Programmation objet avancée : la conception avant tout (design patterns à l'emploi)

Par Michaël Haberzettel (devil may cry)

le Site du Zéro
www.siteduzero.com

Sommaire

Programmation objet avancée : la conception avant tout (design patterns à l'emploi)

Le tutoriel que vous êtes en train de lire est en **bêta-test**. Son auteur souhaite que vous lui fassiez part de vos commentaires pour l'aider à l'améliorer avant sa publication officielle. Notez que le contenu n'a pas été validé par l'équipe éditoriale du Site du Zéro.

Par  Michaël Haberzettel (devil may cry)

Mise à jour : 12/05/2012
Difficulté : Difficile

Bonjour à tous,
Voici un tutoriel qui a pour objectif d'aider les personnes à mieux cerner ce qui se cache derrière la programmation orientée objet. Moi-même venant d'un langage impératif, il a été plutôt difficile de cerner les penchants et les aboutissants de cette philosophie. Après un temps d'adaptation on commence à comprendre les mécanismes et comment les objets doivent se lier entre eux. C'est alors en persévérant et en se renseignant sur diverses sources internet (une grande partie imputable à developpez.com) que l'on se rend compte qu'il faut adopter une certaine rigueur et des bons principes pour avoir un code robuste, suffisamment robuste pour qu'une adaptation ne bouleverse pas tout le code en cascade.

C'est en glanant principalement des codes créés par les gens de mon entourage et des divers post sur les forums que je me suis décidé à créer ce tutoriel : Apprendre aux gens ce qu'est le paradigme objet (son contexte) pour mieux concevoir vos futurs programmes. Après avoir montré **quelques principes essentiels assez peu soulignés** sur le net, je m'arrêterai sur certains design patterns (si vous ne connaissez pas, je vous invite à lire mon introduction) pour vous montrer ce que peuvent donner de bonnes conceptions.

Concernant le tutoriel en lui-même, voici les prérequis :

- Savoir programmer un minimum en orienté objet (hormis pour les 2 premiers chapitres).
- Connaitre la lecture de diagrammes de classes UML.
- Connaitre la syntaxe d'un langage se rapprochant du C++ / Java / C#

Si les termes comme *instance*, *composition*, *héritage*, *polymorphisme* vous sont inconnus, vous n'êtes pas prêt à apprendre ce qui va suivre, renseignez-vous d'abord sur ces notions et soyez sûr de comprendre de quoi il en retourne.

A ce propos, le tutoriel utilisera du code C++ et du code Java Ce sont 2 langages très populaires qui me permettent de toucher quasiment toutes les personnes du monde objet. Cependant, vous ne me verrez pas utiliser les spécificités pointues de chaque langage, le but étant **d'écrire du code suffisamment générique pour le transposer dans un autre langage objet** ! L'objectif principal est de privilégier **la conception**, aussi je vais essayer de produire un code reproductible aux 3 langages cités dans les besoins du tutoriel et je placerai ponctuellement des commentaires sur les fonctionnalités disponibles sur certains d'entre eux.

Avant toute chose, je tiens à remercier lmghs pour sa relecture attentive et ses différents conseils.

Si des personnes souhaitent m'aider à la rédaction du tutoriel, n'hésitez pas à me contacter. Toute aide est la bienvenue.

Vous êtes prêt ?

Partie 1 : Le paradigme objet

Introduction à ce cours

En guise d'introduction sur ce cours, je vais tenter de vous sensibiliser sur les problèmes récurrents lorsqu'on aborde la partie programmation. Il est important de bien comprendre que programmer de manière objet est une tâche qui n'est pas simple, nombre de personnes qui se targuent de coder en objet n'appliquent pas correctement sa philosophie (on parle du paradigme objet). Comme ce cours se veut pour des zéros, je vais survoler des principes à la gestion de projets complexes qui peuvent s'appliquer à des projets de très petites envergures.

Pour aborder ce cours, j'estime qu'il est nécessaire d'avoir un minimum de pratique en programmation en s'étant déjà "cassé les dents" sur diverses difficultés. Vous pourrez cependant trouver la pensée à adopter pour concevoir en orienté objet. Ces notions peuvent se révéler très intéressantes surtout pour les nouveaux qui ont une vision issue des langages impératifs.

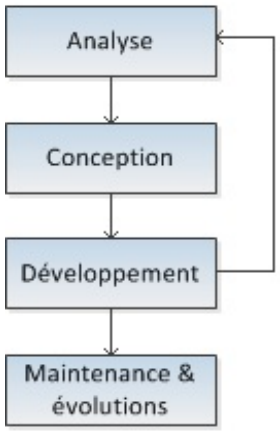
Cycle simplifié d'un projet

Lorsqu'on acquiert suffisamment d'expérience en programmation orienté objet, on constate que pour un problème donné, plusieurs solutions d'implémentations s'offrent à nous. Il est parfois difficile de trouver parmi ces solutions, laquelle offrira un maximum d'avantages sans apporter d'inconvénients. Malgré la simplicité apparente offerte par l'approche objet, certaines solutions qui semblent efficaces au départ se retrouvent désuètes si une évolution doit être apportée.

Ne vous est-il jamais arrivé de vous prendre la tête parce que votre code se révèle finalement peu flexible lorsque vous souhaitez rajouter une possibilité inédite à votre programme ? La relecture du code qui peut sembler évidente au premier abord, peut devenir difficile si des jours se sont écoulés entre temps. Partant de ce constat, il y a un postulat qu'il ne faut pas négliger :

La programmation orientée objet nécessite un minimum de conception pour voir sa réutilisabilité accrue. Je vais même jusqu'à dire que c'est même une étape indispensable pour un projet qui commence à prendre un minimum d'ampleur.

J'insiste sur ce point car on aperçoit que la plupart des problèmes des codes développés en objet viennent de complications soulevées en amont que l'on a voulus reporter à l'étape de codage. Il ne faut pas se leurrer, nous pratiquons ce report tous à plus ou moins grande échelle afin d'arriver le plus rapidement à la programmation. Cependant, c'est une mauvaise manière de faire car établir un programme se base sur le cycle (simplifié) suivant :



Cycle de développement simplifié.

Notez que cette gestion de projet s'inspire du cycle incrémental (mais simplifié). C'est la gestion la plus intuitive que l'on mène dans un projet. C'est à dire qu'on s'attaque à des parties spécifiques (un module, une fonctionnalité) que l'on va résoudre et développer avant de passer à un autre problème (la flèche qui repart du développement vers l'analyse). Réaliser le projet de la sorte permet de tester son programme au fur et à mesure de son avancement et de valider ou changer la partie concernée. Même si d'autres cycles de gestion existent (spirale, en V, méthodes agiles, ...), tous contiennent au moins les phases présentées dans ce tutoriel. Mon discours reste valable dans les grandes lignes pour ces modèles.

Ce modèle emprunté au cycle de gestion d'un projet de grande envergure est intéressant sur plusieurs points mais comme il faudrait un cours complet pour expliquer ses principes, nous nous attarderons seulement sur les grandes phases. Ces phases se suivent dans un ordre bien précis qu'il est pratiquement impossible de faire autrement même lorsque vous écrivez le plus petit programme qu'il soit (sauf si vous êtes un prestataire avec un cahier des charges, ce qui doit être le cas de 0% des zéros) !

Attends une seconde, comment ça on respecte ce cycle ? Puis d'abord j'y comprends rien à ton schéma !

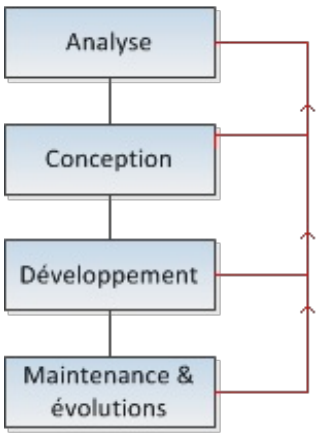
Pas trop vite ! Décrivons ces phases sommairement pour savoir à quoi elles correspondent réellement :

- **Analyse** : C'est la phase qui s'occupe de poser les problématiques générales au projet et d'y répondre. Ici, on n'aborde pas l'aspect programmation.
- **Conception** : Une phase tout aussi importante, elle précède la programmation réelle en s'attardant sur "comment modéliser" tel problème dans mon code.
- **Développement** : C'est le code produit en fonction des deux premiers points, c'est donc une étape qui doit être considérée avec autant de poids que les autres.
- **Maintenance & évolutions** : Un point qui peut paraître secondaire mais il n'en est rien. C'est ce dernier qui le plus consommateur en temps. Les raisons en seront évoquées juste après.

Si vous ne voyez pas le rapprochement avec le code que vous vous amusez à écrire, prenons un cas concret à la mode sur le site du zéro : le RPG et décortiquons. Comment va se structurer votre pensée si vous vous fixez comme objectif de coder un RPG simple en mode console ? Logiquement vous allez vous poser les questions suivantes :

- Que va devoir gérer mon RPG (personnages, armes, magies, ...) ? Quelles sont les limites que je m'impose ?
- Avec cette "analyse", vous arrivez à entrevoir les classes correspondantes et leurs interactions avec d'autres classes.
- Vous lancez votre éditeur de code favori et vous commencez à développer en vous basant sur votre pensée précédente.
- Une fois que vous avez réussi à programmer une partie spécifique (exemple : le personnage avec toutes ses interactions), vous vous attardez sur un élément à appréhender.
- Vous jonglez constamment sur les 3 points précédents lorsque vous écrivez du nouveau code (ce qui s'apparente aux évolutions dans notre cas).

Intrinsèquement vous respectez donc (presque) toujours ce cycle même si c'est de manière implicite. Vous vous demandez encore sûrement pourquoi je vous ai balancé ce schéma. Bien, venons-en aux faits : Que se passe-t-il lorsque vous programmez et que vous tombez sur un problème. Le problème du genre (en reprenant l'exemple d'un RPG) :
"J'ai un magicien qui hérite de personnage et un barbare qui hérite de personnage. Je propose dans mon jeu de pouvoir créer un personnage avec 2 catégories, il faut donc créer un magicien barbare qui hérite de magicien et de barbare. J'ai maintenant une classe voleur (une nouvelle fonctionnalité fraîchement rajoutée dans le jeu), je vais maintenant créer les couples magicien voleur et voleur barbare."
Le problème se profile très clairement, que va t'il se passer si je décide d'ajouter un nouveau type de personnage comme par exemple assassin ? Il faudra créer les classes assassin voleur, magicien assassin, barbare assassin. C'est une solution peu flexible et qui augmente le nombre de classes très rapidement. Cette erreur va vite devenir un cauchemar, jusqu'à ce que vous décidiez de changer votre manière de faire en évitant de reproduire l'erreur. Vous refaites alors de la conception pour corriger cette lacune. Voyons quels sont les impacts ce mode de fonctionnement sur mon schéma :



Les flèches rouges représentent les répercussions que peuvent entraîner une erreur selon la phase. Par exemple, une erreur à

l'étape du développement nécessite de revoir la conception. Revoir la conception peut impliquer de revoir aussi l'analyse.

Que pouvons-nous tirer de ce constat ? Plusieurs points intéressants qui sont :

- Plus la source du problème se dévoile dans les phases en amont, plus les répercussions seront grandes. En l'occurrence ici, le problème se situe soit au niveau de l'analyse avec le raisonnement qu'au début il n'était pas possible de pouvoir créer des personnages de multiples natures. Soit il peut s'agir d'un problème conceptuel car on souhaitait offrir cette possibilité dès le départ, mais par manque de technique et/ou réflexion nous nous retrouvons dans l'impasse.
- Certaines modifications inhérentes à une mauvaise conception peuvent se révéler douloureuses entraînant probablement d'autres correctifs non souhaités.
- Il y a une perte conséquente de temps à pallier ce genre d'imprévus.

Si vous devez retenir une chose de tout ce qui a été dit cela se formulerait par :

Il est important de prendre du temps sur l'analyse et surtout (pour le cas des zéros qui programment par plaisir) **de consacrer des ressources sur la conception** !

De très nombreux maux sur les forums de ce site sont dus à une mauvaise conception. Apprenez à bien concevoir et vous vous rendrez très vite compte des effets bénéfiques que vous pourrez en tirer. La bonne conception vient aussi en pratiquant, planchez vous sur des problèmes en essayant différents angles d'approches et mesurer les avantages/inconvénients sur chacune pour en tirer un maximum de bénéfice.

La conception, votre meilleure amie

Prenez pour acquis que plus vous passerez du temps sur la conception plus le développement s'en retrouvera simplifié, efficace et donc amoindri d'erreurs. Cependant la tâche n'est pas si simple que je voudrais le faire croire car concevoir est une tâche plutôt ardue qui nécessite de la réflexion et surtout, du recul. Ce recul correspond à adopter un maximum d'abstractions pour résoudre un obstacle. Plus la solution aura un taux d'abstraction élevé plus vous pourrez appliquer ce même concept pour un autre problème similaire.

Il est très difficile de répondre à une problématique donnée en s'imaginant la meilleure alternative dans sa tête. Le plus simple étant de prendre une feuille et de modéliser diverses solutions avec des diagrammes. J'utiliserai à partir de maintenant des diagrammes de classes pour argumenter mes propos.

De nombreux informaticiens très expérimentés se sont arrachés les cheveux sur certaines problématiques et ont offerts diverses alternatives. Par la suite, des chercheurs conscients que la conception étant une tâche ardue ont commencé à proposer des recettes suffisamment abstraites pour qu'on puisse les réutiliser dans des circonstances similaires. Ces aboutissements portent le nom de "patrons de conceptions" (design patterns en anglais).

Très tôt dans la programmation orientée objet, des personnes se sont intéressées à la construction de ces design patterns. Les principaux qui seront traités dans ce cours sont issus du GoF (Gang of Four), une équipe de 4 personnes qui a établi une publication de 23 patrons de conceptions dans un livre de 1994 devenu une référence. Depuis, il existe d'autres patrons de conceptions, mais ils sont un peu moins connus (bien que certains soient très utiles).

Mais attend ! S'il existe de telles recettes pour faciliter la conception, ça va maintenant devenir un jeu d'enfants de surmonter mes problèmes non ? A moi la gloire, le futur jeu de la mort, les femmes !

Pas si vite ! Ces outils n'ont pas cette prétention (même si j'aurais préféré que ce soit le cas) .
Certes dans des cas précis, ils vous faciliteront la tâche car vous n'aurez qu'à utiliser le patron correspondant mais : Il faut d'abord comprendre ce patron de conception (Dites-vous qu'il y a une abstraction élevée dans un design pattern et que ce n'est pas si instinctif que ça pour l'implémenter à son cas). Aussi, il est important de savoir si pour le problème auquel on se heurte, il existe le design pattern correspondant (ici non plus ce n'est pas aussi simple que ça en a l'air de savoir si toutes les conditions sont réunies). Même en ayant connaissance de ces éléments, vous passerez la majorité de votre temps à concevoir de manière classique sans vous référer à ces recettes. N'attendez pas de ce "catalogue" le Saint Graal de la programmation !

Mais alors c'est nul ton truc, ça a l'air prise de tête pour pas grand-chose finalement je me trompe ?

Avant que celui du fond ne sorte la cagette de tomates, prenez le temps de lire ce qui suit. Je ne vous le cache pas, c'est un investissement que de passer à apprendre ces patrons de conception, on peut cependant en tirer plusieurs bénéfices notables. Si vous trouvez une partie de votre code qui exige un design pattern voilà les gains apportés :

- Une documentation facilement accessible sur le design pattern utilisé. Si vous même ou quelqu'un d'autre ayant connaissance de ce qu'est un design pattern, il n'y aura pas besoin de commenter de manière complexe le code. Au pire, si la personne qui tombe sur votre code ne connaît pas les patrons de conception, il n'aura que l'apprentissage de celui concerné. C'est un gain de temps notable de tous les côtés et ça évite les prises de têtes (pour vous y compris).
- Votre code (si vous avez correctement implémenté le patron) est ouvert aux apports et sans contrepartie pour la solution que vous avez employée au problème posé. Vous pourrez donc très facilement faire évoluer le code sans risquer de devoir tout modifier.

En conclusion de cette partie, si vous devez retenir une chose c'est que les patrons de conceptions ne sont pas indispensables pour bien concevoir, mais pour une problématique donnée, ils y répondent d'une manière élégante (à leur époque).

Évidemment, ce ne sont pas les seules recherches qui ont fait progresser l'orienté objet, de très grands auteurs comme James O. Coplien ont offert d'autres approches pour concevoir efficacement. Vous verrez un tour d'horizon des principales idées dans le prochain chapitre des influences portées par ces experts.

Comment bien concevoir

Plusieurs ingrédients essentiels doivent être connus en amont pour réussir une bonne conception. Tout d'abord, l'analyse : elle est une partie quasiment obligatoire afin de savoir quelles sont les fonctionnalités de la future application (jeux, outils, applications bureautiques...). Cette distinction qui ne vous semble pas nécessaire prend un intérêt énorme pour "segmenter" son application en modules indépendants (en termes de code, pas forcément en termes de données). Favoriser des découpages en modules est la première clef d'une conception souple. On réduit les problèmes de manière plus locale, plus ciblés. Parvenir à séparer son application en module n'est pas une chose facile et les diagrammes des cas d'utilisations peuvent constituer une excellente première approche :

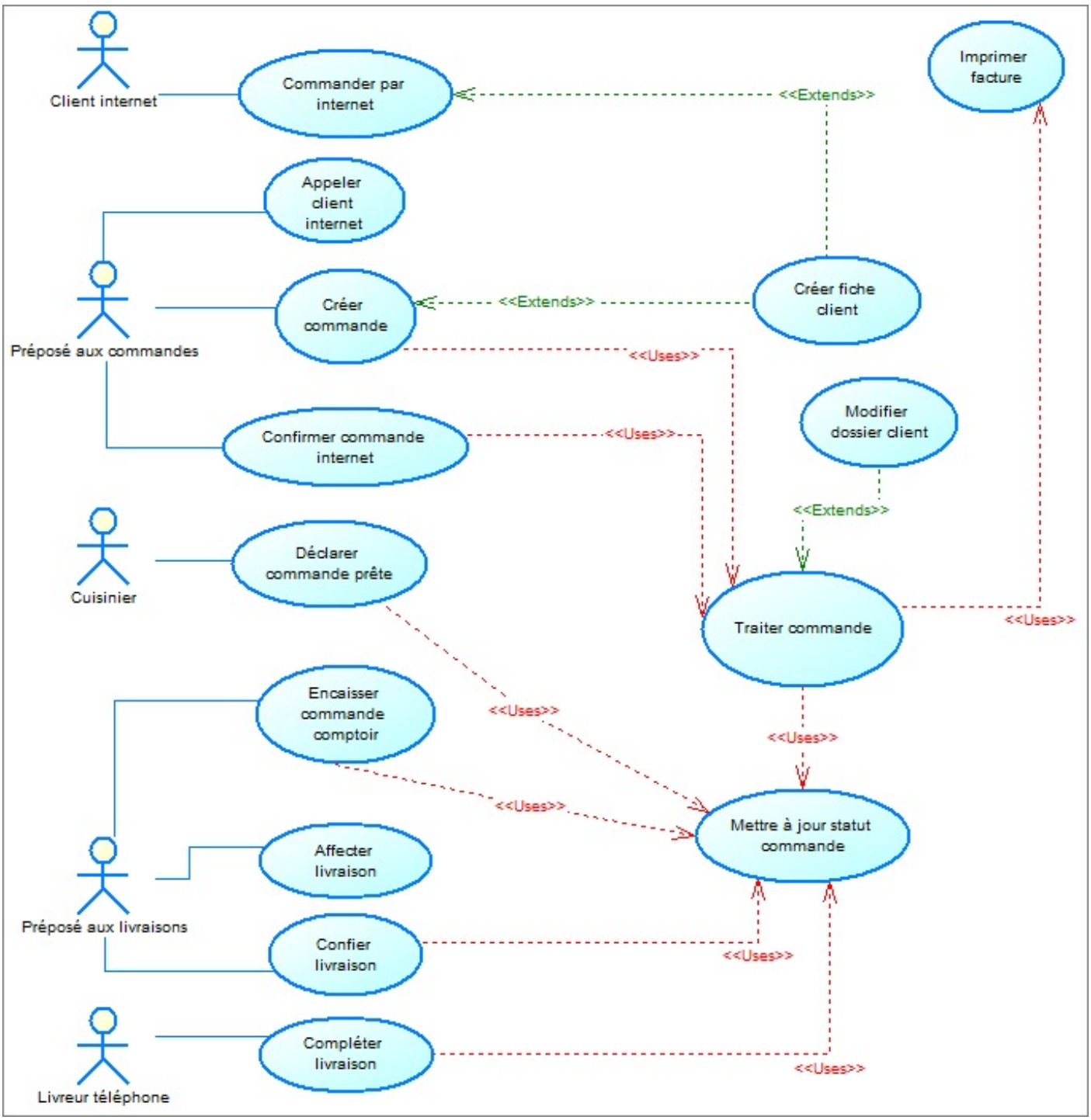


Diagramme de cas d'utilisations d'une pizzeria. Ne vous attardez pas dessus vous ne connaissez pas le contexte. Rappelons juste que ce diagramme permet de représenter les fonctionnalités auxquelles l'application doit répondre.

