DESIGN PATTERNS

CATALOGUE DE MODÈLES DE CONCEPTION RÉUTILISABLES

Erich Gamma, Richard Helm,
Ralph Johnson et John Vlissides

Traduction de Jean-Marie Lasvergères

		Rôle		
		Créateur	**Structurel**	**Coportement**
Domaine	**Classe**	Fabrication (107)	Adaptateur (classe) (139)	Interprète (p. 285) Patron de méthode (p. 381)
	Objet	Fabrique abstraite (p. 101) Monteur (p. 113) Prototype (p. 137) Singleton (p. 149)	Adaptateur (objet) (p. 163) Pont (p. 177) Composite (p. 189) Décorateur (p. 203) Façade (p. 215) Poids Mouche (p. 227) Procuration (p. 241)	Chaîne de responsabilité (p. 259) Commande (p. 271) Itérateur (p. 301) Médiateur (p. 319) Mémento (p. 331) Observateur (p. 343) Etat (p. 357) Stratégie (p. 369) Visiteur (p. 387)

Tableau 1.1 : Espace des modèles de conception

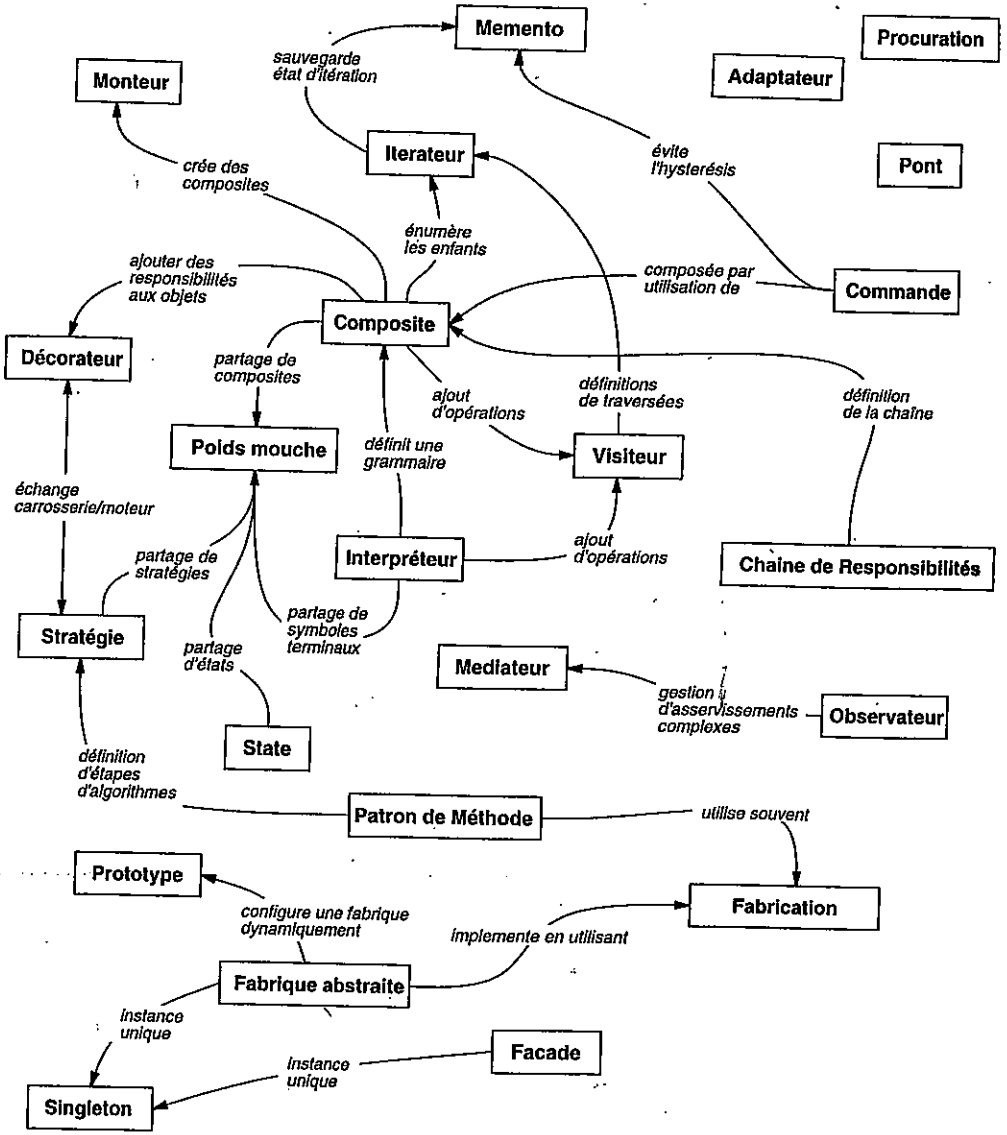


Figure 1.1 : Relations entre modèles de conception

1.4 Catalogue des modèles de conception

Le catalogue qui commence à la page 93 contient 23 modèles de conception. Leurs noms et leurs intentions sont présentés ci-après, afin d'en donner, une vue globale. Le nombre entre parenthèses, figurant après chaque nom de modèle, donne le numéro de la page ou celui-ci est catalogué ; c'est une convention utilisée tout au long de l'ouvrage.

