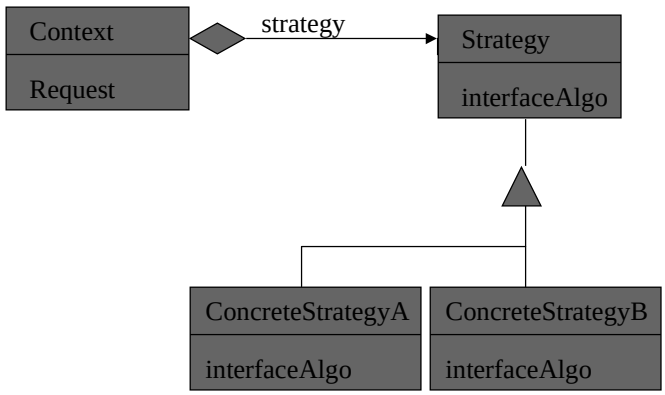


Utilisation

On utilise Strategy lorsque :

- de nombreuses classes associées ne diffèrent que par leur comportement. Stratégie offre un moyen de configurer une classe avec un comportement parmi plusieurs.
- on a besoin de plusieurs variantes d'algorithme.
- un algorithme utilise des données que les clients ne doivent pas connaitre.

Strategy

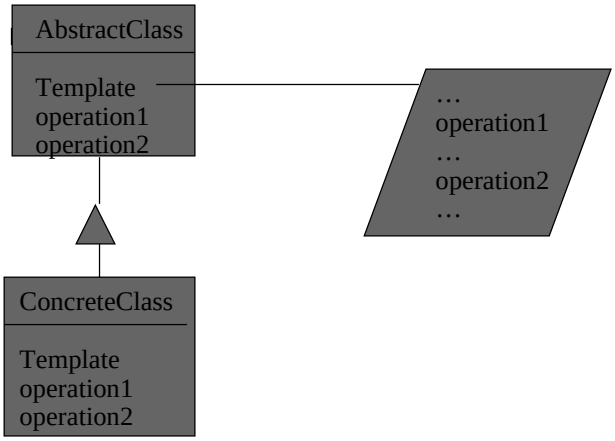


Utilisation

On utilise TemplateMethod :

- pour implanter une partie invariante d'un algorithme.
- pour partager des comportements communs d'une hiérarchie de classes.
- pour contrôler des extensions de sous-classe.

Template Method

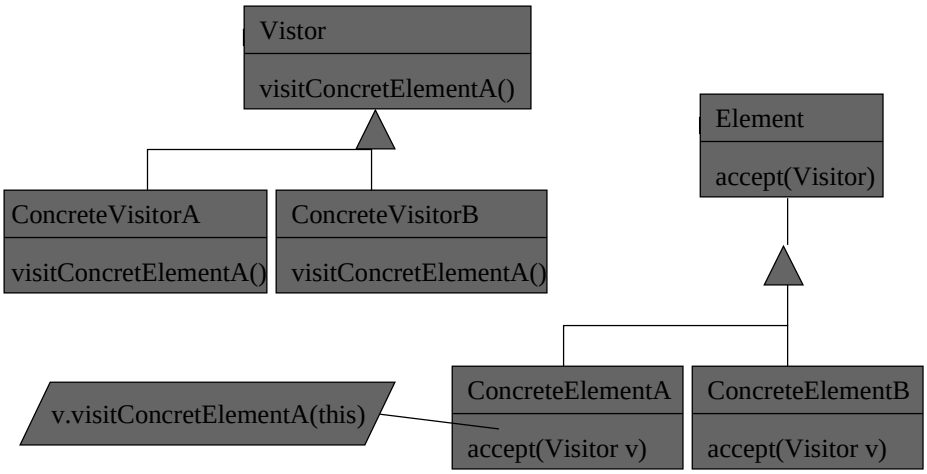


Utilisation

On utilise Visitor lorsque :

- une structure d'objets contient de nombreuses classes avec des interfaces différentes et on veut appliquer des operations diverses sur ces objets.
- les structures sont assez stables, et les opération sur leurs objets évolutives.