Notez que grâce à ce schéma on peut déjà connaître beaucoup de choses sur l'application. Il est possible d'entrevoir un découpage, des traitements et les différentes interfaces hommes machines que l'on devra créer selon le profil utilisateur. Je ne vais pas expliquer comment modéliser ce diagramme mais il ne faut pas oublier qu'il est toujours accompagné d'un descriptif. Nous sommes plus doués pour se représenter les choses par des dessins que par du simple texte, ne sous-estimez donc pas l'importance de passer par cette modélisation visuelle préalable.

D'autres diagrammes tels que le diagramme de séquence, le diagramme d'activité ou encore le diagramme d'états-transitions offrent par la même occasion une aide considérable pour réaliser une conception qui reflète au mieux ce que vous avez cerné. Le plus intéressant lorsqu'on modélise, c'est que sur certains points, on commence à buter et ce sont généralement ces parties-là qui vont vous causer du tort plus tard. Sauf qu'ici, l'impact de ces soucis a des conséquences moindres que si on s'était déjà lancé dans le développement. Si vous démarrez de nouveaux projets qui commence à avoir de l'ampleur, il est **vivement conseillé** de faire une modélisation préalable et **encore plus si vous travaillez en équipe**.

Toutes ces étapes qui précèdent le développement peuvent vous sembler superflues mais c'est tout le contraire. A

défaut de perdre du temps à l'élaboration de l'analyse, vous gagnerez un temps considérable par la suite car la majorité des problèmes auront été anticipés. Toute forme quelconque de texte ou de schéma sert aussi pour créer une documentation solide, dans une autre mesure ces éléments
peuvent être des constituants d'un cahier des charges.

Le second point pour concevoir efficacement est de connaître ce qui se cache derrière le nom de programmation orientée objet, sa raison d'être et comment elle devrait être utilisée. Ce que l'on nomme plus scientifiquement "le paradigme objet". C'est ce sur quoi le prochain chapitre va porter.
J'espère que cette petite introduction vous aura mis l'eau à la bouche. Si vous êtes motivé pour apprendre ce qui suit ou simplement curieux, continuez la lecture, vous ne serez pas déçus. Cependant, avant de se lancer dans l'explication de certains patrons de conception, il me paraît indispensable de vous poser les bases de l'objet. Je n'ai malheureusement pas vu sur le site du zéro un cours s'y rapprochant, je vais donc faire un rapide aparté afin de mieux comprendre comment les patrons de conceptions ont été conçus.

Je pense que vous aborderez la vision de l'objet sous un autre angle à la fin du chapitre suivant si certains des principes évoqués vous étaient méconnus. Si vous êtes toujours là, allons-y !

Adoptez la philosophie de l'objet

Pour bien commencer à concevoir en objet, il faut d'abord s'imprégner des concepts liés à sa philosophie. C'est en prenant pour acquis ces points et en les appliquant que vous pourrez prétendre programmer et penser objet. Ce chapitre peut paraître superflu pour une majorité d'entre vous, il en reste cependant très instructif sur la suite. En effet, les patrons de conception respectent intimement ces principes.

Ne voyez pas dans la programmation orientée objet, un simple rajout de méthodes dans les structures (pour les personnes ayant l'habitude de faire du C). C'est une vision erronée du paradigme objet.

Venons-en maintenant aux faits, si vous êtes toujours là et que les parties suivantes seront nouvelles pour vous, j'ose dire que vous ne verrez plus l'orienté objet de la même façon. C'est parti !

Les codes que je vous présenterai sont donnés à titre d'exemple et pas forcement compilables. Ne soyez donc pas étonnés si vous trouvez que certaines portions sont manquantes, je m'intéresse uniquement à la partie utile du code. Cette remarque est d'autant plus vrai pour les différents exemples donnés en code Java afin d'éviter l'alourdissement du chapitre.

Offrez du service

Le point que je considère comme l'un des plus importants en programmation orientée objet est probablement sur les opérations que proposeront vos classes. Lorsque vous concevez une nouvelle classe qui s'avère complexe dans ses traitements, la première chose que nombre d'entre vous faites est de créer des getters / setters sur vos variables afin de pouvoir y accéder en dehors de la classe. Ce n'est pas obligatoirement une mauvaise chose mais dans la plupart des cas c'est à proscrire.

Pour construire efficacement les méthodes d'accès publiques, il faut penser différemment; Adopter une approche en termes de services rendus.

Mais qu'appelles-tu un service ?

C'est simple, prenons un exemple. Je viens de créer ma classe, il faut simplement imaginer qu'est-ce que peut proposer ma classe aux autres ayant un intérêt. Ou avec du code :

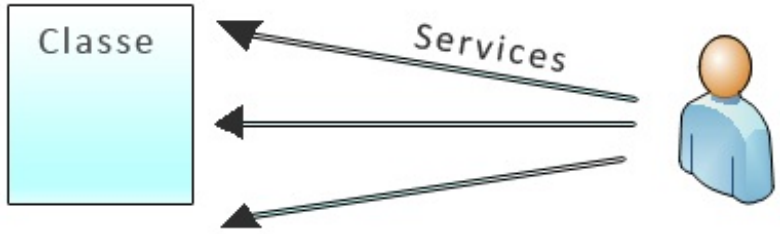
Code : C++

```
MaClasse instance;
instance. // Les méthodes utiles aux utilisateurs de cet objet.
```

Code : Java

```
MaClasse instance = new MaClasse();
instance. // Les méthodes utiles aux utilisateurs de cet objet.
```

Ce qui peut se schématiser par :



Pour établir des services efficaces, il faut se positionner en tant qu'utilisateur extérieur de la classe et définir les méthodes qui lui seront utiles.

C'est donc de construire sa classe depuis l'extérieur sans réfléchir à ce qu'elle contient. En prenant un exemple extrême, ma classe peut contenir 100 variables privées utiles à son bon fonctionnement (pour des raisons de performance par exemple), si une seule méthode est utile pour les utilisateurs de cette classe, alors pas besoin d'offrir des méthodes d'accès inutiles.

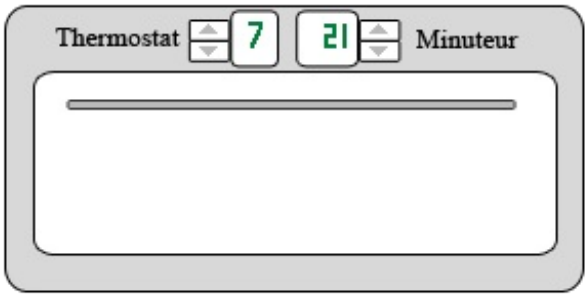
Si vous n'arrivez pas à réaliser efficacement une construction de méthodes avec des services, vous pouvez simplement procéder de la manière suivante :

Code : C++

```
class NouvelleClasse
{
     // Avant n'importe quel code, on va construire toutes les méthodes qui nous semblent indispensables.
     // pour les utilisateurs extérieurs
};
```

Ce n'est pas plus compliqué et vous améliorerez grandement la compréhension de votre code que si vous ne réfléchissiez pas en termes de service. Essayez juste de raisonner simplement pour offrir des services précis et concis.

Pour faire un rapprochement avec la réalité, vous pouvez prendre un four électrique :



Prière de ne pas trop rire de mes talents artistiques.

Seuls quatre boutons sont accessibles pour préparer le four : ce sont les services proposés à l'utilisateur. Pourtant le four doit gérer des choses en interne comme par exemple, s'allumer (si le minuteur est au dessus de 0), mesurer constamment la température pour respecter le thermostat indiqué, décrémenter le compte à rebours ou vérifier que l'utilisateur ne rentre pas un thermostat ou un nombre de minutes en dehors des bornes autorisées. Si cette représentation peut vous aider à réfléchir en terme de services, alors retenez ce moyen mnémotechnique, il représente bien le concept. On parle aussi de construction de sa classe par l'extérieur et non pas par l'intérieur.

Si vous voulez des exemples de services proposés avec une classe très connue, vous pouvez vous inspirer de **string**. En C++, comme en Java ou autre, l'utilisateur de la classe ne se soucie pas de sa gestion en interne (la classe string possède généralement de nombreuses optimisations cachées aux yeux de l'utilisateur). Vous n'avez accès qu'à des méthodes qui camouflent ces détails de bas niveau en augmentant ainsi l'abstraction pour la manipulation de chaînes. Pas la peine de se le cacher, il y a un coût potentiel de performance par rapport à de la gestion pure de chaine mais vous améliorez grandement en gain de temps et maintenance. Sauf si c'est une partie critique de votre code, favorisez une certaine abstraction, vous y gagnerez sur tous les plans.

Plus d'informations à ce sujet :
- Ce principe qui va de paire avec l'encapsulation présentée juste après porte l'acronyme LoD ou s'appelle loi de Déméter.
- Article succinct sur Wikipédia : loi de Déméter.

L'encapsulation règne en maitre !

L'encapsulation c'est facile, il suffit de placer toutes ses variables en accès privé ou protégé et d'encapsuler ces variables par des accesseurs/mutateurs (getters/setter)! Si c'est le réflexe qui vous vient en premier lieu, vous êtes dans le faux, faisons le tour d'horizon de ce raisonnement.

Pour commencer, je vais évoquer la raison précédente : la classe doit être construite pour offrir des services : certaines variables n'ont donc pas d'utilité à être connues de l'extérieur, sauf si vous voulez obscurcir son usage et sa documentation. Ce qui peut être intéressant, c'est de donner un accès en lecture seulement sur certaines propriétés d'un objet à un instant "t". On se retrouve dans ce cas à créer un getter afin d'empêcher toute corruption de l'état d'un objet. L'autre cas utile qui peut justifier de créer un couple de getter/setter sur une propriété est la mise en place de contrôles sur le setter. Ces remarques découlent du bon sens, on ne crée que ce qui est **nécessaire** aux utilisateurs de la classe.

Le cas des getters/setters vidés de contrôles :

Soit le code :

Code : C++

```
class Personne
{
  public :
    const std::string& getNom() const { return m_nom }
    void setNom(const std::string &nouveauNom) { m_nom = nouveauNom;
}

  private :
    std::string m_nom;
};
```

Code : Java

```
class Personne
{
  public String getNom() { return nom; }
  public void setNom(String nouveauNom) { nom = nouveauNom; }

  private String nom;
}
```

Le seul cas où il pourrait être judicieux d'écrire du tel code, c'est lorsque vous suspectez une possible évolution des contrôles qui pourront être appliqués sur le nom. Par exemple, que la méthode setNom vérifie que le nom passé en paramètre ne soit pas vide. Quand le doute est permis, n'hésitez pas à encapsuler la propriété, cela pourrait éviter une refactorisation importante du code. Cependant, ne vous leurrez pas, si vous faites un couple getter/setter comme au-dessus sans tenir compte de la règle (c'est le cas des conventions Java par exemple), vous offrez un accès direct à votre variable membre, ce qui la rend indirectement publique.

Voyons ce que peut donner l'utilisation de ces bonnes pratiques :

Code : C++

```
// Le code suivant va utiliser le mot clef struct en référence à
une structure en c++.
// Cela équivaut à une classe hormis que les attributs et méthodes
sont publics par défaut.

/** Définition d'un point avec une position en X et Y en tant que
structure.
* Pourquoi s'embêter à placer des getters/setters inutiles qui
n'apporteront jamais
* de valeur ajoutée aux attributs publics.
*/
struct Point
{
   Point(int ptX, int ptY) { x = ptX; y = ptY; }
   int x, y;
};

/** Classe permettant de représenter une couleur selon les
composantes Rouge, Vert, Bleu
* et le canal alpha (la transparence)
*/
class Couleur
{
  public : // Tout ce qui suit est en accès publique

   Couleur (int rouge, int vert, int bleu, int alpha = 255) :
      m_r(corrigerIntervalleComposante(rouge)),
      m_v(corrigerIntervalleComposante(vert)),
      m_b(corrigerIntervalleComposante(bleu)),
      m_alpha(corrigerIntervalleComposante(alpha))
   {
```

```
    }


    // Comme il y a un contrôle sur les composantes, l'usage de
getters/setters semble justifié
    int getRouge() const { return m_r; }
    int getVert() const { return m_v; }
    int getBleu() const { return m_b; }
    int getAlpha() const { return m_alpha; }

    void setRouge(int valeur) { m_r =
(corrigerIntervalleComposante(valeur)); }
    void setVert(int valeur)  { m_v =
(corrigerIntervalleComposante(valeur)); }
    void setBleu(int valeur)  { m_b =
(corrigerIntervalleComposante(valeur)); }
    void setAlpha(int valeur) { m_alpha =
(corrigerIntervalleComposante(valeur)); }


  private : // Tout ce qui suit est réservé à la classe ou ses
dérivées
    int corrigerIntervalleComposante(int valeurComposante)
    {
       if(valeurComposante > 255) // Une constante aurait été plus
judicieuse mais inutile pour l'exemple
          return 255;
       else if(valeurComposante < 0)
          return 0;
       return valeurComposante;
    }
    int m_r, m_v, m_b, m_alpha;
};

class Image
{
  public :
    static bool charger(string nomFichier);
    const Point& getTailleImage() const; // Pour les gens du monde
C++, le mot clef const très important

    // Par exemple, la méthode du dessous n'a pas lieu d'être
    void setTailleImage(const Point &nouvelleTaille);

    // Un autre exemple d'accès à une valeur en lecture seule :
    const Couleur& getPixelCouleur(const Point &pixelConcerne) const;

    // ....

  private :
    Point m_tailleImage;
    Couleur **m_matricePixels; // Tableau en 2 dimensions à allouer
dynamiquement
                               // On pourrait utiliser un std::vector
de std::vector
                               // ou la classe multi_array de la
bibliothèque boost
    // D'autres attributs

};
```

Code : Java

```
/** Définition d'un point avec une position en X et Y en tant que
structure.
* Pourquoi s'embêter à placer des getters/setters inutiles qui
n'apporteront jamais
* de valeur ajoutée aux attributs publics.
```

```
*/
class Point
{
   public Point(int ptX, int ptY) { x = ptX; y = ptY; }
   public int x, y;
}

/** Classe permettant de représenter une couleur selon les
composantes Rouge, Vert, Bleu
* et le canal alpha (la transparence)
*/
class Couleur
{
   public Couleur (int rouge, int vert, int bleu, int alpha)
   {
      m_r = corrigerIntervalleComposante(rouge);
      m_v = corrigerIntervalleComposante(vert);
      m_b = corrigerIntervalleComposante(bleu);
      m_alpha = corrigerIntervalleComposante(alpha);
   }

   // Composante alpha fixée à 255
   public Couleur (int rouge, int vert, int bleu)
   {
      this(rouge, vert, bleu, 255);
   }


   // Comme il y a un contrôle sur les composantes, l'usage de
getters/setters semble justifié
   public int getRouge() { return m_r; }
   public int getVert()  { return m_v; }
   public int getBleu()  { return m_b; }
   public int getAlpha() { return m_alpha; }

   public void setRouge(int valeur) { m_r =
(corrigerIntervalleComposante(valeur)); }
   public void setVert(int valeur)  { m_v =
(corrigerIntervalleComposante(valeur)); }
   public void setBleu(int valeur)  { m_b =
(corrigerIntervalleComposante(valeur)); }
   public void setAlpha(int valeur) { m_alpha =
(corrigerIntervalleComposante(valeur)); }


   private int corrigerIntervalleComposante(int valeurComposante)
   {
      if(valeurComposante > 255) // Une constante aurait été plus
judicieuse mais inutile pour l'exemple
         return 255;
      else if(valeurComposante < 0)
         return 0;
      return valeurComposante;
   }
   private int m_r, m_v, m_b, m_alpha;
}

class Image
{
  public static boolean charger(string nomFichier);
  public Point getTailleImage();

   // Par exemple, la méthode du dessous n'a pas lieu d'être
   public void setTailleImage(Point nouvelleTaille);

   // Un autre exemple d'accès à une valeur en lecture seule :
   public Couleur getPixelCouleur(Point pixelConcerne);

   // ....
```

```
  private Point m_tailleImage;
  private Couleur[][] m_matricePixels; // Tableau en 2 dimensions
   // D'autres attributs

}
```

Ce code est déjà un bon exemple des services rendus, si vos classes proposent les bons services, les classes peuvent voir leur implémentation interne complétement changée sans aucune incidence sur le code. Imaginons maintenant que l'application est un logiciel de retouche d'images, il est crucial d'optimiser la place mémoire, on peut améliorer les classes existantes sans impacter le service offert, voyez plutôt :

Code : Java

```
// Utilisation des opérateurs de bits. L'utilisateur de la classe
n'en a que faire !
// Cependant, une couleur prend 4 fois moins d'espace mémoire
qu'avant. il y a cependant un traitement à faire.
class Couleur
{
   public Couleur (int rouge, int vert, int bleu, int alpha)
   {
   m_stockageBinaire = 0;
      m_stockageBinaire |= corrigerIntervalleComposante(rouge) <<
24;
      m_stockageBinaire |= corrigerIntervalleComposante(vert)<< 16;
      m_stockageBinaire |= corrigerIntervalleComposante(bleu)<< 8;
      m_stockageBinaire |= corrigerIntervalleComposante(alpha);
   }

   public Couleur (int rouge, int vert, int bleu)
   {
    this(rouge, vert, bleu, 255);
   }


   // Comme il y a un contrôle sur les composantes, l'usage de
getters/setters semble justifié
   public int getRouge() { return m_stockageBinaire >>> 24; }
   public int getVert()  { return m_stockageBinaire << 8 >>> 24; }
   public int getBleu()  { return m_stockageBinaire << 16 >>> 24; }
   public int getAlpha() { return m_stockageBinaire << 24 >>> 24; }

   public void setRouge(int valeur) {  m_stockageBinaire -=
getRouge()<<24; m_stockageBinaire +=
(corrigerIntervalleComposante(valeur)) << 24; }
   public void setVert(int valeur)  {  m_stockageBinaire -=
getVert()<<16; m_stockageBinaire +=
(corrigerIntervalleComposante(valeur)) << 16; }
   public void setBleu(int valeur)  {  m_stockageBinaire -=
getBleu()<<8; m_stockageBinaire +=
(corrigerIntervalleComposante(valeur)) <<8; }
   public void setAlpha(int valeur) {  m_stockageBinaire &=
~getAlpha(); m_stockageBinaire +=
(corrigerIntervalleComposante(valeur)); }


  private int corrigerIntervalleComposante(int valeurComposante)
   {
      if(valeurComposante > 255) // Une constante aurait été plus
judicieuse mais inutile pour l'exemple
         return 255;
      else if(valeurComposante < 0)
         return 0;
      return valeurComposante;
   };
   private int m_stockageBinaire;
}
```

Pour la petite anecdote, la classe **Couleur** aurait pu être optimisée avec le code suivant (mais disponible que sur certains langages comme le C++ qui possède des types non signés).