Fabrique Abstraite (Abstract Factory, p. 101) Fournit une interface, pour créer des familles d'objets apparentés ou dépendants, sans avoir à spécifier leurs classes concrètes.

Adaptateur (Adapter, p. 163) Convertit l'interface d'une classe en une interface distincte, conforme à l'attente de l'utilisateur. L'adaptateur permet à des classes de travailler ensemble, qui n'auraient pu le faire autrement pour cause d'interfaces incompatibles.

Pont (Bridge, p. 177) Découple une abstraction de son implémentation associée, afin que les deux puissent être modifiés indépendamment.

Monteur (Builder, p. 113) Dans un objet complexe, dissocie sa construction de sa représentation, de sorte que, le même procédé de construction puisse engendrer des représentations différentes.

Chaîne de responsabilité (Chain of responsibility, p. 259) Permet d'éviter de coupler l'expéditeur d'une requête à son destinataire, en donnant la possibilité à plusieurs objets de prendre en charge la requête. Pour ce faire, il chaîne les objets récepteurs, et fait passer la requête tout au long de cette chaîne jusqu'à ce qu'un des objets la prenne en charge.

Commande (Command, p. 271) Encapsule une requête comme un objet, ce qui permet de faire un paramétrage des clients avec différentes requêtes, files d'attente, ou historiques de requêtes, et d'assurer le traitement des opérations réversibles.

Composite (Composite, p. 189) Organise les objets en structure arborescente représentant la hiérarchie de bas en haut. Le Composite permet aux utilisateurs de traiter des objets individuels, et des ensembles organisés de ces objet,s de la même façon.

Décorateur (Decorator, p. 203) Attache des responsabilités supplémentaires à un objet de façon dynamique. Il permet une solution alternative pratique pour l'extension des fonctionnalités, à celle de dérivation de classes.

Façade (Facade, p. 215) Fournit une interface unifiée pour un ensemble d'interfaces d'un sous-système. Façade définit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.

Fabrication (Factory method, p. 125) Définit une interface pour la création d'un objet, tout en laissant à des sous-classes le choix de la classe à instancier. Une fabrication permet de déférer à des sous-classes les instanciations d'une

Poids Mouche (Flyweight, p. 227) Assure en mode partagé le support efficace d'un grand nombre d'objets à fine granularité.

Interprète (Interpreter, p. 285) Dans un langage donné, il définit une représentation de sa grammaire, ainsi qu'un interprète qui utilise cette représentation pour analyser la syntaxe du langage.

Itérateur (Iterator, p. 301) Fournit un moyen pour accéder en séquence aux éléments d'un objet de type agrégat sans révéler sa représentation sous-jacente.

Médiateur (Mediator, p. 319) Définit un objet qui encapsule les modalités d'interaction de divers objets. Le médiateur favorise les couplages faibles, en dispensant les objets d'avoir à faire référence explicite les uns aux autres ; de plus, il permet de modifier une relation indépendamment des autres.

Mémento (Memento, p. 331) Sans violer l'encapsulation, acquiert et délivre à l'extérieur une information sur l'état interne d'un objet, afin que celui-ci puisse être rétabli ultérieurement dans cet état.

Observateur (Observer, p. 343) Définit une corrélation entre objets du type un à plusieurs, de façon que, lorsqu'un objet change d'état, tous ceux qui en dépendent, en soient notifiés et mis à jour automatiquement.

Prototype (Prototype, p. 137) Spécifie les espèces d'objets à créer, en utilisant une instance de type prototype, et crée de nouveaux objets par copies de ce prototype.

Procuration (Proxy, p. 241) Fournit un subrogé ou un remplaçant d'un autre objet, pour en contrôler l'accès.

Singleton (Singleton, p. 149) Garantit qu'une classe n'a qu'une seule instance, et fournit à celle-ci, un point d'accès de type global.

Etat (State, p. 357) Permet à un objet de modifier son comportement lorsque son état interne change. L'objet paraîtra changer de classe.

Stratégie (Strategy, p. 369) Définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Une stratégie permet de modifier un algorithme indépendamment de ses clients.

Patron de méthode (Template Method, p. 381) Définit le squelette de l'algorithme d'une opération, en déléguant le traitement de certaines étapes à des sous-classes. Le patron de méthode permet aux sous-classes de redéfinir certaines étapes d'un algorithme sans modifier la structure de l'algorithme.