Visitor



Application des formes lors de la conception

BIS

- Trouver les bons objets
- Bien choisir la granularité des objets
- Spécifier les interfaces des objets
- Spécifier l'implantation des objets
- Mieux réutiliser
  - héritage vs composition
  - délégation
- Compiled-Time vs Run-Time Structures
- Concevoir pour l'évolution

Trouver les bons objets

Les patterns proposent des abstractions qui n'apparaissent pas "naturellement" en observant le monde réel :

- Composite : permet de traiter uniformément une structure d'objets hétérogènes
- Strategy : permet d'implanter une famille d'algorithmes interchangeables
- State

Ils améliorent la flexibilité et la réutilisabilité

Bien choisir la granularité des objets

La taille des objets peut varier considérablement ; comment choisir ce qui doit être décomposé ou au contraire regroupé ?

- Facade
- Flyweight

- Abstract Factory
- Builder

Spécifier les interfaces des objets

Qu’est-ce qui fait partie d’un objet ou non ?

- Memento ; mémorise les états, retour arrière
- Decorator ; augmente l'interface
- Proxy ; interface délégué
- Visitor ; regroupe des interfaces
- Facade ; cache une structure complexe d'objet

Spécifier l'implantation des objets

Différence type-classe…

- Chain of Responsibility ; même interface, mais implantations différentes
- Composite ; les Components ont une même interface dont l'implantation est en partie partagée dans le Composite
- Command, Observer, State, Strategy ne sont souvent que des interfaces abstraites
- Prototype, Singleton, Factory, Builder sont des abstractions pour créer des objets qui permettent de penser en termes d'interfaces et de leur associer différentes implantations

Mieux réutiliser

- héritage vs composition
  - white-box reuse ; rompt l'encapsulation - stricte ou non
  - black-box reuse ; flexible, dynamique

"Préférez la composition à l'héritage"

- délégation (redirection)
  - Une forme de composition...qui remplace l'héritage
  - Bridge découple l'interface de l'implantation
  - Mediator, Visitor, Proxy

Compiled-Time vs Run-Time Structures

- Aggrégation
  composition, is-part-of
  dépendence, durée de vie liée, responsabilité
  plus stable ~ compiled-time, statique

- Connaissance (accointance)
  lien, association
  relation fugitive, utilisation ponctuelle
  dynamique, run-time

Concevoir pour l'évolution (1)

Quelques raisons de "reengineering" :

- Création d'un objet en référençant sa classe explicitement...Lien à une implantation particulière...pour éviter utilisez AbstractFactory, FactoryMethod, Prototype
- Dépendance d'une opération spécifique...pour rendre plus souple utilisez Chain Of Responsibility, Command
- Dépendance d'une couche matérielle ou logicielle...AbstractFactory, Bridge

Concevoir pour l'évolution (2)

Quelques raisons de "reengineering" :

- Dépendance d'une implantation...pour rendre plus souple utilisez AbstractFactory, Bridge, Memento, Proxy
- Dépendance d'un algorithme particulier...Builder, Iterator, Strategy, TemplateMethod, Strategy
- Couplage fort...relâcher les relations utilisez AbstractFactory, Bridge, Chain Of Responsibility, Command, Facade, Mediator, Observer

Concevoir pour l'évolution (3)

Quelques raisons de "reengineering" :

- Etendre les fonctionnalités en sous-classant peut être couteux (tests, compréhension des superclasses, etc) utilisez aussi la délégation, la composition...Bridge, Chain Of Responsibility, Composite, Decorator, Observer, Strategy, Proxy
- Impossibilité de modifier une classe...absence du source, trop de répercussions, voyez Adapter, Decorator, Visitor

Autres patterns

Architecture :

- Couche : systèmes en couches (OS, piles OSI)
  proche de Facade

Algorithmique distribué :

- Jeton circulant
- Vague de calculs diffusant

Conclusion

- Architecture de logiciel
- Capitalisation d'expériences dans les Patterns
- Une forme se mémorise bien, s'adapte nécessairement
- Pas réservé aux objets
- Comme la prose, on les utilise sans le savoir ! mais quand on en est conscient, on améliore sa réflexion.