Code : C++

```
// En s'assurant que le char soit codé en 8 bits (256 valeurs) :
struct Couleur
{
   Couleur (unsigned char rouge, unsigned char vert,
            unsigned char bleu, unsigned char canalAlpha = 255)
   {
       r = rouge;
       v = vert;
       b = bleu;
       alpha  = canalAlpha;
   }

   unsigned char r, v, b, alpha;
};
```

Je rappelle que ce tutoriel n'a pas la prétention d'optimiser chaque ligne de code, surtout que ces différences sont issues des particularités de langages généralement.

En conception pure, la première classe Couleur codée est valide, n'oubliez pas que le langage vient normalement se greffer après et c'est là que viennent les optimisations. Noter aussi qu'au lieu de pondre ces solutions, il existe un design pattern qui permet de réduire énormément la place que peuvent prendre des petits objets identique (c'est le cas avec les couleurs d'une image). Ce patron de conception s'appelle poids mouche ou flyweight et sera abordé plus tard.

Il est utile de constater que les classes qui servent uniquement de structures de données brutes se passent très généralement du couple getter/setter. En ce sens, le C++ offre le mot clef **struct** qui offre une meilleur sémantique que le Java par exemple.

Ouf, pour résumer, voilà comment on pourrait formuler la chose (je vais ré-insister pour être sûr qu'il n'y ai pas de malentendu) :

- Lorsqu'on construit sa classe, on va proposer des services.
- Certains de ces services correspondent à l'accès en lecture des propriétés de la classe.
- On regarde ensuite s'il y a des propriétés qui doivent être en écriture dans les services proposés.
- Dans le cas de structures de données, la variable membre devrait être publique sinon on écrit l'accesseur et/ou mutateur correspondant.

Si vous n'êtes pas convaincu que les accesseurs/mutateurs ne sont pas toujours une bonne solution, beaucoup de sujets traitent de ce problème sur internet. Une simple recherche avec l'expression "why getters setters are evil" vous donnera une foulée d'arguments qui se tiennent. Voilà un exemple qui pourra vous faire changer d'avis sur l'utilisation non réfléchie des setters :

Code : C++

```
class Compteur
{
  public :
    Compteur(unsigned int tickDepart = 0);
    unsigned int getTick() const ;
    void setTick(unsigned int nouvelleValeur);
    void reinitialiser();

  private :
   unsigned int m_cpt;
};
```

Code : Java

```
class Compteur
{
   public Compteur();
   public Compteur(int tickDepart);
   public int getTick();
   public  void setTick(int nouvelleCaleur);
   public  void reinitialiser();

  private int m_cpt;
}
```

Ce code présente deux inconvénients majeurs. Tout d'abord, il révèle en quelque sorte la structure interne de la classe Compteur car on peut y déceler les méthodes get/set correspondantes, rappelez-vous, l'utilisateur n'a pas à faire de suspicion sur les rouages internes de la classe. Le second problème plus grave est que le setter offre une trop grande liberté à l'utilisateur, un compteur ne doit pas offrir la possibilité de changer sa valeur lorsqu'il... compte ! Une implémentation largement plus claire qui corrigerait ces deux défauts pourrait être :

Code : C++

```
class Compteur
{
  public :
    // Rajout d'une variable pas pour plus de souplesse
    Compteur(unsigned int tickDepart = 0, unsigned int pas = 0);
    unsigned int tickActuel() const ;
    unsigned int incrementer(); // Incrémente du pas et retourne la
nouvelle valeur
    void reinitialiser();

  private :
   unsigned int m_cpt, m_pas;
};
```

Code : Java

```
class Compteur
{
   public Compteur();
   public Compteur(int tickDepart);
   public Compteur(int tickDepart, int pas); // Ajout d'un pas pour
plus de souplesse
   public int tickActuel();
   public int incremente(); // Incrémente du pas et retourne la
nouvelle valeur du compteur
   public void reinitialiser();

  private int m_cpt, m_pas;
}
```

Même si c'est une classe triviale, il faut bien comprendre les enjeux pour des structures plus complexes. Le raisonnement n'est pas si difficile à avoir et le code gagne énormément en auto-documentation. Vous m'accuserez peut être de tricher en rajoutant un pas pour le compteur; C'est vrai, mais je voulais surtout prendre un exemple qui mettait l'accent sur deux codes semblables dont l'un n'est pas fondamentalement faux mais dont le second est beaucoup plus idiomatique (respect des bonnes pratiques).

Plus d'informations à ce sujet :
- Blog d'Emmanuel Deloget : La guerre des accesseurs.
- Article sur javaworld.com : [Anglais] Why getter and setter methods are evil de Allen Holub.

Responsabilité limitée

Cette remarque ne concerne pas que la conception objet. Dans tout système, il est important de décomposer les rôles qu'auront vos classes. Il faut surtout éviter se retrouver avec une classe "utilitaire" qui s'occupe de beaucoup de ressources à la fois et

dont son rôle est mal défini. Prenons pour exemple une classe ConvertHTML avec le code suivant :

Code : C++

```
class ConvertHTML
{
  public:
   ConvertHTML(const std::string &source);

   void toPdfFile(const std::string &nomFichierDest);
   void toRtfFile(const std::string &nomFichierDest);
   void toTxtFile(const std::string &nomFichierDest);
   void toWordFile(const std::string &nomFichierDest);
   // ...
};
```

Code : Java

```
class ConvertHTML
{
  public ConvertHTML(String source);

  public  void toPdfFile(String nomFichierDest);
  public  void toRtfFile(String nomFichierDest);
  public  void toTxtFile(String nomFichierDest);
  public  void toWordFile(String nomFichierDest);
   // ...
}
```

Cet exemple montre vite que trop de choses sont gérées au sein de cette même classe, on va vite arriver à des complexités dans le code qui vont impliquées des modifications successives. C'est surtout le cas pour le format Pdf ou le format Word qui permettent des structures d'affichages aussi complexes que le Html. L'idéal est simplement de déporter du code afin d'avoir une architecture de code plus flexible. Il suffit donc d'éclater la classe en plusieurs plus petites qui se chargent de remplir un rôle mieux défini et de manière indépendante. Grâce à cette division en sous classes vous favorisez aussi la réutilisation de ces portions de codes qui sont moins dépendants d'un contexte précis. Dans l'exemple précédent, il faudrait que chaque méthode voit son traitement déporté dans des classes indépendantes comme **PdfFromHTML**.

Il faut faire preuve d'un peu de bon sens dans cet éclatement des rôles. Si vous voyez une utilité très faible à découpler alors ne le faites pas. Essayez juste de voir si votre classe peut fonctionner dans un autre contexte que le votre.

Si vous avez tendance à vouloir attribuer beaucoup de responsabilité à une classe, essayez de découper le problème, vous sentirez très vite son importance. D'ailleurs, beaucoup de patrons de conceptions se servent de ce principe pour avoir des codes très modulables. Certes, le projet voit son nombre de classes augmenter mais sa maintenance en sera réduite.

Une chose importante que je n'ai pas spécialement mis en évidence : il faut éviter que vos classes soient trop dépendantes les unes par rapports aux autre. En rejoignant un peu le principe de responsabilité unique, plus vos classes peuvent se passer de liens étroits avec d'autres classes en relation, plus serez apte à produire du code interchangeable ou réutilisable dans d'autres contextes avec d'autres avantages. Essayez de favoriser un couplage faible est une clef de réussite. La cohésion pour un système est aussi un atout majeur pour savoir si votre découpage en classes est efficace.

Mais c'est quoi la cohésion dans la conception ?

Il suffit tout simplement de prendre le nom de la classe, par exemple "ConvertHTML" et de savoir quels sont les services proposés par celle-ci. Si le nom de la classe est suffisamment évocateur pour les opérations proposées, on suppose que la classe joue un rôle que l'on peut clairement définir. Plus la définition du rôle est précise, plus la cohésion sera forte.

Plus d'informations à ce sujet :
- Le concept de base est appelé SRP.
- L'autre concept présenté après porte le nom de couplage et cohésion (application avec les patrons GRASP en Java).
- Blog d'Emmanuel Deloget : le principe de responsabilité unique.
- Article MSDN : cohésion et couplage.

Je vous conseille vivement de lire les deux articles présentés. Il sont très enrichissants et accessible facilement au niveau de la difficulté.

Privilegiez la composition à l'héritage

Derrière ce titre provocateur, je souligne plutôt une erreur causée par bon nombre de débutants : L'utilisation de l'héritage à tort et à travers. Je vous rassure tout de suite, l'héritage est utile dans certains cas mais je tenais à avoir votre attention.

Le vrai concept qui se cache derrière ce titre porte le nom de "substituabilité".

Beaucoup pensent que pour réutiliser le code, il n'y a rien de mieux que l'héritage. Cette erreur arrive fréquemment et même les développeurs assez chevronnés ont réussi à tomber dans le piège. On peut par exemple citer Java et la fameuse classe Stack (la structure de données pile) avec laquelle les développeurs de l'API ont décidé de la faire hériter de Vector (un tableau dynamique à accès direct). Le gros problème est que la classe Vector offre un surplus de fonctionnalités qui est d'une part inutile pour la classe Stack, d'autre part dans cet exemple précis va carrément offrir des comportements contraires à ce que devrait proposer une pile. On peut en effet ajouter un élément n'importe où et pas seulement sur le haut du tas, sans compter d'autres opérations dénuées de sens. Bref, réfléchissez à deux fois avant de d'hériter naïvement.

La solution pour ce genre d'embarras est simplement d'utiliser la composition, voyez plutôt :

Code : C++

```
//Mauvaise version
template <typename T>
class Pile : public std::vector<T>
{ /* TOUTES LES METHODES DE VECTOR SERONT PROPOSEES + CELLES DE LA
PILE. */ };

// Solution acceptable avec l'héritage privé. Revient au même que
la composition dans ce cas précis.
template <typename T>
class Pile : private std::vector<T>
{ };

// La solution à base de composition.
template <typename T>
class Pile
{
   private :
    std::vector<T> m_pile; // La composition consiste à avoir une
variable du type souhaité au lieu d'en hériter.
};
```

Code : Java

```
//Mauvaise version, il n'y a pas d'héritage privé en Java
class Pile extends ArrayList
{ /* TOUTES LES METHODES D'ACCES DIRECT SERONT PROPOSEES + CELLES DE
LA PILE. */ }

// La solution à base de composition.
class Pile<T>
{
   private ArrayList<T> m_pile; // La composition consiste à avoir
une variable du type souhaité au lieu d'en hériter.
}
```

Il est tentant d'hériter pour éviter de réécrire du code mais une chose essentielle a été oubliée. L'héritage doit être effectué lorsque la classe fille "est une" classe mère avec des fonctionnalités en plus **mais jamais moins**. La classe Pile n'est pas une classe Vector spécialisée, elle adopte un comportement différent et un peu de bon sens suffit à s'en rendre compte. Pour vous aider, rares sont les cas où hériter publiquement (type héritage imposé dans java) d'une classe concrète est préférable à une composition.

Le postulat de Barbara Liskov :
Cette personne ayant mené des recherches sur la programmation par contrat (je vous invite à vous renseigner si vous ne connaissez pas le sujet), une théorie en est ressortie pour concevoir une bonne relation d'héritage. Je vais vous éviter d'étaler sa théorie ici mais juste en faire une formulation simple, ce qui donne :
"Pour prétendre à l'héritage, une sous-classe doit être conçue de sorte que ses instances puissent se substituer à des instances de la classe de base partout où cette classe de base est utilisée."

En gros, cette phrase met l'accent sur la substituabilité d'une classe fille lorsqu'on utilise l'interface de la classe mère. Même si ce n'est pas toujours facile d'obtenir une classe fille qui calque ses comportements exactement de la même manière que la classe mère, c'est une bonne pratique que d'essayer de s'en rapprocher un maximum. Pour formuler différemment, les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir.


Les interfaces sont d'excellents moyens pour répondre à cette bonne pratique.

Cependant, ce n'est pas la seule chose qui est mise en avant par cette personne. Son domaine d'étude s'attardait sur la programmation par contrat en orienté objet et d'autres principes ont été évoqués lors d'un héritage. Le lien wikipédia donné pourra plus vous informer si le sujet vous intéresse.

Plus d'informations à ce sujet :
- Acronyme LSP pour la théorie de Liskov.
- Article sur developpez : Composition et héritage de Romain Guy.
- Article sur Wikipédia : Principe de substitution de Liskov.
- Inspiration du contenu : ~~www.crossbowlabs.com~~ (site fermé).

Pensez interface

Comme vous avez pu vous en rendre compte, les principes évoqués précédemment peuvent s'appliquer à d'autre paradigmes que celui de l'objet, on peut aussi bien placer ces arguments pour un langage impératif voir même fonctionnel. Une interface quant à elle représente une façade qui propose des opérations. Plus généralement, elle décrit des comportements que l'on peut utiliser. En programmation objet il est possible de représenter ce concept explicitement. Je vais faire un rappel mais si ce terme est nouveau pour vous c'est peut être que vous n'avez pas un niveau suffisant pour lire ce cours.


Les interfaces n'ont rien à voir avec ce que l'on appelle IHM. Ce n'est en rien une représentation visuelle de quoi que ce soit.

Une interface se définit comme suit :

Code : C++

```
/** En C++ une interface n'est rien d'autre qu'une classe qui ne
comporte que des méthodes
* publiques et virtuelles pures. Il n'y a aucune implémentation de
code (donc pas de variables membres)
*
* Cette interface va permettre de récupérer la valeur selon le nom
d'un champ dans une source de
* données.
*/
class IRequete
{
   public:
      // Retourne faux si le champ n'a pas été trouvé dans la source
de données.
      // Si le champ est trouvé, sa valeur sera écrite dans le
second paramètre.
      virtual bool requeter(const std::string &nomChamp, std::string
&valeurChamp) const = 0;
      // Ces méthodes ne sont pas nécessairement constantes.
};
```

Code : Java

```
/** Java comme d'autres langages plus récents possède un mot clef
```

```
destiné à cet usage.
* C'est un palliatif obligatoire à l'héritage multiple qui n'existe
pas.
*
* Cette interface va permettre de récupérer la valeur selon le nom
d'un champ dans une source de
* données.
*/
interface IRequete
{
   // Retourne la valeur du champ et null si celui-ci n'a pas été
trouvé dans la source de données.
   public String requeter(String nomChamp);
}
```

Définir une interface n'est pas bien compliqué, c'est une base pour induire des comportements à des classes plus concrètes. Un code possible serait :

Code : C++

```
class BDDRelationnelle : public IRequete// Comme un héritage
{
   // On est maintenant obligé de redéfinir les méthodes issues de
l'interface.
};

class XML : public IRequete// Comme un héritage
{
   // On est maintenant obligé de redéfinir les méthodes issues de
l'interface.
};
```

Code : Java

```
class BDDRelationnelle implements IRequete
{
  // On est maintenant obligé de redéfinir les méthodes issues de
l'interface.
}

class XML implements IRequete
{
  // On est maintenant obligé de redéfinir les méthodes issues de
l'interface.
}
```

Certains pourront se poser la question de la différence avec une classe abstraite. La réponse est simple, une classe abstraite permet une factorisation de code des classes filles. L'interface va forcer à **implémenter des comportements** (on peut l'assimiler à un contrat). On pourra dès lors passer par une interface pour effectuer des requêtes alors qu'on manipule des objets concrets. Je vais mettre un exemple pour illustrer plus facilement :

Code : C++

```
// On va utiliser le polymorphisme pour utiliser l'interface à la
place de la classe concrète.
std::string valeurRetour;
bool succesRequete;

// Pour faire du polymorphisme, le C++ oblige à utiliser les
pointeurs.
// On pourrait utiliser des pointeurs intelligents mais ce n'est pas
l'objet de ce cours.
IRequete *requeteur = new BDDRelationnelle();
```

```
succesRequete = requeteur->requeter("nomSARL", valeurRetour); //  On
requête sur la base de données relationnelle
if(succesRequete)
   std::cout << valeurRetour;
delete requeteur;

requeteur = new XML();
succesRequete = requeteur->requeter("nomSARL", valeurRetour); //  On
requête maintenant sur la source de données XML
if(succesRequete)
   std::cout << valeurRetour;
delete requeteur;
```

Code : Java

```
// On va utiliser le polymorphisme pour utiliser l'interface à la
place de la classe concrète.
String valeurRetour;

IRequete requeteur = new BDDRelationnelle();
valeurRetour = requeteur.requeter("nomSARL"); // On requête sur la
base de données relationnelle
if(valeurRetour != null)
   System.out.println(valeurRetour);

requeteur = new XML();
valeurRetour = requeteur.requeter("nomSARL"); // On requête sur la
base de données relationnelle
if(valeurRetour != null)
   System.out.println(valeurRetour);
```

A travers cet exemple on voit qu'on élève le niveau d'abstraction du programme, il devient simple de rajouter une nouvelle source de données (le CSV pour la forme) sans impacter le reste du code. Si je sais qu'il adoptera le comportement de mon interface, alors je pourrais l'interroger de la même manière, peu importe que son implémentation soit proche ou très éloignée de celle du XML.

Les interfaces sont la base de nombreux design pattern car elles apportent énormément de flexibilité en se dispensant de faire la moindre supposition sur les objets qui l'implémenteront. On sait juste que ces derniers doivent se contenter de satisfaire ces actions. L'ouvrage du GoF se permet même de faire l'assertion suivante :

"Programmer pour une interface, non pour une implémentation"

Si vous programmez avec des bonnes interfaces votre code sera beaucoup plus robuste aux changements et vous pourrez utiliser certaines fonctionnalités communes efficacement. L'exemple le plus parlant est sans doute l'itérateur Java pour parcourir une collection. Pas besoin de connaitre la structure interne d'une collection parmi les nombreuses existantes pour naviguer sur les éléments qui la constitue, il suffit d'utiliser l'interface Iterator. Vous pouvez donc parcourir une liste chaînée, un tableau dynamique, une structure en arbre ou je ne sais quoi de la même manière. Avouez que c'est vraiment utile.