Visiteur (Visitor, p. 387) Représente une opération à effectuer sur les éléments d'une structure d'objet. Le visiteur permet de définir une nouvelle opération sans modifier les classes des éléments sur lesquels il opère.

SINGLETON

Objet - Créateur

Intention

Garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès de type global à cette classe.

Motivation

Il est important pour certaines classes de n'avoir qu'une seule instance. Alors qu'il peut y avoir de nombreuses imprimantes dans un système, il ne doit y avoir qu'un serveur d'impression. Il ne doit y avoir qu'un fichier système et qu'un seul gestionnaire de fenêtres. Un filtre binaire ne peut avoir qu'un seul convertisseur A/D. Un système de comptabilité sera dédié au service d'une seule entreprise.

Comment assurer qu'une classe n'a qu'une instance, et que cette dernière est facilement accessible ? Une variable globale permet d'accéder à un objet, mais elle n'empêche pas des instanciations multiples de cet objet.

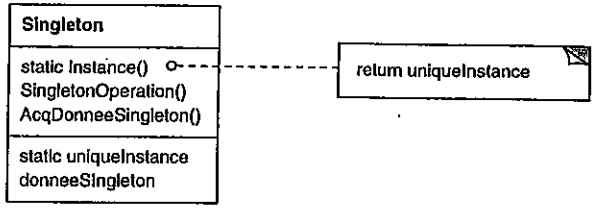
Une meilleure solution consiste à confier à la classe elle même la responsabilité d'assurer l'unicité de son instance. La classe peut assurer qu'aucune autre instance ne sera créée (en interceptant les requêtes demandant à créer de nouveaux objets), et elle peut fournir un moyen pour accéder à cette instance. C'est le modèle Singleton.

Indications d'utilisation

On utilisera le modèle Singleton :

- S'il doit n'y avoir exactement qu'une instance d'une classe, qui, de plus, doit être accessible aux clients en un point bien déterminé.
- Si l'l'instance unique doit être extensible par dérivation en sous-classe, et si l'utilisation d'une instance étendue doit être permise aux clients, sans qu'ils aient à modifier leur code.

Structure



Constituants

- **Singleton**

  Le Singleton définit une opération Instance qui donne au client l'accès à son unique instance. Instance est une opération de classe (c'est-à-dire, une méthode de classe en Smalltalk, et une fonction membre statique en C++).

  Il peut avoir la charge de créer sa propre instance unique.

Collaborations

- Les clients accèdent à l'instance d'un Singleton par le seul intermédiaire de l'opération Instance de Singleton.

Conséquences

Le modèle Singleton a de nombreux avantages :

1. *Accès contrôlé à une instance unique.* Du fait que la classe Singleton encapsule son unique instance, elle peut contrôler exactement comment et quand les clients y accèdent.
2. *Réduction de l'espace des noms.* Le modèle Singleton est une amélioration des variables globales. Il évite de polluer l'espace des noms avec des variables globales destinées à stocker des instances uniques.
3. *Raffinement des opérations et de la représentation.* La classe Singleton peut être étendue par dérivation, et l'on peut facilement constituer une application avec une instance de cette classe étendue. On peut composer l'application, à l'exécution, avec une instance de la classe recherchée.
4. *Autorise un nombre variable d'instances.* Le modèle permet de changer de stratégie en autorisant plus d'une instance de la classe Singleton. De

plus, on peut utiliser la même approche pour contrôler le nombre d'instances demandées par l'application. Seules les opérations qui garantissent l'accès à l'instance Singleton doivent changer.

5. *Souplesse améliorée par rapport aux opérations de classes.* Les opérations de classes (les fonctions membres statiques en C++, ou les méthodes de classes en Smalltalk) fournissent une autre façon de mettre en place une fonctionnalité Singleton. Mais les particularités techniques des deux langages font qu'il est difficile dans ce cas de changer une réalisation pour permettre plus d'une seule instance d'une classe. De plus les fonctions membres statiques de C++ ne sont jamais virtuelles et donc ne peuvent être surchargées en utilisant le polymorphisme avec des sous-classes.

ETAT

Objet - Comportemental

Intention

Permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe.

Alias

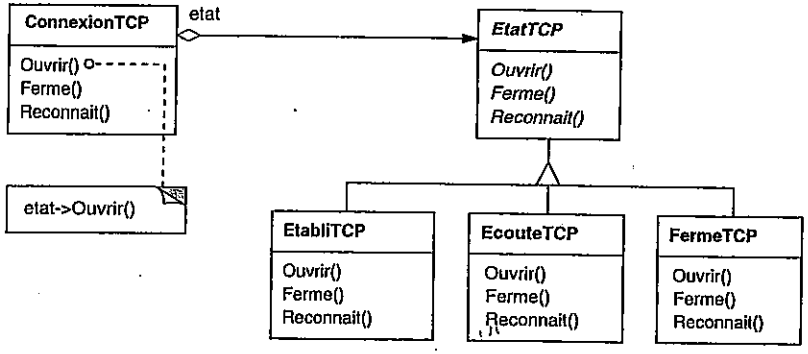
Objets d'état.

Motivation

Considérons une classe ConnexionTCP représentant une connexion de réseau. Un objet ConnexionTCP peut se trouver dans un état parmi plusieurs : établi, écoute, fermé. Quand un objet ConnexionTCP reçoit une requête d'autres objets, il répond de différentes façons, en fonction de son état courant. Par exemple, l'effet d'une demande d'ouverture est différent selon que la connexion est dans l'état fermé, ou dans l'état établi. Le modèle Etat décrit les différents comportements que peut avoir ConnexionTCP dans chacun de ses états.

L'idée de base de ce modèle est d'introduire une classe abstraite appelée EtatTCP, qui représente les états de la connexion du réseau. La classe EtatTCP déclare l'interface commune à toutes les classes représentant les différents états opérationnels. Les sous-classes de EtatTCP implémentent les comportements spécifiques des états. Par exemple, les classes EtabliTCP et FermeTCP implémentent les comportements particuliers aux états Etabli et Ferme de ConnexionTCP.

La classe ConnexionTCP gère un objet état (une instance d'une sous-classe de EtatTCP) qui représente l'état de la connexion TCP. La classe ConnexionTCP délègue toutes les requêtes spécifiques d'un état, à son objet état. ConnexionTCP utilise des interfaces de sa sous-classe EtatTCP pour effectuer des opérations particulières spécifiques d'un état de la connexion.



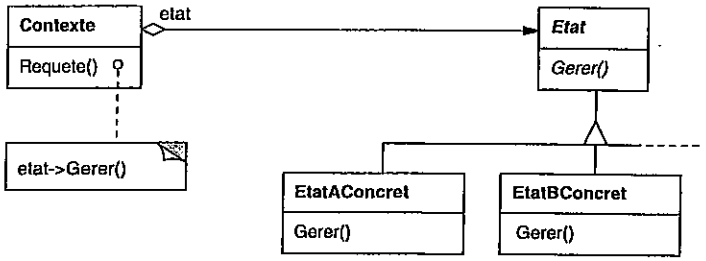
Chaque fois que la connexion change d'état, l'objet ConnexionTCP change l'objet état qu'il utilise. Si la connexion passe, par exemple, de Etabli à Ferme, ConnexionTCP remplacera l'instance EtabliTCP, par l'instance FermeTCP.

Indications d'utilisation

On utilisera le modèle Etat dans l'un des cas suivants :

- Le comportement d'un objet dépend de son état, et ce changement de comportement doit intervenir dynamiquement, en fonction de cet état.
- Les opérations comportent de grands pans entiers de déclarations conditionnelles fonctions de l'état de l'objet. Cet état est généralement désigné par une ou plusieurs constantes d'énumération. Souvent, plusieurs opérations différentes, contiendront la même structure conditionnelle. Le modèle Etat place dans une classe séparée, chacune des branches de la condition. Ceci permet de traiter l'état de l'objet, comme un objet à part entière, qui peut varier indépendamment des autres objets.

Structure



Constituants

- **Contexte** (ConnexionTCP)
  - Définit l'interface intéressant les clients.
  - Gère une instance d'une sous-classe EtatConcret qui définit l'état en cours.
- **Etat** (EtatTCP)
  - Définit une interface qui encapsule le comportement associé avec un état particulier de Contexte.
- **Sous-classes EtatConcret** (EtabliTCP, EcouteTCP, FermeTCP)
  - Chaque sous-classe implémente un comportement associé avec l'état de contexte.

Collaborations

- Le Contexte délègue les requêtes spécifiques d'état à l'objet EtatConcret courant.
- Un contexte peut passer en argument sa propre référence, à l'objet Etat en charge de gérer la requête. Cet objet peut donc accéder au contexte, si nécessaire.
- Contexte est l'interface primaire pour les utilisateurs. Ceux-ci peuvent composer un contexte à l'aide d'objets Etat. Une fois qu'ils ont configuré un contexte, les utilisateurs n'ont plus à « traiter » directement avec les objets Etat.
- est soit à Contexte, soit aux sous-classes EtatConcret, qu'il revient de décider, de l'état qui succède à un autre état, et selon quelles modalités.

Conséquences

Le modèle Etat a les effets suivants :

1. *Il isole les comportements spécifiques d'états et fait un partitionnement des différents comportements état par état.* Le modèle Etat place dans un unique objet, tous les comportements associés à un état déterminé. Du fait que tout le code spécifique d'un état, réside dans une sous-classe Etat, les nouveaux états et les transitions, pourront être ajoutées aisément par la définition de nouvelles sous-classes.