L'itérateur est issu d'un design pattern du même nom "iterator" qui n'est pas si simple à mettre en place. Ce que vous pouvez constater si vous avez déjà utilisé ces objets (je l'espère pour vous), c'est qu'il est plutôt "simple" de s'en servir du point de vue utilisateur.

Retenez donc que les interfaces sont le début de la programmation flexible en objet. Grâce à elles, vous pouvez appliquer le principe de substitution pour vous passer d'éléments concrets lors de la conception.

La sensibilité de l'objet à subir des modifications

Cette partie ne concerne pas que le monde objet. La programmation en général supporte assez mal la modification de code existant; Que ce soit dû à la correction d'un problème ou l'ajout d'une nouvelle fonctionnalité nécessitant d'adapter l'ancien code.

Appliquer ces modifications est souvent source de nouveaux bugs entrainant des effets de bord que l'on n'avait pas prévus dans notre scénario initial. Vous savez, c'est le genre de petit bug qui survient très rarement (donc difficile à diagnostiquer) et

dont la provenance est insoupçonnée. C'est un fait, nous ne sommes pas parfaits, patcher un code qui était fonctionnel peut avoir des incidences car de nouveaux cas non anticipés surviennent.

La solution à ce problème est de prévoir que de nouvelles fonctionnalités donneront lieu à des ajouts dans le code **sans toucher à l'existant**. Je ne vous le cache pas, c'est une tâche ardue que de rendre sa conception suffisamment solide pour que les nouveaux besoins viennent se greffer sur ce qui existe. Ce serait une sorte d'addon si vous voulez. Les patrons de conception misent énormément pour éviter une modification de code si une fonctionnalité venait à se produire. C'est peut être l'un de leur plus gros point fort, l'ajout d'un besoin rajoutera uniquement du code sans aucune forme d'altération. C'est d'ailleurs pourquoi ils sont si efficaces lorsqu'ils répondent à un problème : Le risque d'erreur d'implémentation est très faible.

En reprenant l'exemple précédant sur la classe ConvertHTML et son implémentation sans découpage des responsabilités voici ce qu'on obtient :

Code : C++

```
class ConvertHTML
{
  public:
   ConvertHTML(const std::string &source);

   void toPdfFile(const std::string &nomFichierDest) { /*        CODE
D'IMPLEMENTATION */ }
   void toRtfFile(const std::string &nomFichierDest) { /*        CODE
D'IMPLEMENTATION */ }
   void toTxtFile(const std::string &nomFichierDest) { /*        CODE
D'IMPLEMENTATION */ }
   void toWordFile(const std::string &nomFichierDest) { /*       CODE
D'IMPLEMENTATION */ }
   // ...
};
```

Code : Java

```
class ConvertHTML
{
  public ConvertHTML(String source);

  public  void toPdfFile(String nomFichierDest) { /*            CODE
D'IMPLEMENTATION */ }
  public  void toRtfFile(String nomFichierDest) { /*            CODE
D'IMPLEMENTATION */ }
  public  void toTxtFile(String nomFichierDest) { /*            CODE
D'IMPLEMENTATION */ }
  public  void toWordFile(String nomFichierDest) { /*           CODE
D'IMPLEMENTATION */ }
   // ...
}
```

Si on a un problème on va être obligé d'intervenir sur cette classe. Une nouvelle conversion va aussi imposer de mettre son implémentation dans ce code. Vous vous retrouvez donc avec une classe de milliers de lignes de code. Une structuration plus intelligente utiliserait une interface capable d'abstraire le comportement des formats de destination. Quel est le comportement que l'on cherche à généraliser lors de la conversion ? Tout simplement la transformation, on peut donc créer l'interface correspondante :

Code : C++

```
// En C++
class IConvertisseurHTML
{
  public :
   virtual void fromHTMLToFile(const std::string &html, const
std::string & fichierDestination) const = 0; // virtuelle pure
};
```

Code : Java

```
// En Java
interface IConvertisseurHTML
{
  public void fromHTMLToFile(String html, String
fichierDestination);
}
```

Ensuite on peut imaginer que la classe qui transforme le Html en Pdf va implémenter cette interface. Cependant je ne m'étends pas plus sur le sujet et pour cause le design pattern strategy répond à ce problème posé de manière très efficace. Ce n'est cependant pas le seul design pattern à user de ce principe. Si vous souhaitez savoir comment, suivez la suite du tutoriel !

Plus d'informations à ce sujet :
- le terme désigné pour ce concept se nomme OCP. On parle aussi de respecter le principe Open/Closed.
- Article en C++ sur Régis Medina.

Pour aller plus loin

Si vous appliquez la plupart des théories du paradigme explicitée plus haut vous êtes déjà paré pour concevoir d'une manière efficace. Ce sont des points qui ont tous pour vocation de rendre la programmation orientée objet réutilisable (l'argument majeur qui est tout le temps ressorti) en obtenant un maximum de flexibilité/souplesse au code. Si certains de ces concepts étaient nouveaux pour vous, certaines choses doivent être encore floues ou difficilement réalisables. Et oui, une bonne conception objet **est tout, sauf une tâche simple**, du moins pas aussi simple que certains voudraient le croire.

Vous comprenez peut être mieux la motivation qu'ont eu certaines personnes dans l'écriture de catalogues sur des patrons de conceptions réutilisables. Ces recettes ont été le fruit de beaucoup de remaniements avant d'arriver à une version très aboutie. Les auteurs avouent eux même que l'élaboration de patrons de conception est *"le résultat travail obscur de remaniements de la conception et du code effectué par les développeurs"*.

Une chose importante dont je n'ai pas parlé, dans la grande majorité des situations vous n'êtes pas sensé connaître le type précis d'un objet. L'utilisation des opérateurs *instance_of, is, type_info, typeid , getClass(), etc.* révèle souvent comme une erreur de conception. Soyez sûr que vous voulez connaitre le type non pas par fainéantise mais par nécessité. Retenez qu'une bonne conception devrait se passer de savoir l'instance réelle d'un objet en se substituant par **polymorphisme** sauf si elle engendre une complexité marquante.

Si vous souhaitez encore approfondir votre connaissance dans le paradigme objet, voici des sujets et les mots clefs correspondants qui pourront vous aiguiller dans vos recherches. Bien sûr, les quelques explications qui sont données ne sont qu'une vulgarisation de ces sujets :

Construction par couches :

Un concept qui recoupe pas mal ce qui a été dit plus haut dans ce chapitre est de développer avec de l'abstraction. On peut visualiser une application comme une succession de couches qui partent d'un niveau d'abstraction très élevé et en empilant des couches de plus en plus concrètes. En procédant de cette manière on s'assure que les attentes fixées dans les couches abstraites sont respectées dans les classes plus concrètes.
Mot clef associé : DIP ou Principe d'inversion des dépendances.

La différentiation des classes "entité" et classe "valeur" :

Bien que cette notion ait plus d'importance en C++ que dans d'autres langages plus modernes, il est toujours intéressant de connaître ce que sont les classes entités à opposer aux classes de valeurs. Les classes à sémantique de valeur représentent des classes dont deux instances différentes peuvent être considérées égales si elles ont le même contenu (par exemple la comparaison de 2 Couleurs). A l'inverse, une classe à sémantique d'entité n'autorise pas de dire que deux instances différentes au contenu identique sont équivalentes (une personne est unique même si deux peuvent avoir le même nom et prénom). **Elle connote une identité propre** à chaque objet.

Faire cette distinction permet de savoir quelles opérations particulières doivent être implémentées. Les liens donnés juste après rentrent un peu plus en détail. L'intérêt bien que moins important en Java est de savoir quand écrire certaines méthodes comme par exemple equals() ou des opérations d'ajout, de soustraction, etc.

Références :
- Classes à sémantique de valeur : faq developpez.
- Classes à sémantique d'entité : faq developpez.

Évitez la duplication de code :

Derrière ce principe simple, il n'est pas aisé d'avoir un code qui ne possède pas de redondance ou très peu. L'usage d'une bonne

conception objet va réduire drastiquement cette duplication. Il existe d'autres pratiques permettant de poser les bases sur l'écriture unique de code. Cette étape vient généralement pendant l'écriture du code et se retrouve donc en aval de la conception.
Mot clef associé : DRY

Pour aller encore plus loin :
- Réalisation d'interfaces avec découplage de l'implémentation : le pattern NVI, faq sur developpez.
- Commonalities & variation points : un sujet complexe de Coplien qui permet d'avoir une vision très abstraite permettant de voir plus loin que le simple mécanisme du polymorphisme.

Auteurs ayant activement participé à la conception objet (non exhaustif):
- Jim Coplien.
- Robert C Martin.
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Gof)

Que de choses complexes ont été relatées à travers ce chapitre. Oui, vous êtes fixés, respecter les différents postulats survolés nécessite une grosse part d'investissement intellectuel pour réussir à les satisfaire mais les bénéfices sont sans appel : Vous pourrez non seulement vous targuer d'avoir des codes qui ne seront plus remis en doute à la moindre occasion et votre application pourra évoluer beaucoup plus sereinement. Aussi, les corrections de bogues auront beaucoup moins de chances de se propager en cascade.

En route pour découvrir un patron de conception très largement utilisé aujourd'hui, "l'observer". La plupart des concepts évoqués vont être utilisés, soyez à l'aise avec ces grandes lignes et vous pourrez avoir un regard critique sur le "pourquoi" ils en sont devenus des références aujourd'hui. Ne vous contentez pas d'apprendre les patterns, assimilez les mécanismes sous-jacents pour pourvoir les réutiliser lorsque vous tomberez sur un "os".

Partie 2 : Commençons en douceur

Vous êtes imprégné des concepts objets qui favorisent une grande flexibilité et ré-utilisabilité ? Voyons maintenant des design patterns simples qui feront une excellente initiation.

Le patron de conception Observateur (Observer)

Pour bien comprendre pourquoi et comment utiliser un design pattern, il faut une raison. Je vais donc vous définir un sujet particulier sur lequel vous devrez cogiter. Oui, vous avez bien lu, il va falloir réfléchir. Le meilleur moyen de retenir durablement ce qui va suivre est de buter sur le problème. Pour ma part je suivrai un cheminement pour essayer progressivement de m'approcher de la solution.

Essayez de vous forcer un minimum à réfléchir avant de lire la solution. Si je ne devais donner qu'un seul conseil, ce serait de prendre un papier et un crayon afin de tenter la résolution par vous-même.

Posons le problème

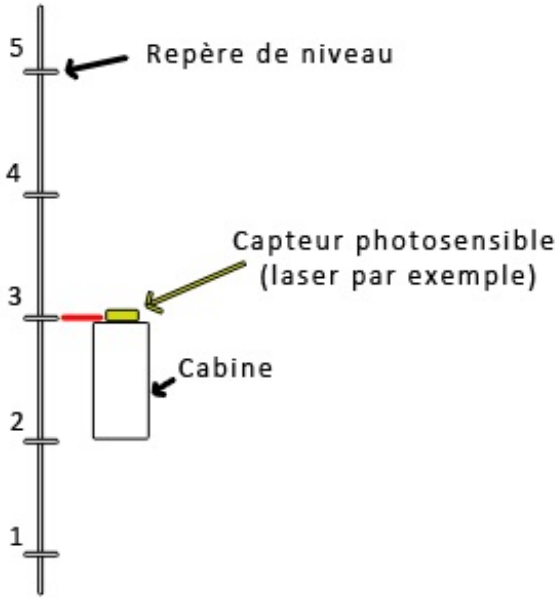
Soit un ascenseur, sa fonction première est de monter ou de descendre des étages pour satisfaire des requêtes de personnes. Comme vous le savez peut être, l'installation d'un ascenseur ne se réduit pas à placer la cabine qui se déplacera dans un rail. De multiples capteurs sont présents pour que la cabine puisse se comporter idéalement selon les situations, en voici une petite liste :

- Un capteur qui indique qu'un utilisateur ne se retrouve pas entre deux niveaux lorsque les portes doivent s'ouvrir.
- Un capteur situé au niveau des portes pour savoir si quelqu'un n'est pas totalement rentré dans l'ascenseur.
- Un capteur qui mesure la pression exercée sur les portes de l'ascenseur lorsqu'elles sont ouvertes. Ce dispositif permet de s'assurer que rien ne sera écrasé si une trop forte résistance s'oppose lors de la fermeture.

Cette liste n'est pas exhaustive, mais c'est pour vous montrer que sous son apparence simpliste, un ascenseur comporte des mécanismes complexes (sans parler du traitement des requêtes utilisateurs).

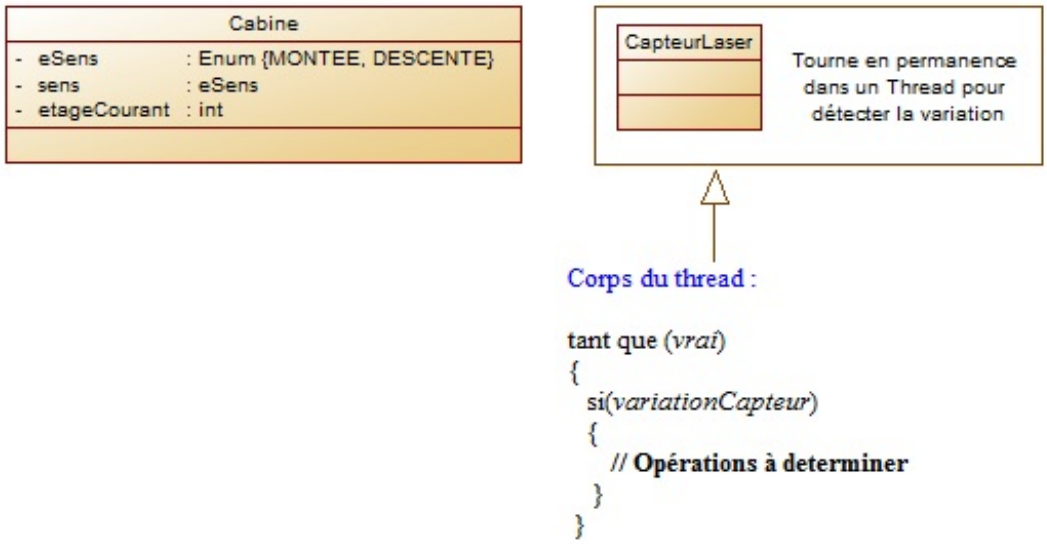
Les personnes ayant connaissance des automates programmables, des grafcets ou de la création de schémas électriques complexes savent de quoi il est question.

Par cette brève introduction, rentrons maintenant dans le vif du sujet. Vous êtes en charge de gérer **la bonne position** de la **cabine de l'ascenseur**. Pour cela, la cabine possède diverses informations comme son sens courant de déplacement et son étage actuel. Le seul moyen pour vous de savoir quand l'ascenseur changera d'étage sera par l'intermédiaire d'un **capteur photosensible** placé sur l'ascenseur et détectant un **repère placé sur chaque étage**. Comme un schéma est souvent plus clair pour montrer de quoi il est question, je vous propose le suivant :



Dès que le laser passe sur le repère, une variation va être détectée. Le capteur laser sait donc qu'un étage a été décelé. La cabine doit maintenant mettre à jour son nouvel étage courant.

Voici les éléments qui sont donnés pour résoudre ce problème :



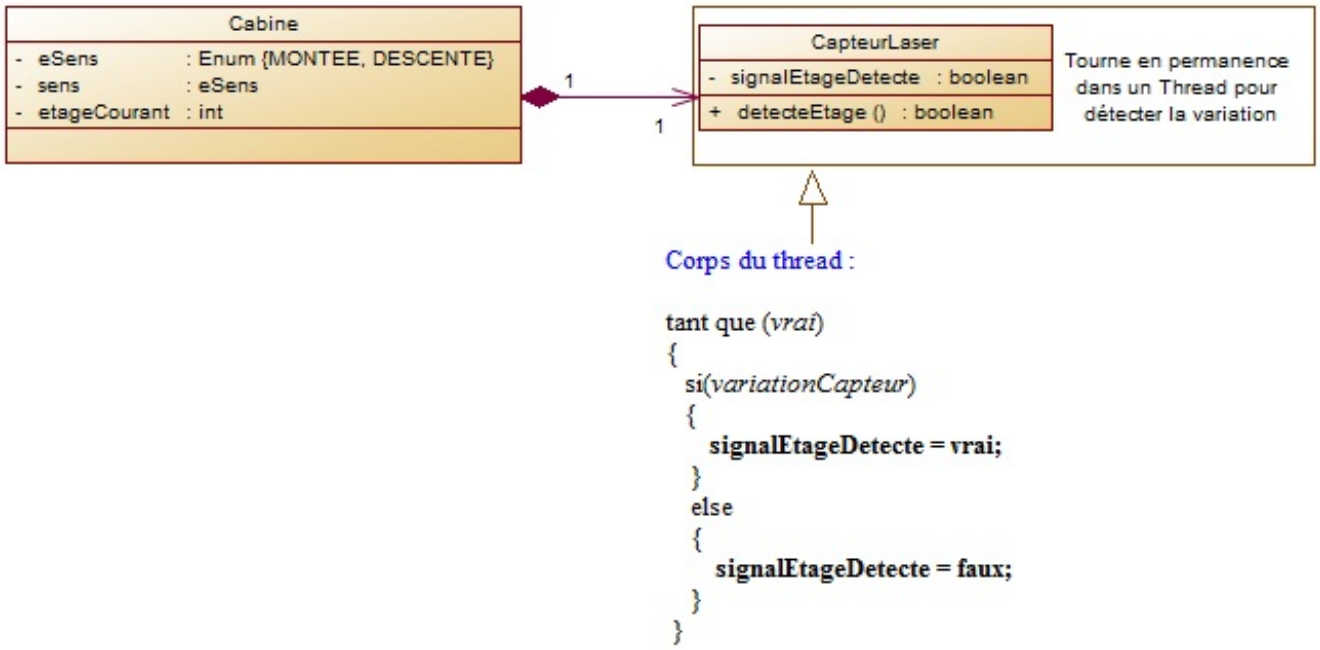
Vous pouvez voir que le capteur est dans un thread, cela importe peu, concentrez-vous juste sur les opérations à faire lorsqu'une variation est détectée.
En gros, ne touchez à rien d'autre dans la méthode du thread que le corps de la boucle.

Avec ces éléments de base, vous pouvez modifier les classes à votre souhait, rajouter autant de méthodes que vous le souhaitez, l'essentiel étant de faire diffuser l'information du capteur vers la cabine et de mettre à jour l'étage de celle-ci. A vos brouillons !

Résolvons progressivement notre cas

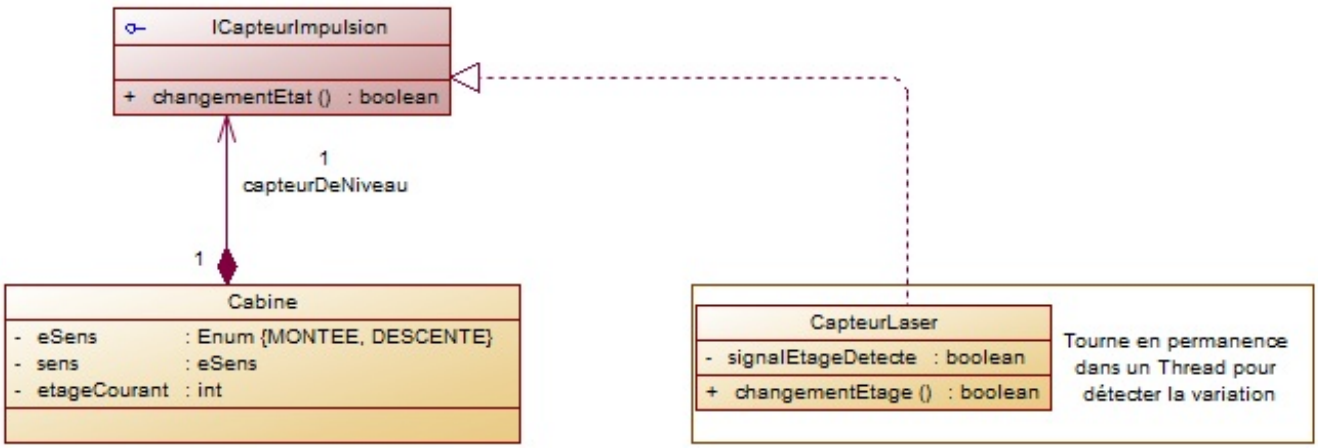
Comme dit précédemment dans ce tutoriel, je vais décrire plusieurs approches qui sont issues de raisonnements simples pour montrer comment doit s'organiser votre pensée. Vous verrez qu'il s'agit de l'approche de la réussite par l'échec. Si vous avez suffisamment assimilé le sujet, deux voire trois options vous sont peut-être venues à l'esprit, je vais tenter de décrire chacune d'entre-elles et les confronter aux principes objets que j'ai évoqué dans le chapitre précédent.

Première approche, scruter le capteur pour déterminer une variation :
On peut partir du principe que le capteur vérifie en permanence le repère d'un étage. Avec cette approche, il suffit de créer une méthode renvoyant un booléen qui indique si oui ou non une variation a été détectée. Voyons une version de ce diagramme UML :



Il ne manque plus qu'à vérifier que la méthode *detecteEtage()* renvoie *vrai*.

Oui dans les faits, c'est une manière plutôt élégante de résoudre ce problème, vous respectez même les grands principes objets. En effet, la cabine peut juste consulter l'état d'un objet sans en altérer le comportement, aussi chaque objet peut évoluer indépendamment. En rajoutant un peu d'abstraction on pourrait même se payer le luxe de pouvoir représenter n'importe quel capteur qui signale juste un changement d'état. Comment ? à l'aide d'une interface bien sûr, je vous laisse réfléchir sur le diagramme suivant :



Voilà une conception très flexible, si on décide de changer de capteur (par exemple un capteur à aimantation), il suffira de créer une nouvelle classe qui implémente l'interface. Le code pour la cabine ne changera pas.

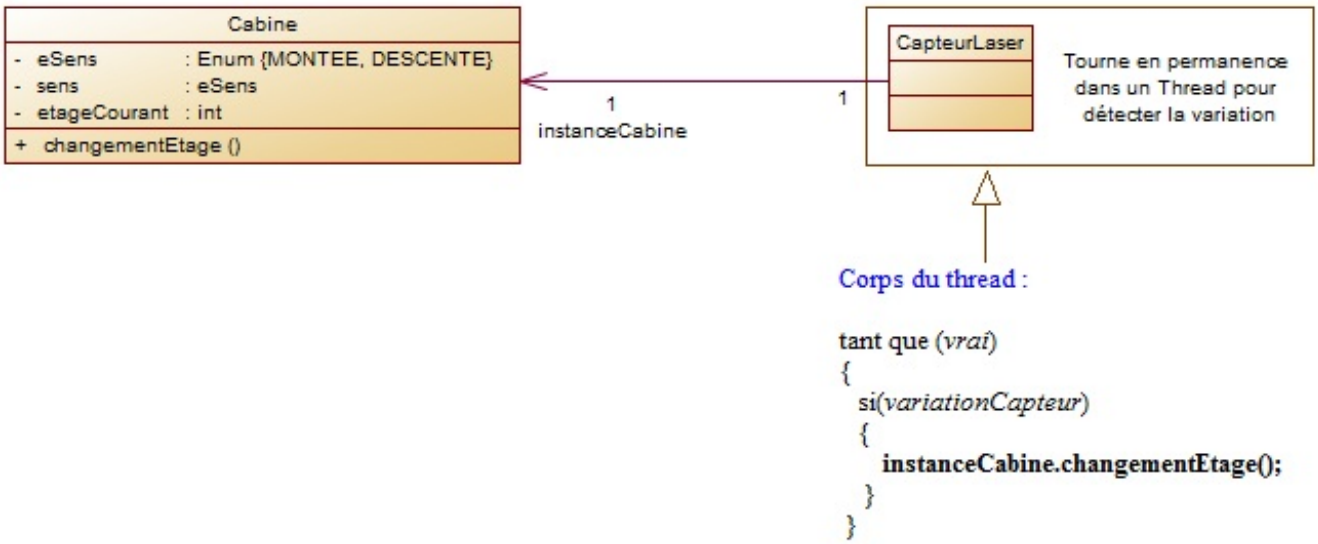
Cependant cette approche a un problème et vous avez probablement remarqué que je n'en ai pas parlé. Quel est le code exact à mettre dans la classe Cabine ? Puisqu'on doit scruter les changements d'états du capteur, il faut pouvoir connaître en permanence l'état courant de celui-ci. Si vous avez deviné, oui il s'agit bien d'une boucle qui s'occupe de vérifier la variation. Et c'est encore une boucle infinie comme pour le capteur et bloquante de surcroît si on ne la place pas dans un autre thread. L'implémentation est donc assez fastidieuse dans la cabine mais elle est possible. L'inconvénient majeur est que malgré tout, on va consommer des ressources processeur inutilement et il faudra éventuellement gérer les problèmes de synchronisme de thread. Imaginez que le capteur détecte toutes les 60èmes de seconde, il est important que la cabine consulte **au minimum** 60 fois par seconde aussi.

Le framework MFC de microsoft reprenait dans les grandes lignes cette approche pour gérer les évènements. S'il paraît absurde aujourd'hui de procéder de cette manière, à l'époque rien ne choquait particulièrement.

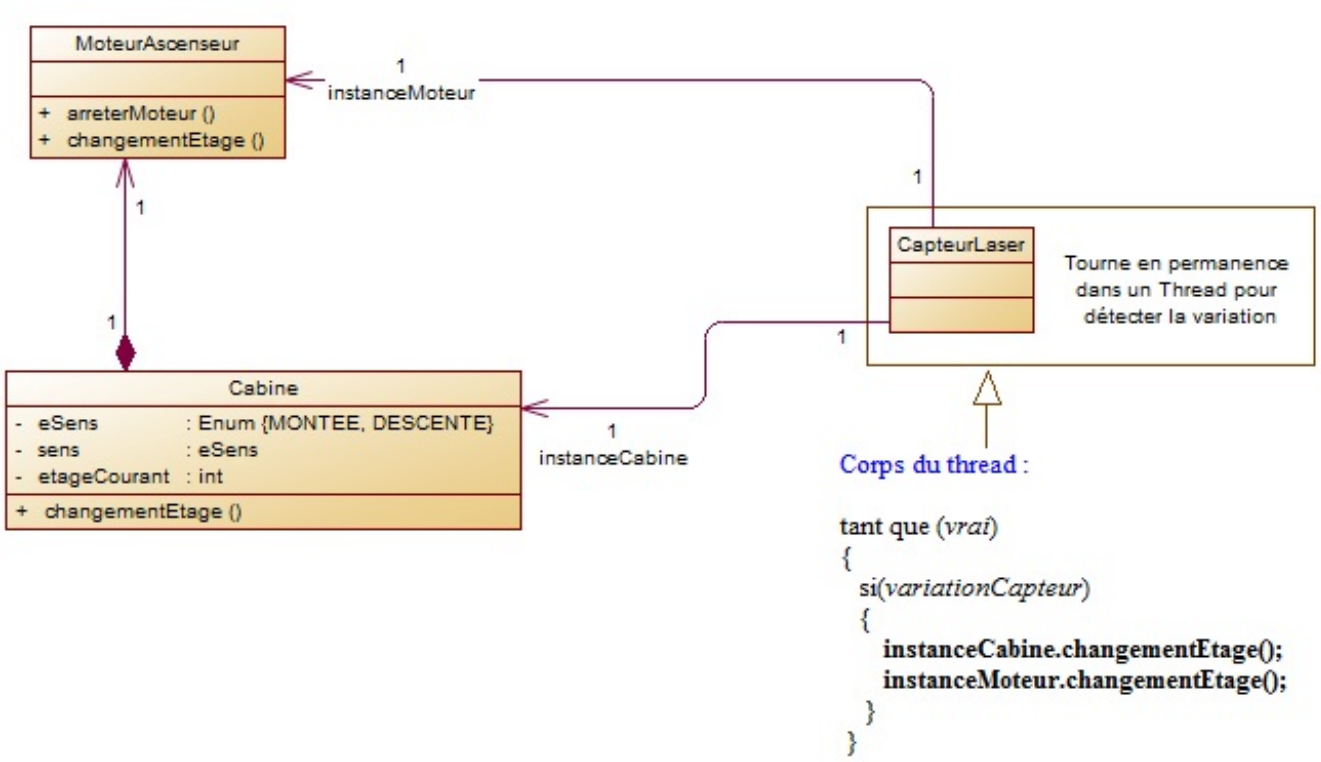
Retenez que cette solution n'est pas mauvaise en soit mais pour des raisons de performances/complexités, elle n'est pas spécialement adaptée à nos besoins. C'est pourquoi nous l'abandonnons pour un autre raisonnement tout aussi trivial. Voyez tout de même la démarche adoptée pour réussir à pouvoir définir un comportement commun à certains capteurs, ce qui fait que l'écriture du thread pour la cabine n'aurait pas changé si on "substituait" un autre capteur au même comportement. Adopter cette démarche est la voie de la programmation réutilisable et flexible.

Seconde approche, avoir l'instance de l'ascenseur dans la classe capteur :

Une autre solution serait tout simplement de prévenir la cabine qu'une variation a été détectée. Cette méthode est beaucoup plus efficace et ne comporte pas de défaut particulier. Il faut juste que la cabine puisse proposer une méthode sur laquelle on pourra informer un changement d'étage. Comme à mon habitude, je vous propose un diagramme UML de cette solution :



A vrai dire, si vous modélisez le problème de cette manière, vous vous passez d'une deuxième boucle comme traitée dans la première approche et vous respectez par la même occasion les grands principes objets. En fait pour tout vous dire, c'est une implémentation concrète du design pattern observateur. Le seul inconvénient majeur de notre capteur est que si nous voulons ajouter une autre classe qui souhaite savoir qu'une variation a eu lieu, il va falloir rajouter une référence explicite dans le capteur laser et rajouter l'instruction dans notre boucle de thread.

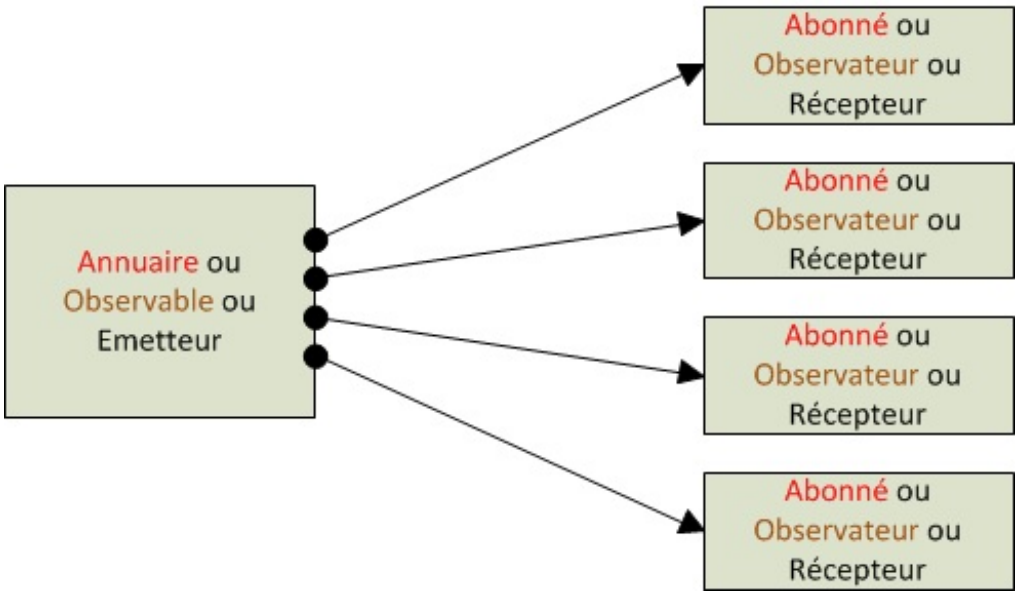


On pourrait imaginer que la cabine contrôle le moteur. Admettons que pour offrir une sécurité supplémentaire, ce moteur vérifie indépendamment si l'ascenseur n'est pas à un niveau hors des bornes (13ème étage par exemple alors que le bâtiment ne possède que 12 étages).

Dans ce cas précis, on voit bien que le code du capteur va devoir être modifié pour renseigner cette information au moteur. Rappelez-vous qu'une bonne conception doit être fermée autant que possible aux modifications sur le code existant. Si vous avez suivi mon discours jusqu'à présent, vous pouvez déjà entrevoir une solution à l'aide d'interfaces, cette abstraction qui permettra au capteur laser de signaler à un nombre quelconque de classes sa variation sans modifier le code existant se rapprochera du patron de conception observateur. Voyons maintenant de quoi il est question.

L'observateur

Ce que nous cherchons à faire ici se résume simplement avec un schéma :



Transposé à notre cas, l'émetteur d'une information se révélerait être le capteur. Les récepteurs de signalement seraient la cabine ainsi que le moteur.

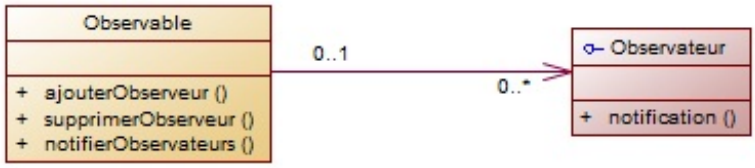
Je vous ai mis plusieurs couples de mots qui peuvent aider certaines personnes à comprendre dans quel cas nous pouvons appliquer le problème. Notez que les termes "Observable" et "Observateur" sont les mots les plus représentatifs car ils sont **plus abstraits** (assez difficile à juger tout de même). Cependant vous ne serez pas en tort si vous utilisez les autres termes .

Grâce à cette représentation, on peut directement constater quels termes vont apparaître dans notre conception pour manipuler

des abstractions. Il y a cependant un détail à noter : Un annuaire, pour prévenir ses abonnés qu'un changement d'état existe (on parle de notification), doit contenir la liste des intéressés. Ce qui implique que l'annuaire doit proposer des méthodes permettant à n'importe qui de s'inscrire et de se désinscrire aux moments souhaités.

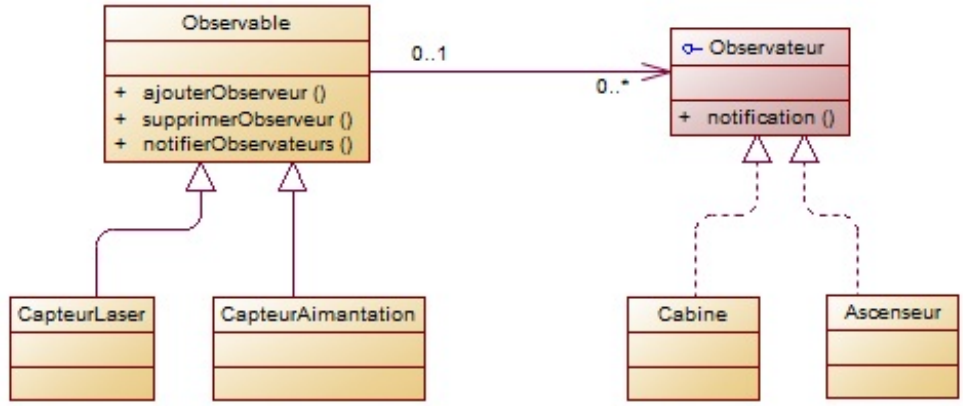
Ces opérations communes à tout ce qui pourrait être observable nécessitent d'écrire une implémentation. L'interface ne sera donc pas adaptée pour représenter un annuaire.

L'observateur lui, se contente de s'inscrire à ce qui l'intéresse, il a juste besoin d'être prévenu lors d'une parution. Ce sera donc une interface avec la méthode permettant de notifier.



La méthode notification() permet de signaler un changement de la part de l'observable.