Une solution alternative consisterait à utiliser des valeurs de données décrivant les états internes, et disposer d'opérations de Contexte, testant explicitement ces données. Mais on aurait, alors, des déclarations du genre des tests comparatifs, ou des aiguillages cas par cas (switch - case), éparpillées dans tout le code de Contexte. L'ajout d'un nouvel état nécessiterait alors, de modifier plusieurs opérations, compliquant d'autant la maintenance.

Le modèle Etat évite ce problème, mais peut en introduire un autre, car il distribue les comportements correspondants aux différents états, parmi différentes sous-classes Etat. Ceci augmente le nombre des classes et est donc moins dense qu'une classe unique. Mais une telle répartition est effectivement bonne dans le cas de nombreux états, qui auraient nécessité autrement des déclarations conditionnelles de grande taille.

Comme les grandes procédures, les grandes déclarations de type conditionnel ne sont pas souhaitables. Elles ont un caractère monolithique, ont tendance à rendre le code moins explicite, ce qui a pour effet de compliquer leur modification et leur extension. Le modèle Etat propose un meilleur moyen pour structurer du code traitant des états. La logique qui détermine les transitions d'états ne réside plus, dès lors, dans des déclarations if et switch monolithiques, elle est, au contraire, répartie entre les sous-classes Etat. L'encapsulation dans une classe, de chaque transition d'état et de chaque action, transforme la notion d'état de traitement, en celle de statut général d'un objet. Ceci impose au code une structure, et rend sa destination plus manifeste.

2. *Il rend les transitions d'état plus explicites.* Lorsqu'un objet définit son état courant, uniquement à partir de valeurs de données internes, les représentations de ses changements d'état ne sont pas très explicites ; elles n'apparaissent que sous la forme de valeurs assignées à certaines variables. L'introduction d'objets distincts pour différents états, rend les transitions plus explicites. De plus, les objets Etat évitent les états internes non significatifs du Contexte, car les transitions d'état relatives à un Contexte sont dénombrables - elles s'expriment par la redéfinition de type d'*une seule* variable (la variable objet Etat de Contexte), et non de plusieurs [dCLF93].
3. *Les objets Etat peuvent être partagés.* Si les objets Etat n'ont pas de variables d'instance - c'est-à-dire si l'état qu'ils représentent est intégralement exprimé dans leur type - alors, différents contextes peuvent partager un objet Etat. Quand des états sont ainsi partagés, il s'agit généralement de poids mouche (voir le modèle Poids Mouche (p. 227)), sans état intrinsèque, mais seulement dotés d'un comportement.

STRATEGIE

Objet - Comportemental

Intention

Définit une famille d'algorithmes, encapsule chacun d'entre eux, et les rend interchangeables. Le modèle Stratégie permet aux algorithmes d'évoluer indépendamment des clients qui les utilisent.

Alias

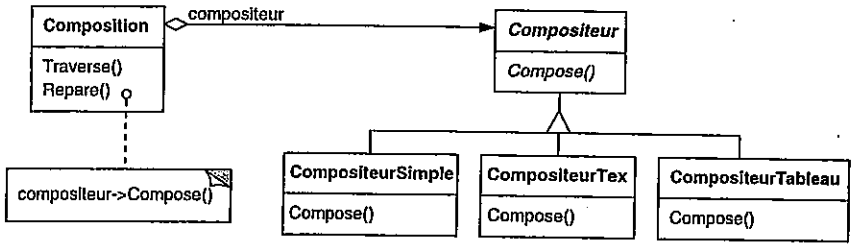
Politique.

Motivation

Il existe de nombreux algorithmes pour découper un flot de texte en lignes. Le codage en dur de tels algorithmes dans les classes mêmes qui les requièrent, n'est pas recommandé pour plusieurs raisons :

- Les programmes client qui réclament ce découpage en lignes, deviennent plus compliqués s'ils incorporent le code correspondant. Ils deviennent plus volumineux, et plus difficiles à maintenir, particulièrement s'ils offrent toute une variété d'algorithmes de découpage.
- Différents algorithmes conviendront à différents instants. On ne souhaite pas offrir de multiples algorithmes de découpage en lignes, si tous ne sont pas utilisés.
- Il est difficile d'ajouter de nouveaux algorithmes ou de modifier ceux existants, lorsque le découpage en lignes fait partie intégrante du client.

On peut éviter ce type de problème en définissant des classes qui encapsulent les divers algorithmes de découpage en lignes. Un algorithme encapsulé de la sorte est nommé **stratégie**.