Nous avons notre version minimale du patron de conception Observateur ! Voyez que les meilleures solutions ne sont pas forcément les plus compliquées (c'est même souvent le contraire). Il a suffi d'une classe et d'une interface pour pouvoir modéliser ce comportement. En l'appliquant à notre sujet de départ voici le diagramme de classes UML que l'on obtiendrait :



Passons maintenant à notre implémentation (je ne vais que m'attarder sur la partie code sur l'observateur et l'observable) :

Code : C++

```
#include <string>
#include <list>
#include <iostream>
using namespace std;

class IObservateur
{
  public:
    virtual void notifier() = 0;
};

class Observable
{
  public:
    void notifierObservateurs() const
    {
        // Notifier tous les observateurs
        list<IObservateur*>::const_iterator end =
m_observateurs.end();
        for (list<IObservateur*>::const_iterator it =
m_observateurs.begin(); it != end; ++it)
            (*it)->notifier();
    }
```

```
    void ajouterObservateur(IObservateur* observateur)
    {
        // On ajoute un abonné à la liste en le plaçant en premier
(implémenté en pull).
        // On pourrait placer cet observateur en dernier (implémenté
en push, plus commun).
        m_observateurs.push_front(observateur);
    }

    void supprimerObservateur(IObservateur* observateur)
    {
        // Enlever un abonné a la liste
        m_observateurs.remove(observateur);
    }

  private:
        list<IObservateur*> m_observateurs;
};

class Cabine : public IObservateur
{
  public:
    void notifier()
    {
        cout << "Cabine a recu la notif" << endl;
        //  Changement  d'étage  selon  son  sens  et  sa  position
précédente.
    }
};

class Moteur : public IObservateur
{
  public:
    void notifier()
    {
        cout << "Moteur a recu la notif" << endl;
        // Verification que l'étage soit dans les bornes autorisées.
    }
};

class CapteurLaser : public Observable
{
  public:

   // Le code de la boucle while en environnement Threadé
   void run()
   {
       while(true)
       {
           if(m_detecteVariation)
               notifierObservateurs();
       }
   }

   private:
     bool m_detecteVariation;
};

int main()
{
    Cabine instanceCabine;
    Moteur instanceMoteur;

    CapteurLaser capteurEtage;

    capteurEtage.ajouterObservateur(&instanceCabine);
    capteurEtage.ajouterObservateur(&instanceMoteur);

    // On simule manuellement une variation (normalement c'est le
thread qui s'en charge)
```

```
    capteurEtage.notifierObservateurs();

    // La cabine et le moteur ont reçu une notification sur leur
méthode notifier()

    capteurEtage.supprimerObservateur(&instanceMoteur);
    cout << "Suppression du moteur dans les abonnes" << endl;

    capteurEtage.notifierObservateurs();


    return 0;
}
```

Et le java correspondant :

Code : Java

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

interface IObservateur
{
 public void notifier();
}

class Observable
{

 public Observable()
 {
  m_observateurs = new LinkedList<IObservateur>();
 }

 public void notifierObservateurs()
 {
    Iterator<IObservateur> it = m_observateurs.iterator();
     // Notifier tous les observers
    while(it.hasNext()){
     IObservateur obs = it.next();
     obs.notifier();
    }
 }

 void ajouterObservateur(IObservateur observateur)
 {
     // On ajoute un abonné à la liste en le plaçant en premier
(implémenté en pull).
            //  On pourrait placer cet observateur en dernier
(implémenté en push, plus commun).
     m_observateurs.add(observateur);
 }

 void supprimerObservateur(IObservateur observateur)
 {
     // Enlever un abonné a la liste
     m_observateurs.remove(observateur);
 }

 private List<IObservateur> m_observateurs;
}

class Cabine implements IObservateur
{
 public void notifier()
 {
  System.out.println("Cabine a recu la notif");
  // Changement d'étage selon son sens et sa position précédente.
```

```
  }
}

class Moteur implements IObservateur
{
 public void notifier()
 {
   System.out.println("Moteur a recu la notif");
   // Verification que l'étage soit dans les bornes autorisées.
 }
}

class CapteurLaser extends Observable
{
   // Le code de la boucle while en environnement Threadé
 public void run()
 {
  while(true)
  {
   if(m_detecteVariation)
    notifierObservateurs();
  }
 }

 private boolean m_detecteVariation;
}


public class Run {

 /**
* @param args
*/
 public static void main(String[] args)
 {
  Cabine instanceCabine = new Cabine();
  Moteur instanceMoteur = new Moteur();

  CapteurLaser capteurEtage = new CapteurLaser();;

  capteurEtage.ajouterObservateur(instanceCabine);
  capteurEtage.ajouterObservateur(instanceMoteur);

  // On simule manuellement une variation (normalement c'est le
thread qui s'en charge)
  capteurEtage.notifierObservateurs();

  // La cabine et le moteur ont reçu une notification sur leur
méthode notifier()

  capteurEtage.supprimerObservateur(instanceMoteur);
  System.out.println("Suppression du moteur dans les abonnes");

  capteurEtage.notifierObservateurs();
 }
}
```

La sortie obtenue après exécution :

Code : Console

```
Cabine a recu la notif
Moteur a recu la notif
Suppression du moteur dans les abonnes
Cabine a recu la notif
```

Le principal dans tout ce que vous avez lu jusqu'à présent est le **cheminement suivi**. Si vous avez assimilé cette démarche c'est que l'essentiel sur ce chapitre vous a été retransmis. Il est très important de de suivre un raisonnement et de réfléchir avant de partir tête baissée dans le code. J'insiste encore sur le fait qu'**une bonne conception est la clef de voute pour du code réutilisable, extensible et robuste**. **Je rappelle que mon objectif premier sur ce cours n'est pas de vous apprendre à utiliser les design patterns mais d'apprendre à concevoir d'une meilleure manière**. Il existe tellement de sources sur internet qui traitent des patrons de conception que je n'ai aucune prétention de faire mieux. Je développe plutôt la démarche qui montre le résultat d'une conception très réfléchie à partir d'idées concrètes.

Maintenant sachez qu'en Java vous ne serez pas obligés de devoir écrire le code de l'Observateur et de l'Observable, ce design pattern étant tellement utilisé, il existe dans l'API de base :

Code : Java

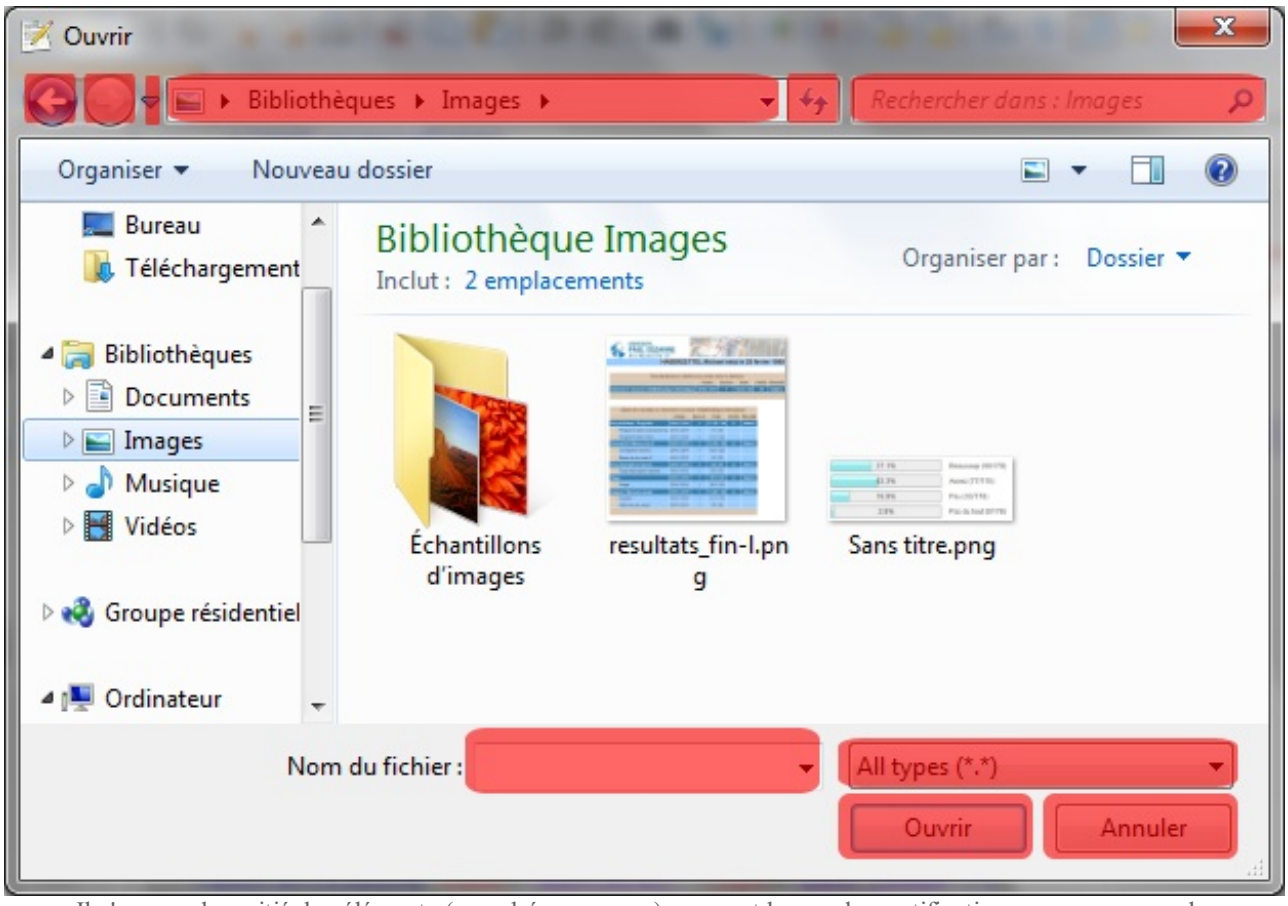
```
// Exemple tiré de wikipédia
class Signal extends Observable {

  void setData(byte[] lbData){
    setChanged(); // Positionne son indicateur de changement
    notifyObservers(); // (1) notification
  }
}
/*
On crée le panneau d'affichage qui implémente l'interface
java.util.Observer.
Avec une méthode d'initialisation (2), on lui transmet le signal à
observer (2).
Lorsque le signal notifie une mise à jour, le panneau est redessiné
(3).*/

class JPanelSignal extends JPanel implements Observer {

  void init(Signal lSigAObserver) {
    lSigAObserver.addObserver(this); // (2) ajout d'observateur
  }