Supposons une classe Composition chargée de la gestion et de la mise à jour des ruptures de lignes d'un texte projeté à l'écran d'un afficheur de texte. Les stratégies de découpage en lignes ne sont pas implémentées par cette classe. En fait elles sont implémentées séparément par des sous-classes de la classe abstraite Compositeur. Les sous-classes de Compositeur implémentent différentes stratégies :

- **CompositeurSimple** implémente une stratégie simple qui détermine les ruptures de lignes une à une.
- **CompositeurTeX** implémente l'algorithme TEX pour déterminer les ruptures de lignes. Cette stratégie tente d'optimiser globalement les ruptures de lignes, c'est-à-dire pour tout un paragraphe à la fois.
- **CompositeurTableau** implémente une stratégie qui choisi les ruptures de lignes de façon telle que chaque ligne contienne un nombre prédéterminé d'éléments. C'est pratique, par exemple, pour répartir une série d'icônes sur des lignes.

Une Composition gère une référence à un objet Compositeur. Chaque fois qu'une Composition remanie le format de son texte, elle délègue cette responsabilité à son objet Compositeur. Le client de Composition précise celui des Compositeur qu'il choisit d'employer, en installant ce dernier dans la Composition.

Indications d'utilisation

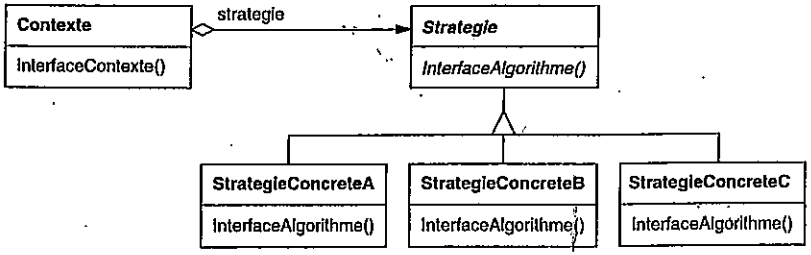
On utilisera le modèle Stratégie dans les cas où :

- Plusieurs classes apparentées ne diffèrent que par leur comportement. Les Stratégies donnent le moyen d'appareiller une classe avec un comportement parmi plusieurs autres.
- On a besoin de diverses variantes d'un algorithme. Par exemple, on peut définir des algorithmes représentants différents compromis encombrement mémoire / temps d'exécution. Les Stratégies peuvent être utilisées

quand ces variantes sont implémentées sous la forme d'une hiérarchie de classe d'algorithmes [HO87].

- Un algorithme utilise des données que les clients n'ont pas à connaître. Utiliser la Stratégie dispense d'avoir à révéler des structures complexes de données spécifiques des algorithmes.
- Une classe définit de nombreux comportements, qui figurent dans ses opérations sous la forme de déclarations conditionnelles multiples. Plutôt que laisser subsister ces expressions conditionnelles, on déplacera les sections correspondantes aux différentes branches, dans des classes Strategie qui leurs sont propres.

Structure



Constituants

- **Stratégie** (Compositeur)
  - Déclare une interface commune à tous les algorithmes représentés. Contexte utilise cet interface pour appeler l'algorithme défini par une StrategieConcrete.
- **StrategieConcrete** (CompositeurSimple, CompositeurTeX, Compositeur-Tableau)
  - Implémente l'algorithme, en utilisant l'interface Stratégie.
- **Contexte** (Composition)
  - Est composé à l'aide d'un objet StrategieConcrete.
  - Gère une référence à un objet Strategie.
  - Peut définir une interface qui permette à Stratégie d'accéder à ses données.

Collaborations

- Strategie et Contexte interagissent dans l'implémentation de l'algorithme choisi. Un Contexte peut fournir à la stratégie toutes les données que requiert cet algorithme, lorsqu'il est appelé. De même, le contexte peut se communiquer lui-même, en tant qu'argument, aux opérations de Strategie. C'est ce qui permet à la stratégie de se mettre en rappel sur le contexte en cas de besoin.
- Un contexte transmet les requêtes de ses clients à sa stratégie. En général, les clients créent un objet StrategieConcrete, et le passent au contexte ; par la suite, les clients interagissent exclusivement avec le contexte. Souvent le client pourra choisir à partir d'une famille de classes StrategieConcrete.

Conséquences

Le modèle Stratégie possède les avantages et les inconvénients suivants :

1. *Familles d'algorithmes apparentés.* Les hiérarchies de classes Strategie définissent une famille d'algorithmes ou comportements réutilisables par les contextes. L'héritage peut faciliter l'isolation des fonctionnalités communes aux algorithmes.
2. *Une solution alternative à la dérivation en sous-classes.* L'héritage représente un autre moyen de fournir une variété d'algorithmes ou de comportements. On peut directement dériver une classe Contexte, pour la doter de nouveaux comportements. Mais cela a pour effet de coder en dur le comportement dans le contexte, donc de mélanger l'implémentation de l'algorithme avec celle du contexte, rendant ce dernier plus difficile à appréhender, maintenir, et enrichir. De plus cela ne permet pas de modifier l'algorithme dynamiquement. On aboutit avec de nombreuses classes apparentées, qui n'ont pour seule différence que l'algorithme qu'elles emploient ou le comportement qui s'y rattache. L'encapsulation de l'algorithme dans des classes Strategie séparées, permet de modifier les algorithmes indépendamment du contexte, en facilitant ainsi la substitution, la compréhension, et l'extension.
3. *Les stratégies dispensent de l'utilisation de déclarations conditionnelles.* Le modèle Stratégie offre une solution alternative aux expressions conditionnelles permettant de choisir le comportement désiré. Si différents comportements sont regroupés dans une même classe, il est difficile de ne pas utiliser les déclarations conditionnelles pour sélectionner les comportements adéquats. Encapsuler ceux-ci dans différentes classes Strategie, permet de supprimer ce genre de déclarations.