  void update(Observable observable, Object objectConcerne) {
    repaint();  // (3) traitement de l'observation
  }
}
```

Le code varie un peu, on peut voir que la méthode update(...) correspond à notre méthode notifier().

Mais attend ! Pourquoi il y a deux arguments supplémentaires dans la méthode  ?

Pas de panique, un petit tour dans la documentation nous éclaire très vite. Le premier argument correspond à l'objet émetteur de la notification, alors que le second paramètre permet de passer des valeurs relatives à cet évènement. Prenons pour exemple un thermomètre qui lorsqu'il change sa température va envoyer la notification à ses abonnés. Oui mais on veut aussi obtenir quelle température était indiquée au thermomètre lorsqu'il a émis la notification. C'est le rôle du second argument. Le premier argument quant à lui permet de reconnaitre qui a indiqué la notification si notre objet s'est abonné à plusieurs sources (par exemple un objet qui reçoit la notification de deux thermomètres différents).

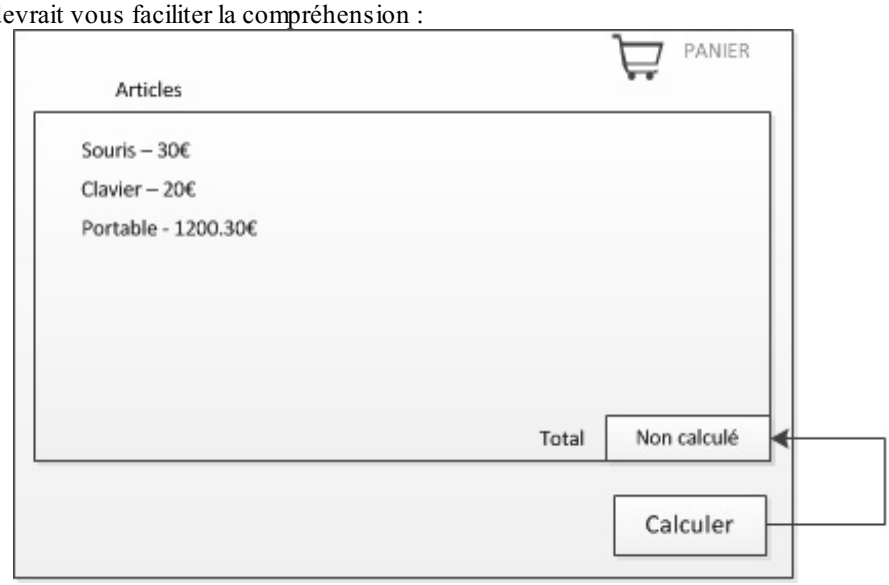
Connaître l'émetteur peut vous sembler superflu mais dans la programmation événementielle ce cas arrive assez souvent. Pour ceux qui auraient un doute, la programmation événementielle s'utilise très souvent lorsque les logiciels fonctionnent avec une interface graphique. Tout est évènement sur les interfaces et regardez sur une fenêtre à l'apparence simple le nombre d'évènements que l'on peut lancer :



Il n'y a que la moitié des éléments (encadrés en rouge) pouvant lancer des notifications sur mon exemple.

Sachez que chaque bouton, liste déroulante ou autre utilise en principe une base de ce design pattern (même si certains IDE camouflent leur utilisation à l'aide d'outils graphiques pour dessiner les fenêtres). C'est un patron de conception qui a un très fort taux d'utilisation.

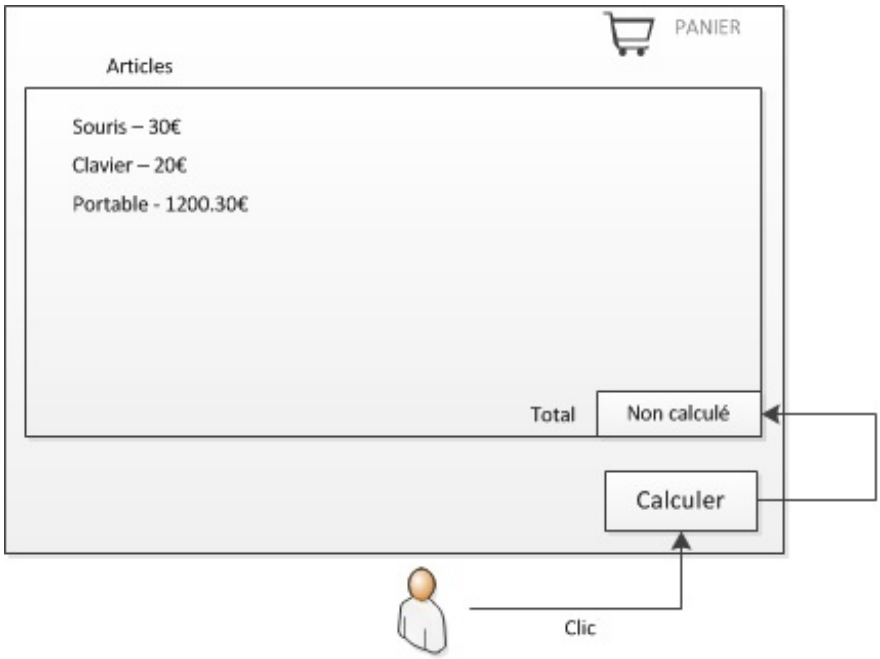
Pour bien insister sur le fonctionnement de ce patron de conception, je vais vous proposer un exemple dans une application graphique. Il sera ici question d'une interface de panier et vous verrez que son utilisation est moins intuitive qu'elle en a l'air. Mais penchons nous sur le sujet : Une liste d'articles est représentée et affichée dans un contrôle avec un montant total n'est pas calculé à l'affichage. Un bouton permet d'estimer le coût total des composants et de le renseigner dans un champ prévu à cet effet. L'image qui suit devrait vous faciliter la compréhension :



IHM d'un panier. On peut voir le bouton qui permet de calculer le prix et de remplir le champ "Total".

Le problème avec le bouton, c'est qu'aucune capture d'évènement n'est associée dessus. Un bouton est à la base un simple composant comme le reste, inerte. Il est nécessaire de venir lui indiquer qu'un clic utilisateur va engendrer un comportement spécifique. C'est là que le pattern observateur entre en jeu : **l'utilisateur est donc observé** pour savoir si une pression sur le bouton a été exercée, tandis que le **bouton devient un observateur**. Autant vous le dire tout de suite, les programmes avec interfaces graphiques **camouflent la partie d'utilisateur observé** (le framework ou le système d'exploitation de charge de cet

élément). Vous, en tant que programmeur, vous n'avez qu'à vous préoccuper du bouton et de le rendre observateur. La méthode pour rendre un composant graphique observateur diffère selon le langage et les outils utilisés mais le principe reste toujours le même. Le résultat sera donc le suivant :



Le bouton devient observateur d'un clic. Lorsque l'utilisateur appuiera sur le bouton, l'interface graphique va appeler l'évènement que le bouton écoute.

C'est ainsi que s'achève notre première étude de cas. J'espère que ce chapitre aura été enrichissant pour vous. si tel est le cas, j'aurai réussi mon pari. Assurez vous de comprendre comment fonctionne ce patron de conception car il est utilisé dans de nombreux cas et même dans d'autres patrons de conception plus complexes.

Si vous n'avez pas saisi quelque chose, relisez ce chapitre à tête.

Le patron de conception État (State)

Toujours présents ?
Nous voilà fin prêt à attaquer sur un autre sujet. Les puristes des design patterns constateront que je vais m'attarder sur un patron de conception qui ne présente pas un taux d'utilisation élevé. Je rappelle avant tout que mon cours ne se veut pas un catalogue des patrons de conception, je préfère aborder les notions sur la *bonne conception*. Et puis, il existe peu de documentation française claire sur ce pattern, ce qui rend cette partie d'autant plus intéressante.
Comme pour le chapitre précédent, je vais poser un problème sur lequel j'attaquerai différents angles d'approche. Si vous le permettez, lançons-nous dans la fosse !

Le tramway

Voici un cas un peu plus original que l'ascenseur vu précédemment. J'ai nommé à la barre un système de contrôle sur un tramway. Ne vous affolez pas, je vais faire en sorte de bien vous guider pour comprendre ce que j'attends de ce système.

L'exemple qui va suivre ne correspond pas forcément à la réalité, inutile de me préciser que le tramway s'utilise différemment. Le but étant de comprendre le mini cahier des charges qui va suivre, pas de vous focaliser sur les divergences par rapport à la réalité.

Ce que l'on souhaite gérer, c'est le comportement de ce tramway selon diverses situations. Par exemple, lorsque le tramway est à l'arrêt, il est possible d'ouvrir les portes pour pouvoir sortir. Cette action n'est pas possible lorsque celui-ci est en déplacement. Le tramway se voit donc offrir différentes interactions possibles qui n'auront pas les mêmes incidences selon sa configuration.

Je vais sans plus attendre, donner les contraintes d'une manière plus formelle. Attaquons dans le vif du sujet avec un cahier des charges très simpliste. Bien qu'incomplètes, ces spécifications seront suffisantes à notre niveau.

Le cahier des charges :
Le tramway propose plusieurs actions possibles lorsqu'on est passager :

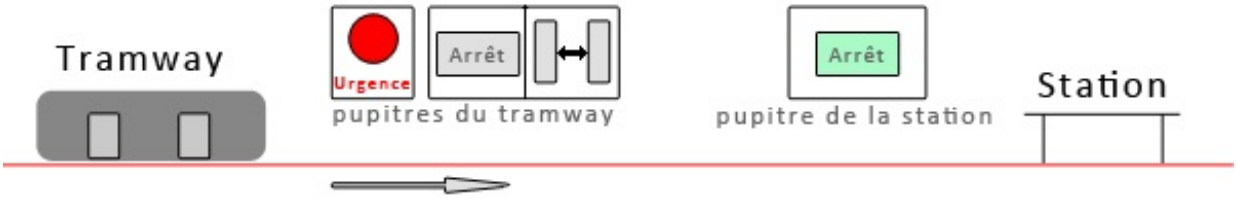
- Un bouton arrêt d'urgence pour arrêter le tramway à tout moment.
- Un bouton pour pouvoir ouvrir les portes du tramway lorsqu'il est en station.
- Un bouton pour demander l'arrêt à la prochaine station

Les stations quant à elles, ont :

- Un bouton pour prévenir le tramway qui arrive de s'arrêter pour récupérer des gens.
- Un capteur qui indique au tramway qu'il est bien positionné pour s'arrêter.
- Un compteur qui envoie une impulsion au tramway pour indiquer qu'il peut repartir lors d'une attente respectée en station.

Vous constaterez que les actions engendrées par ces boutons devraient vous rappeler quelque chose. Aller, un petit effort !
Et oui, pour ceux qui auraient suivi, il s'agit d'évènements : cette notion est très souvent rattachable au patron de conception Observateur. Il va falloir revoir le concept s'il n'est pas maitrisé, vous n'y couperez pas.

Afin d'éclaircir le fonctionnement du tramway, je vous offre le schéma suivant :



On peut voir les pupitres de commandes que proposeront le tramway et la station.

Selon la circonstance, ces boutons ne vont pas avoir le même comportement. Je vais décrire pour chaque situation comment le tramway devra réagir :
tramway en déplacement sans arrêt prévu :

- L'impulsion du compteur de station n'a aucun effet.
- Le bouton d'arrêt d'urgence stoppe le tramway.
- Le bouton pour ouvrir les portes est désactivé.
- Le bouton pour demander le prochain arrêt fonctionne (du tramway ou de l'extérieur). Une pression sur celui-ci entraine le tramway en mode arrêt imminent.

tramway en déplacement avec arrêt prévu (arrêt imminent) :

- L'impulsion du compteur de station n'a aucun effet.
- Le bouton d'arrêt d'urgence stoppe le tramway.
- Le bouton pour ouvrir les portes est désactivé.
- Le bouton pour demander le prochain arrêt est inactif (du tramway ou de l'extérieur).

tramway en arrêt de station :

- L'impulsion du compteur de station va relancer le tramway.
- Le bouton d'arrêt d'urgence stoppe le tramway.
- Le bouton pour ouvrir les portes fonctionne.
- Le bouton pour demander le prochain arrêt est actif (du tramway ou de l'extérieur). Une pression sur celui-ci entraine le tramway en mode arrêt imminent dès qu'il repartira.

tramway en arrêt d'urgence :

- L'impulsion du compteur de station n'a aucun effet.
- Les demandes pour les prochaines stations sont effacées.
- Le bouton d'arrêt d'urgence rebascule le tramway dans un fonctionnement normal. Le wagon devra donc reprendre sa situation précédente (en déplacement ou en station).
- Le bouton pour ouvrir les portes fonctionne.
- Le bouton pour demander le prochain arrêt est désactivé (du tramway ou de l'extérieur).

Aussi, le tramway a la responsabilité de gérer :

- L'ouverture de ses portes.
- La fermeture de ses portes.
- Le démarrage du wagon pour aller vers la station suite ou pour repartir suite à un arrêt d'urgence..
- L'arrêt de son wagon, uniquement lorsqu'il arrive en station ou quand l'arrêt d'urgence est enclenché.

Mais c'est une blague ton truc ?

Ne soyez pas surpris, sous des apparences très simples, de tels systèmes peuvent vite receler des difficultés. Moi-même, j'ai déjà sous-estimé ce genre de problèmes. 

Bon, si vous lisez à tête reposée vous verrez que ce système n'est pas difficile à comprendre. Il y a même beaucoup de redondances dans l'énoncé mais je préfère être clair sur ce que j'attends du système. J'aurais pu complexifier le problème mais pour éviter de vous noyer sous une masse d'informations, on se contentera de ces éléments. Vous verrez qu'on a déjà de quoi cogiter !

Pour notre étude de cas, on ne va pas s'occuper du fonctionnement des pupitres de commandes, je vais juste vous donner leurs interfaces d'utilisation. Si vous ne le saviez pas, une interface permet aussi de pouvoir simuler des bouts de codes dont on ne possède pas l'implémentation. C'est très pratique si on veut simuler des environnements dont nous ne sommes pas toujours maîtres. Les éléments que je vous fournis sont les suivants :

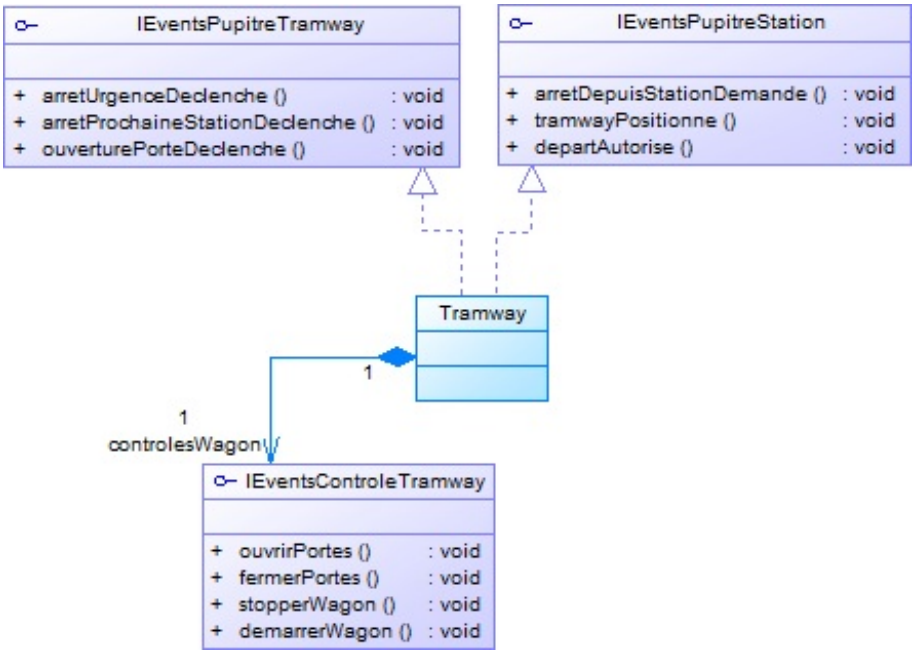
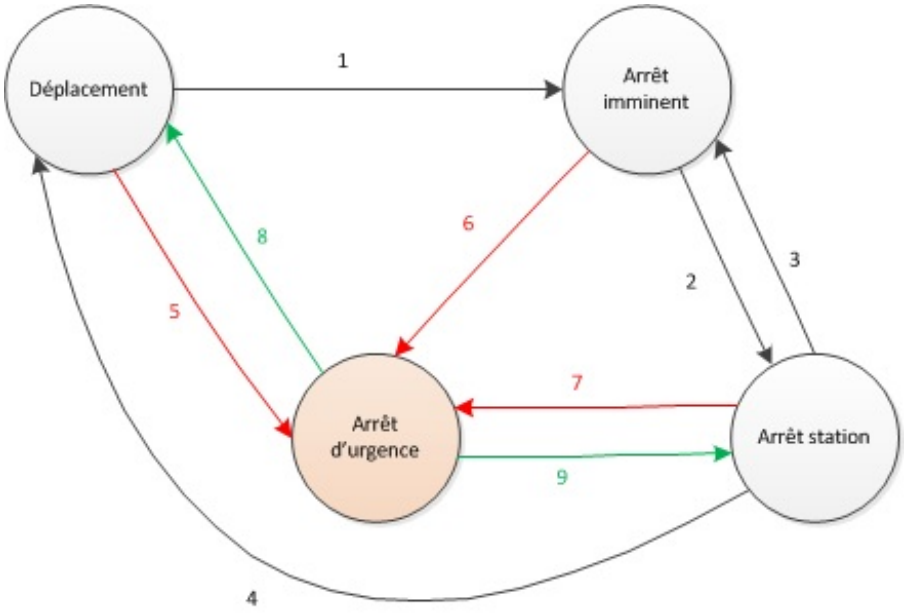


Diagramme de classes représentant les interfaces existantes. Ce n'est pas à vous de les implémenter.
Le code à fournir va juste concerner la classe Tramway.

Pour éclaircir le fonctionnement du tramway, je vous propose un diagramme d'états-transitions. Il s'agit d'une version un peu dérivée de sa représentation formelle pour faciliter sa lecture à notre cas.



Arc	Évènement(s) déclencheur(s)	Condition(s)
1	arretProchaineStationDeclenche() ***OU*** arretDepuisStationDemande()	-
2	tramwayPositionne()	-
3	departAutorise()	portes fermées demande d'arrêt
4	departAutorise()	portes fermées
5	arretUrgenceDeclenche()	-
6	arretUrgenceDeclenche()	-

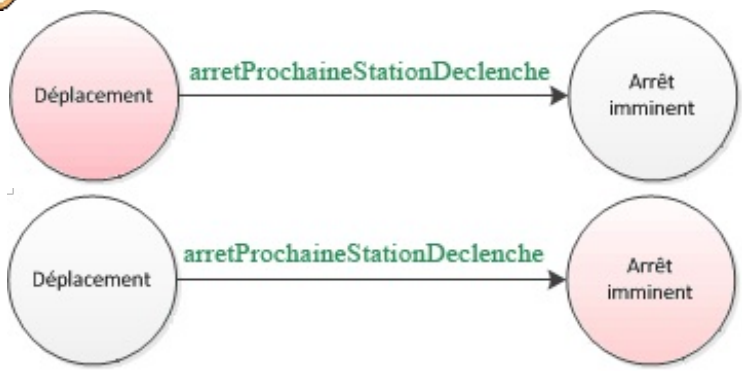
7	arretUrgenceDeclenche()	-
8	arretUrgenceDeclenche()	était en déplacement
9	arretUrgenceDeclenche()	était en station

Diagramme d'états-transitions du tramway.

Si on s'attarde un peu sur ce schéma, il n'est pas difficile de le comprendre. Je vais prendre un exemple pour que vous compreniez comment le lire.
Le tramway possède 4 états qui représentent ses différentes situations :

- Déplacement : Lorsque le tramway se déplace sur les rails.
- Arrêt imminent : Lorsque le tramway se déplace et qu'il doit s'arrêter à la prochaine station.
- Arrêt station : Lorsque le tramway est en train de déposer/récupérer des gens en station.
- Arrêt d'urgence : Lorsque le tramway a le mode "arrêt d'urgence" activé. Peut se produire à tout moment.

Maintenant, admettons que le tramway se déplace sans que personne ne réclame l'arrêt pour la prochaine station (on se situe donc sur l'état Déplacement). Tout à coup, un utilisateur va demander un arrêt pour être déposé à la station qui arrive. L'évènement déclencheur *arretProchaineStationDeclenche()* est émis par l'intermédiaire du bouton prévu à cet effet. En jetant un œil au tableau, on peut voir que l'arc numéro 1 attend cet évènement. Il est de ce fait, possible d'emprunter cet arc (ou transition) pour se retrouver dans l'état qui est pointé : Arrêt imminent. Puisque je suis généreux, je vais encore vous offrir un schéma sur ce qui se passe :



Exemple du passage de l'état "Déplacement" vers "Arrêt imminent". L'arc est emprunté car la condition est satisfaite.

C'est toujours la même logique à adopter pour passer d'état en état. Avec un peu de bon sens vous pouvez facilement comprendre ce diagramme, d'ailleurs je vous invite à le faire .

Pour les gens qui connaissent les automates à états, vous ne serez pas déboussolé, c'est exactement le même principe. D'ailleurs, le diagramme d'états-transitions que j'ai représenté plus haut s'apparente plus à un automate qu'autre chose.

Et vous dans tout ça :

Comme vous pouvez le constater, je vous ai donné pas mal d'informations. Vous n'avez qu'un seul objectif : **Gérer le wagon du tramway, ne vous préoccupez pas du reste**. Les seuls évènements dont vous serez la source, seront les ouvertures/fermetures de portes, le démarrage et le freinage. Pour les autres signaux, vous êtes juste "observateurs".

A partir d'ici vous pouvez commencer à fournir vos propres solutions. Si vous ne savez pas par où commencer, le début de la partie suivante vous aiguillera.

Bonne chance

Lançons nous

Pour bien commencer à attaquer le problème, on va implémenter les évènements déclenchés ou captés par le tramway en écrivant les interfaces respectives. On pose le socle du système pour ensuite nous intéresser à la partie essentielle de notre problème. Ainsi, on ne s'éparpille pas et on respecte les contraintes intrinsèques du système. Le code à écrire est le suivant :

Code : C++

```
class IEventsPupitreTramway
{
```

```
   public :
    virtual void arretUrgenceDeclenche() = 0;
    virtual void arretProchaineStationDeclenche() = 0;
    virtual void ouverturePorteDeclenche() = 0;
};

class IEventsPupitreStation
{
   public :
    virtual void arretDepuisStationDemande() = 0;
    virtual void tramwayPositionne() = 0;
    virtual void departAutorise() = 0;
};

class IEventsControleTramway
{
   public :
    virtual void ouvrirPortes() = 0;
    virtual void fermerPortes() = 0;
    virtual void stopperWagon() = 0;
    virtual void demarrerWagon() = 0;
};

class Tramway : public IEventsPupitreTramway, public
IEventsPupitreStation
{

   public:
    virtual void arretUrgenceDeclenche()           { /* CODE ICI */ }
    virtual void arretProchaineStationDeclenche() { /* CODE ICI */ }
    virtual void ouverturePorteDeclenche()         { /* CODE ICI */ }
    virtual void arretDepuisStationDemande()       { /* CODE ICI */ }
    virtual void tramwayPositionne()               { /* CODE ICI */ }
    virtual void departAutorise()                  { /* CODE ICI */ }

   private:
    IEventsControleTramway m_controlesWagon;
}
```

Code : Java

```
interface IEventsPupitreTramway
{
   public void arretUrgenceDeclenche();
   public void arretProchaineStationDeclenche();
   public void ouverturePorteDeclenche();
}

interface IEventsPupitreStation
{
   public void arretDepuisStationDemande();
   public void tramwayPositionne();
   public void departAutorise();
}

interface IEventsControleTramway
{
   public void ouvrirPortes();
   public void fermerPortes();
   public void stopperWagon();
   public void demarrerWagon();
}

class Tramway implements IEventsPupitreTramway,
IEventsPupitreStation
{
    public void arretUrgenceDeclenche()           { /* CODE ICI */ }
    public void arretProchaineStationDeclenche() { /* CODE ICI */ }
```

```
        public void ouverturePorteDeclenche()         { /* CODE ICI */ }
        public void arretDepuisStationDemande()       { /* CODE ICI */ }
        public void tramwayPositionne()               { /* CODE ICI */ }
        public void departAutorise()                  { /* CODE ICI */ }

        private IEventsControleTramway m_controlesWagon;
    }
```

La chose la plus intuitive consiste à écrire les comportements directement dans la classe Tramway. Pour ce cas précis, il est même probablement judicieux de procéder de cette manière. Avec une bonne appréhension du sujet, on peut facilement écrire du code propre qui corresponde au cahier des charges. Il va cependant falloir conserver des informations utiles pour que le tramway adopte le bon comportement dans toutes les situations. Par exemple, lorsqu'il sort du mode arrêt d'urgence, le wagon doit savoir quelle était son action précédente (arrêt d'un station ou en déplacement). Un autre exemple : le tramway doit savoir à certains moments si il doit s'arrêter à la prochaine station mais ce ne sont pas les seules choses à sauvegarder.

Avec toutes ces astuces données, vous devriez être à même de vous lancer dans une implémentation. Pour réussir efficacement cette épreuve, respectez point par point cette étude de cas et vérifiez que chaque élément est respecté à la lettre.

Il est temps de vous mettre au travail ! (Mais qu'est-ce que c'est que ces mines dépitées, plus vite que ça moussaillon ! ) Ne vous découragez pas, c'est en fait assez simple, pensez juste à tous les cas.

Passons à la correction. Une première version du code qui correspondrait aux différentes contraintes pourrait être :

Code : C++

```
#include <iostream>

using namespace std;

/**
* Implémentation de l'interface IEventsControleTramway pour pouvoir
faire des tests.
* On se contente juste de redéfinir les opérations avec des
affichages.
*/
class ControlesTramway : public IEventsControleTramway
{
   public:
    virtual void ouvrirPortes()  {cout << "Portes    ouvertes." <<
endl;}
    virtual void fermerPortes()  {cout << "Portes    fermées." <<
endl;}
    virtual void stopperWagon()  {cout <<  "Le    tramway    freine
jusqu'à son arrêt." << endl;}
    virtual void demarrerWagon()  {cout << "Le tramway se lance en
prenant de la vitesse." << endl;}
};

class Tramway : public IEventsPupitreTramway, public
IEventsPupitreStation
{
    public:
     /**
* Constructeur, on va considérer que le train commence à l'état
"arrêt station".
*/
     Tramway() : m_portesOuvertes(false), m_enDeplacement(false),
         m_requeteProchaineStation(false),
m_arretUrgenceEnclenche(false)
     {
         m_controlesWagon = new ControlesTramway();
     }

     virtual ~Tramway()
     {
        // DTOR
         delete m controlesWagon;
```

```
        }

        /**
* L'arrêt d'urgence doit savoir son état précédent pour adopter un
comportement.
* De plus, il faut savoir si l'évènement correspond à l'activation
ou désactivation de l'urgence.
*/
        virtual void arretUrgenceDeclenche()
        {
            m_arretUrgenceEnclenche = !m_arretUrgenceEnclenche; //   On
déclare le nouvel état
            if(!m_arretUrgenceEnclenche)
            {
                if(m_portesOuvertes) // Les portes peuvent ne pas être
ouvertes
                     m_controlesWagon->fermerPortes();
                if(m_enDeplacement)
                    m_controlesWagon->demarrerWagon();
            }
            else
            {
                m_controlesWagon->stopperWagon();
            }

        }

        virtual void arretProchaineStationDeclenche()
        {
            m_requeteProchaineStation = true;
            cout << "Appel  utilisateur  d'un  arrêt  à  la  prochaine
station depuis le wagon." << endl;
        }

        virtual void arretDepuisStationDemande()
        {
            m_requeteProchaineStation = true;
            cout << "Appel  utilisateur  d'un  arrêt  à  la  prochaine
station depuis la station." << endl;
        }

        /**
* L'ouverture des portes va s'assurer que le wagon n'est pas en
déplacement ou que le
* système est en arrêt d'urgence.
*/
        virtual void ouverturePorteDeclenche()
        {
            if(m_arretUrgenceEnclenche || !m_enDeplacement)
            {
                m_controlesWagon->ouvrirPortes(); //   On   ouvre   les
portes
                m_portesOuvertes = true;
            }
            else
            {
                cout << "Ouverture porte impossible." << endl;
            }
        }


        /**
* Le capteur de station émet un signal pour déclarer que le tramway
doit commencer sa phase
* d'arrêt si des requêtes sont à satisfaire.
*/
        virtual void tramwayPositionne()
        {
            if(m_requeteProchaineStation)
            {
```

```
                m_controlesWagon->stopperWagon();
                m_enDeplacement = false;

                m_requeteProchaineStation = false;
            }
            else
                cout << "Station passée sans s'arrêter." << endl;
        }


        virtual void departAutorise()
        {
            m_controlesWagon->fermerPortes();
            m_controlesWagon->demarrerWagon();
            m_enDeplacement = true;
        }

    private:
        IEventsControleTramway *m_controlesWagon;
        bool m_portesOuvertes;
        bool m_enDeplacement;
        bool m_requeteProchaineStation;
        bool m_arretUrgenceEnclenche;
};

int main()
{
   Tramway tram;
   // On va simuler les évènements en les appelant manuellement.

   tram.arretDepuisStationDemande();
   tram.departAutorise();
   tram.tramwayPositionne();
   tram.ouverturePorteDeclenche();
   tram.departAutorise();
   tram.tramwayPositionne();
   tram.ouverturePorteDeclenche(); // Portes impossibles à ouvrir
   tram.tramwayPositionne();
   tram.arretUrgenceDeclenche();
   tram.ouverturePorteDeclenche();
   tram.arretUrgenceDeclenche();

   tram.arretProchaineStationDeclenche();
   tram.tramwayPositionne();

   return 0;
}
```

Code : Java

```
/**
* Implémentation de l'interface IEventsControleTramway pour pouvoir
faire des tests.
* On se contente juste de redéfinir les opérations avec des
affichages.
*/
class ControlesTramway implements IEventsControleTramway
{
    public void ouvrirPortes() {System.out.println("Portes
ouvertes.");}
    public void fermerPortes() {System.out.println("Portes
fermées.");}
    public void stopperWagon() {System.out.println("Le       tramway
freine jusqu'à son arrêt.");}
    public void demarrerWagon() {System.out.println("Le  tramway  se
lance en prenant de la vitesse.");}
}
```