FACADE

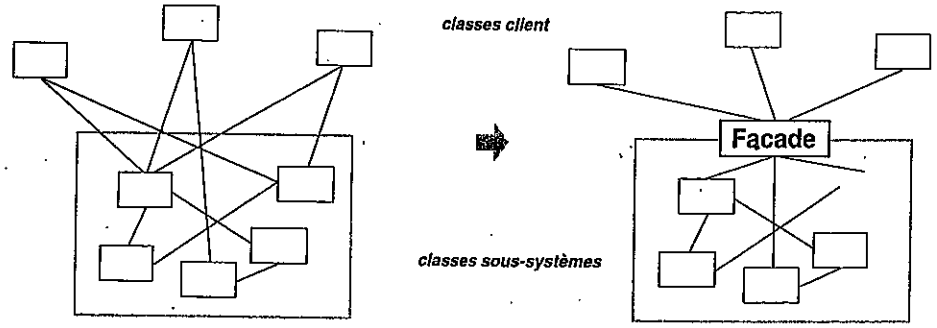
Objet - Structurel

Intention

Fournit une interface unifiée, à l'ensemble des interfaces d'un sous-système. La façade fournit une interface de plus haut niveau, qui rend le sous-système plus facile à utiliser.

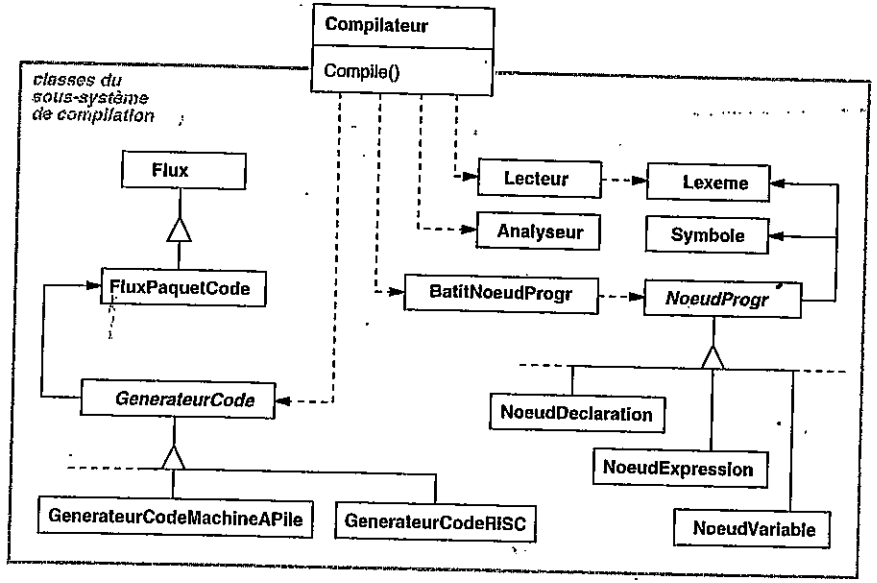
Motivation

Structurer un système en sous-systèmes aide à réduire la complexité de l'ensemble. Un des objectifs commun à toutes les conceptions est de réduire la multiplicité des communications et des liens de dépendance entre sous-systèmes. Un moyen pour y parvenir, consiste à introduire un objet Facade, qui offre une interface unique et simplifiée aux service de haut niveau d'un sous-système.



Considérons, par exemple, un environnement de programmation qui donnerait accès au sous-système de son compilateur. Ce sous-système contient des classes telles que Lecteur, Analyseur, NoeudProgr, FluxPaquetCode, et ComposeNoeudProgr qui sont le fondement de l'implémentation du compilateur. Certaines applications spécialisées, peuvent avoir besoin d'accéder à ces classes directement. Mais la plupart des clients ne s'intéressent pas à des détails, tels que, analyse syntaxique et génération de code ; ils souhaitent plutôt compiler certains segments de code. Pour eux les interfaces puissantes, mais de bas niveau du sous-système compilateur, ne font que compliquer leur tâche.

Pour offrir une interface de plus haut niveau, qui puisse occulter aux clients ces classes, le sous-système compilateur comporte également une classe Compilateur. Cette classe fournit une interface standard d'accès aux fonctionnalités du compilateur. La classe Compilateur joue le rôle de façade : elle offre au client une interface unique et simple vers le sous-système compilateur. Elle combine les classes qui implémentent les fonctionnalités du compilateur, sans les cacher totalement. La façade d'un compilateur facilite la vie de bien des programmeurs, sans empêcher l'accès aux fonctionnalités de niveau très spécifique, auxquelles quelques uns demandent accès.



Indications d'utilisation

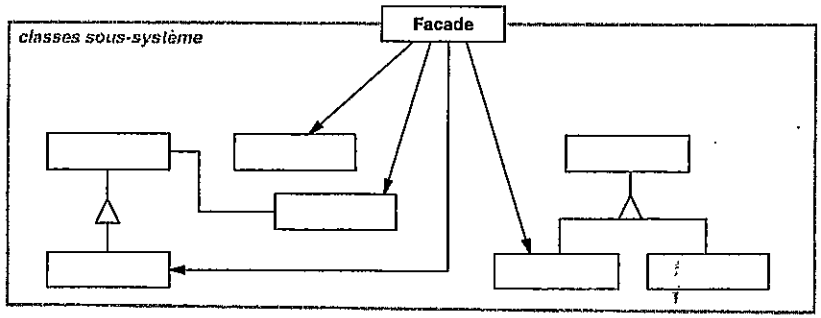
On utilisera le modèle Façade lorsque :

- On souhaite disposer d'une interface simple pour un sous-système complexe. Les sous-systèmes deviennent souvent plus complexes au fur et à mesure de leur évolution. La plupart des modèles, lorsqu'ils sont employés, engendrent des classes plus nombreuses et plus petites. Ceci rend le sous-système plus réutilisable et plus facile à personnaliser, mais il devient aussi plus difficile à employer pour les classes qui n'ont pas besoin de personnalisation. Une façade propose une simple vue par défaut du sous-système, qui est suffisante pour la plupart des clients.

Seuls les clients demandant plus de spécificité devront regarder derrière la façade.

- Il y a beaucoup de relations de dépendance entre les clients et les classes d'implémentation d'une abstraction. On introduira une façade pour découpler le sous-système des clients et des autres sous-systèmes, favorisant ainsi l'indépendance et la portabilité du sous-système.
- On cherche à structurer en niveaux un sous-système. On utilisera la façade pour définir un point d'entrée à chaque niveau du sous-système. Si des sous-système sont interdépendants, on peut simplifier les relations entre eux en les faisant communiquer l'un avec l'autre, uniquement à travers leurs façades.

Structures



Constituants

- **Facade** (Compilateur)
  - connaît les classes du sous-système compétentes pour une requête.
  - délègue le traitement des requêtes clients aux objets appropriés du sous-système.
- **Les classes du sous-système** (Lecteur, Analyseur, NoeudProgr, etc.)
  - implémentent les fonctionnalités du sous-système.
  - gèrent les travaux assignés par l'objet Facade.
  - ne connaissent pas la façade ; c'est-à-dire qu'elles n'ont pas de références à celle-ci.

Collaborations

- Les clients communiquent avec le sous-système en envoyant des requêtes à la façade, qui répercute celles-ci aux objets appropriés du sous-système. Bien que les objets du sous-système effectuent réellement le travail, la façade peut avoir à charge, en propre, le travail de transcription de son interface à ceux du sous-système.
- Les clients qui utilisent la façade n'ont pas à accéder directement aux objets de son sous-système.

Conséquences

Le modèle Façade présente les avantages suivants :

1. Il masque au client des composants du sous-système, donc réduit le nombre d'objets, dont celui-ci doit tenir compte, et rend le sous-système plus facile à utiliser.
2. Il favorise le couplage faible entre le sous-système et ses clients. Les composants d'un sous-système sont souvent fortement couplés. Le couplage faible permet de faire évoluer les composants d'un sous-système, sans affecter ses clients. Les façades aident à la structuration en couches d'un système et facilitent les relations entre objets. Elles peuvent éliminer les interdépendances complexes ou circulaires. Cet avantage peut être particulièrement important si le client et le sous-système ont été implémentés indépendamment.

   Réduire les interdépendances à la compilation est vital dans les grands logiciels système. On peut vouloir économiser le temps de re-compilations, lorsqu'il y a modification de classes du sous-système. Avec les façades, la réduction des interdépendances à la compilation, peut limiter la re-compilation qu'exigerait même une petite modification dans un sous-système important. Une façade peut également simplifier le portage de systèmes vers d'autres plates-formes, puisque la construction d'un sous-système ne nécessite pas forcément celle de tous les autres.
3. Il n'empêche pas les applications d'utiliser les classes du sous-système, si nécessaire. On peut ainsi choisir entre confort d'utilisation et exhaustivité.