```
}

class Tramway implements IEventsPupitreTramway,
IEventsPupitreStation
{
    /**
* Constructeur, on va considérer que le train commence à l'état
arrêt station.
*/
    public Tramway()
    {
        // Ligne qui nécessite d'écrire une implémentation de
IEventsControleTramway
        m_controlesWagon = new ControlesTramway();

        m_portesOuvertes = false;
        m_enDeplacement = false;
        m_requeteProchaineStation = false;
        m_arretUrgenceEnclenche = false;
    }

    public void arretUrgenceDeclenche()
    {
        m_arretUrgenceEnclenche = !m_arretUrgenceEnclenche; //    On
change l'état de l'arrêt d'urgence
        if(!m_arretUrgenceEnclenche)
        {
            if(m_portesOuvertes) // Les portes ne sont pas forcement
ouvertes à ce moment là.
                m_controlesWagon.fermerPortes();
            if(m_enDeplacement)
                m_controlesWagon.demarrerWagon();
        }
        else
        {
            m_controlesWagon.stopperWagon();
        }

    }

    public void arretProchaineStationDeclenche()
    {
        m_requeteProchaineStation = true;
        System.out.println("Appel  utilisateur  d'un  arrêt  à  la
prochaine station depuis le wagon.");
    }

    public void arretDepuisStationDemande()
    {
        m_requeteProchaineStation = true;
        System.out.println("Appel  utilisateur  d'un  arrêt  à  la
prochaine station depuis la station.");
    }

    /**
* L'ouverture des portes va s'assurer que le wagon n'est pas en
déplacement ou que le
* système est en arrêt d'urgence.
*/
    public void ouverturePorteDeclenche()
    {
        if(m_arretUrgenceEnclenche || !m_enDeplacement)
        {
            m_controlesWagon.ouvrirPortes(); // On ouvre les portes
            m_portesOuvertes = true;
        }
        else
        {
            System.out.println("Ouverture porte impossible.");
        }
```

```
        }


        /**
     * Le capteur de station émet un signal pour déclarer que le tramway
     doit commencer sa phase
     * d'arrêt si des requêtes sont à satisfaire.
     */
        public void tramwayPositionne()
        {
            if(m_requeteProchaineStation)
            {
                m_controlesWagon.stopperWagon();
                m_enDeplacement = false;

                m_requeteProchaineStation = false;
            }
            else
                System.out.println("Station passée sans s'arrêter.");
        }

        public void departAutorise()
        {
            m_controlesWagon.fermerPortes();
            m_controlesWagon.demarrerWagon();
            m_enDeplacement = true;
        }

        private IEventsControleTramway m_controlesWagon;
        private boolean m_portesOuvertes;
        private boolean m_enDeplacement;
        private boolean m_requeteProchaineStation;
        private boolean m_arretUrgenceEnclenche;
    }

    public class Run {
        public static void main(String[] args) {
            Tramway tram = new Tramway();
            //  On  va  simuler  les  évènements  en  les  appelant
    manuellement.
            tram.arretDepuisStationDemande();
            tram.departAutorise();
            tram.tramwayPositionne();
            tram.ouverturePorteDeclenche();
            tram.departAutorise();
            tram.tramwayPositionne();
            tram.ouverturePorteDeclenche(); //  Portes  impossibles  à
    ouvrir
            tram.tramwayPositionne();
            tram.arretUrgenceDeclenche();
            tram.ouverturePorteDeclenche();
            tram.arretUrgenceDeclenche();

            tram.arretProchaineStationDeclenche();
            tram.tramwayPositionne();

        }
    }
```

Ce qui affiche à l'exécution :

Code : Console

```
Appel utilisateur d'un arrêt à la prochaine station depuis la station.
Portes fermées.
Le tramway se lance en prenant de la vitesse.
Le tramway freine jusqu'à son arrêt.
Portes ouvertes.
Portes fermées.
```

```
Le tramway se lance en prenant de la vitesse.
Station passée sans s'arrêter.
Ouverture porte impossible.
Station passée sans s'arrêter.
Le tramway freine jusqu'à son arrêt.
Portes ouvertes.
Portes fermées.
Le tramway se lance en prenant de la vitesse.
Appel utilisateur d'un arrêt à la prochaine station depuis le wagon.
Le tramway freine jusqu'à son arrêt.
```

Plusieurs points notables sur le code :

- Il répond parfaitement à la problématique posée.
- Il est nécessaire de conserver la plupart des statuts du tramway à un instant "t" (les différents booléens dans notre classe).
- Le code est concis et sans réelle difficulté.

Mais alors, quel est le problème ?

Souvenez vous qu'une majorité du temps est consacré à la correction de bogues ou à l'ajout de nouvelles fonctionnalités. Si dans notre cas, un ajout de fonctionnalité va se traduire par une modification de la classe tramway, il est difficile de faire autrement. On se concentrera donc plutôt sur l'aspect maintenance. Je ne sais pas pour vous, mais moi je trouve que le code que j'ai écrit est difficile à relire. En effet, les conditions et les états qui sont sauvegardés qui sont codés ne correspondent pas **explicitement** à mes spécifications. Pour moi, il est difficile de reprendre ce code car je dois m'imprégner de la logique du développeur qui n'est pas forcement évidente. Par exemple, regardez l'implémentation de la méthode ouverturePorteDemandee() : Les conditions écrites sont disséminées dans le sujet et pas si simple à retrouver. Dans le cas d'un contexte simple comme le tramway, l'intérêt de rendre plus compréhensible ce code est minime mais il faut se placer dans des cas plus complexes ou prévoir une évolution du système.

C'est pourquoi il est judicieux de réfléchir pour rendre notre code plus lisible. La solution qui serait très utile dans notre cas serait de calquer le code sur le diagramme d'états-transitions. Comment ? La première solution qui viendrait à l'esprit serait d'utiliser une énumération, on stockerait ensuite les états spécifiques du système :

Code : C++

```
class Tramway
{
   public :
    enum Etat {DEPLACEMENT, ARRET_IMMINENT, ARRET_STATION,
ARRET_URGENCE};

   private :
    Etat etatCourant;
    // ...
}
```

Code : Java

```
class Tramway
{
   public enum Etat {DEPLACEMENT, ARRET_IMMINENT, ARRET_STATION,
ARRET_URGENCE};
   private Etat etatCourant;
   // ...
}
```

Le code pour exprimer les comportements du tramway deviennent ensuite beaucoup plus naturels et lisibles. La lecture s'en retrouve grandement simplifiée, je vous laisse juger par vous même :

Code : C++

```
class Tramway : public IEventsPupitreTramway, public
IEventsPupitreStation
{

    public:
     enum Etat {DEPLACEMENT, ARRET_IMMINENT, ARRET_STATION,
ARRET_URGENCE};
     /**
* Constructeur, on va considérer que le train commence à l'état
"arrêt station".
*/
     Tramway() : m_etatCourant(ARRET_STATION),
m_portesOuvertes(false)
     {
         m_controlesWagon = new ControlesTramway();
     }

     virtual ~Tramway()
     {
        // DTOR
         delete m_controlesWagon;
     }

     /**
* L'arrêt d'urgence doit savoir son état précédent pour adopter un
comportement.
* De plus, il faut savoir si l'évènement correspond à l'activation
ou désactivation de l'urgence.
*/
     virtual void arretUrgenceDeclenche()
     {

         if(m_etatCourant != ARRET_URGENCE)
         {
            m_etatPrecedentArretUrgence = m_etatCourant;
            m_etatCourant = ARRET_URGENCE;
         }
         else
         {
             m_etatCourant = m_etatPrecedentArretUrgence;
         }

         switch(m_etatCourant)
         {
            case ARRET_URGENCE :
               m_controlesWagon->stopperWagon();
               break;

            case DEPLACEMENT :
               if(m_portesOuvertes)
                  m_controlesWagon->fermerPortes();
               m_controlesWagon->demarrerWagon();
               break;

            case ARRET_STATION :
               m_controlesWagon->fermerPortes();
               break;

            default:
               m_controlesWagon->stopperWagon();
               // Il faudrait créer un nouveau type d'exception qui
dérive d'Exception.
               throw "Etat   incohérent   sur   le   bouton   arrêt
d'urgence";
         }
     }

     virtual void arretProchaineStationDeclenche()
```

```
        {
            m_etatCourant = ARRET_IMMINENT;
            cout << "Appel utilisateur d'un arrêt à la prochaine
station depuis le wagon." << endl;
        }

        virtual void arretDepuisStationDemande()
        {
            m_etatCourant = ARRET_IMMINENT;
            cout << "Appel utilisateur d'un arrêt à la prochaine
station depuis la station." << endl;
        }

        /**
* L'ouverture des portes va s'assurer que le wagon n'est pas en
déplacement ou que le
* système est en arrêt d'urgence.
*/
        virtual void ouverturePorteDeclenche()
        {
            switch(m_etatCourant)
            {
               case ARRET_URGENCE :
                  m_controlesWagon->ouvrirPortes(); // On ouvre les
portes
                  m_portesOuvertes = true;
                  break;

               case ARRET_STATION :
                  m_controlesWagon->ouvrirPortes(); // On ouvre les
portes
                  m_portesOuvertes = true;
                  break;

               default:
                  cout << "Ouverture porte impossible." << endl;
            }
        }


        /**
* Le capteur de station émet un signal pour déclarer que le tramway
doit commencer sa phase
* d'arrêt si des requêtes sont à satisfaire.
*/
        virtual void tramwayPositionne()
        {
            switch(m_etatCourant)
            {
               case ARRET_IMMINENT :
                  m_controlesWagon->stopperWagon();
                  m_etatCourant = ARRET_STATION; // Changement d'état
                  break;

               default:
                  cout << "Station passée sans s'arrêter." << endl;
            }
        }


        virtual void departAutorise()
        {
            m_controlesWagon->fermerPortes();
            m_controlesWagon->demarrerWagon();

            if(m_etatCourant == ARRET_STATION) // Aucune requête pour
la prochaine station
                m_etatCourant = DEPLACEMENT; // Changement d'état
            // Sinon l'état a déjà été redéfini.
        }
```

```
    private:
     IEventsControleTramway *m_controlesWagon;
     Etat m_etatCourant;
     Etat m_etatPrecedentArretUrgence;
     bool m_portesOuvertes;
};
```

Code : Java

```
class Tramway implements IEventsPupitreTramway,
IEventsPupitreStation
{
    public enum Etat {DEPLACEMENT, ARRET_IMMINENT, ARRET_STATION,
ARRET_URGENCE};

    /**
* Constructeur, on va considérer que le train commence à l'état
arrêt station.
*/
    public Tramway()
    {
        // Ligne qui nécessite d'écrire une implémentation de
IEventsControleTramway
        m_controlesWagon = new ControlesTramway();
        m_etatCourant = Etat.ARRET_STATION;
        m_portesOuvertes = false;
    }

    public void arretUrgenceDeclenche() throws Exception
    {
        if(m_etatCourant != Etat.ARRET_URGENCE)
        {
           m_etatPrecedentArretUrgence = m_etatCourant;
           m_etatCourant = Etat.ARRET_URGENCE;
        }
        else
        {
            m_etatCourant = m_etatPrecedentArretUrgence;
        }

        switch(m_etatCourant)
        {
            case ARRET_URGENCE :
               m_controlesWagon.stopperWagon();
               break;

            case DEPLACEMENT :
               if(m_portesOuvertes)
                   m_controlesWagon.fermerPortes();
               m_controlesWagon.demarrerWagon();
               break;

            case ARRET_STATION :
               m_controlesWagon.fermerPortes();
               break;

            default:
               m_controlesWagon.stopperWagon();
              // Il faudrait créer un nouveau type d'exception qui
dérive d'Exception.
              throw new Exception("Etat incohérent sur le bouton
arrêt d'urgence");
        }

    }

    public void arretProchaineStationDeclenche()
```

```
        {
            m_etatCourant = Etat.ARRET_IMMINENT;
            System.out.println("Appel utilisateur d'un arrêt à la
prochaine station depuis le wagon.");
        }

        public void arretDepuisStationDemande()
        {
            m_etatCourant = Etat.ARRET_IMMINENT;
            System.out.println("Appel utilisateur d'un arrêt à la
prochaine station depuis la station.");
        }

        /**
* L'ouverture des portes va s'assurer que le wagon n'est pas en
déplacement ou que le
* système est en arrêt d'urgence.
*/
        public void ouverturePorteDeclenche()
        {
            switch(m_etatCourant)
            {
                case ARRET_URGENCE :
                    m_controlesWagon.ouvrirPortes(); // On ouvre les
portes
                    m_portesOuvertes = true;
                    break;

                case ARRET_STATION :
                    m_controlesWagon.ouvrirPortes(); // On ouvre les
portes
                    m_portesOuvertes = true;
                    break;

                default:
                    System.out.println("Ouverture porte impossible.");
            }


        }


        /**
* Le capteur de station émet un signal pour déclarer que le tramway
doit commencer sa phase
* d'arrêt si des requêtes sont à satisfaire.
*/
        public void tramwayPositionne()
        {
            switch(m_etatCourant)
            {
                case ARRET_IMMINENT :
                    m_controlesWagon.stopperWagon();
                    m_etatCourant = Etat.ARRET_STATION; //     Changement
d'état
                    break;

                default:
                    System.out.println("Station passée sans s'arrêter.");
            }

        }

        public void departAutorise()
        {
            m_controlesWagon.fermerPortes();
            m_controlesWagon.demarrerWagon();

            if(m_etatCourant == Etat.ARRET_STATION) // Aucune requête
pour la prochaine station
```

```
                m_etatCourant = Etat.DEPLACEMENT; // Changement d'état
            // Sinon l'état a déjà été redéfini.
        }

        private IEventsControleTramway m_controlesWagon;
        private boolean m_portesOuvertes;
        private Etat m_etatCourant;
        private Etat m_etatPrecedentArretUrgence;
    }
```

Je sais pas pour vous, mais je trouve ce code beaucoup plus simple à appréhender. Même si le nombre de lignes augment quantité de code augmente, ce handicap est largement compensé par sa facilité de relecture. Par la même occasion, certains booléens ont disparu pour ne garder que les états du tramway. Notez donc qu'en changeant au minimum son approche, on se permet de se décoller d'une implémentation trop ciblée et on diminue les futurs prises de têtes. De plus, il est maintenant très facile de prendre en compte un nouvel état en rajouter le cas dans un switch.

Il y a tout de même quelque chose de gênant dans ce code ne reflétant toujours pas notre diagramme d'états-transition de manière naturelle. La retranscription fait surtout ressortir les arcs et non les états en eux-mêmes. Par exemple, il est difficile de savoir quand notre tramway se retrouve dans un état spécifique. En plus, si de nouveaux états ou arcs apparaissent, la classe Tramway va vite devenir énorme et être difficilement maintenable.

Bon et alors tu proposes quoi ? De déporter le code en dehors de cette classe ?

Si vous avez pensé de la sorte, vous êtes dans le juste.
Le but est de pouvoir retranscrire fidèlement ce diagramme d'états-transitions (ou automate) en le découplant du Tramway. Seulement, il faut le faire de manière intelligente et c'est là que le design pattern État intervient.

Le pattern État nous vient en aide !

Le design pattern État porte bien son nom : Il intervient lorsqu'un diagramme d'états-transitions ou d'automates à états est possible. Le principe est simple, il doit être possible de représenter n'importe quel état et le traiter indifféremment d'un autre. Un état doit pouvoir recevoir des évènements qu'ils puissent les traiter et éventuellement muter vers d'autres états. On sent qu'une abstraction va être nécessaire pour représenter un "état" ! Voyons maintenant ce que propose ce patron de conception :

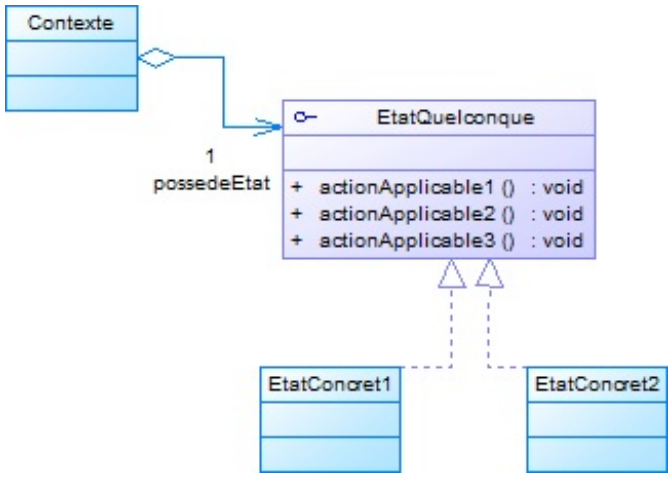
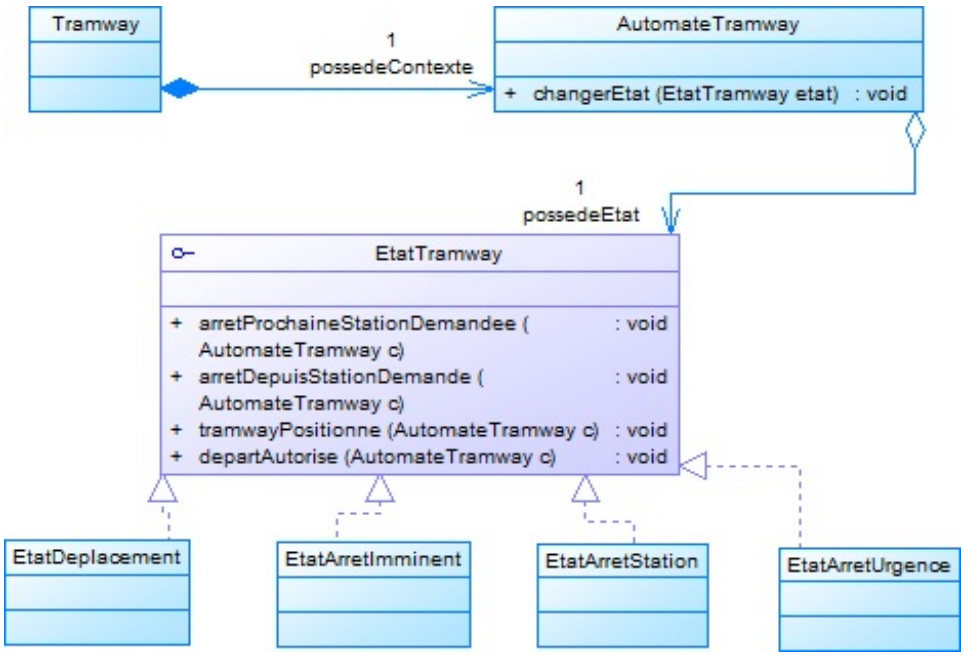


Diagramme de classes générique du design pattern état.

Effectivement, annoncé de la sorte, ce n'est pas simple de comprendre comment l'utiliser.
Je vais le transposer à notre cas pour que vous puissiez comprendre ce que signifie chaque élément du diagramme :



Application du pattern à notre cas.

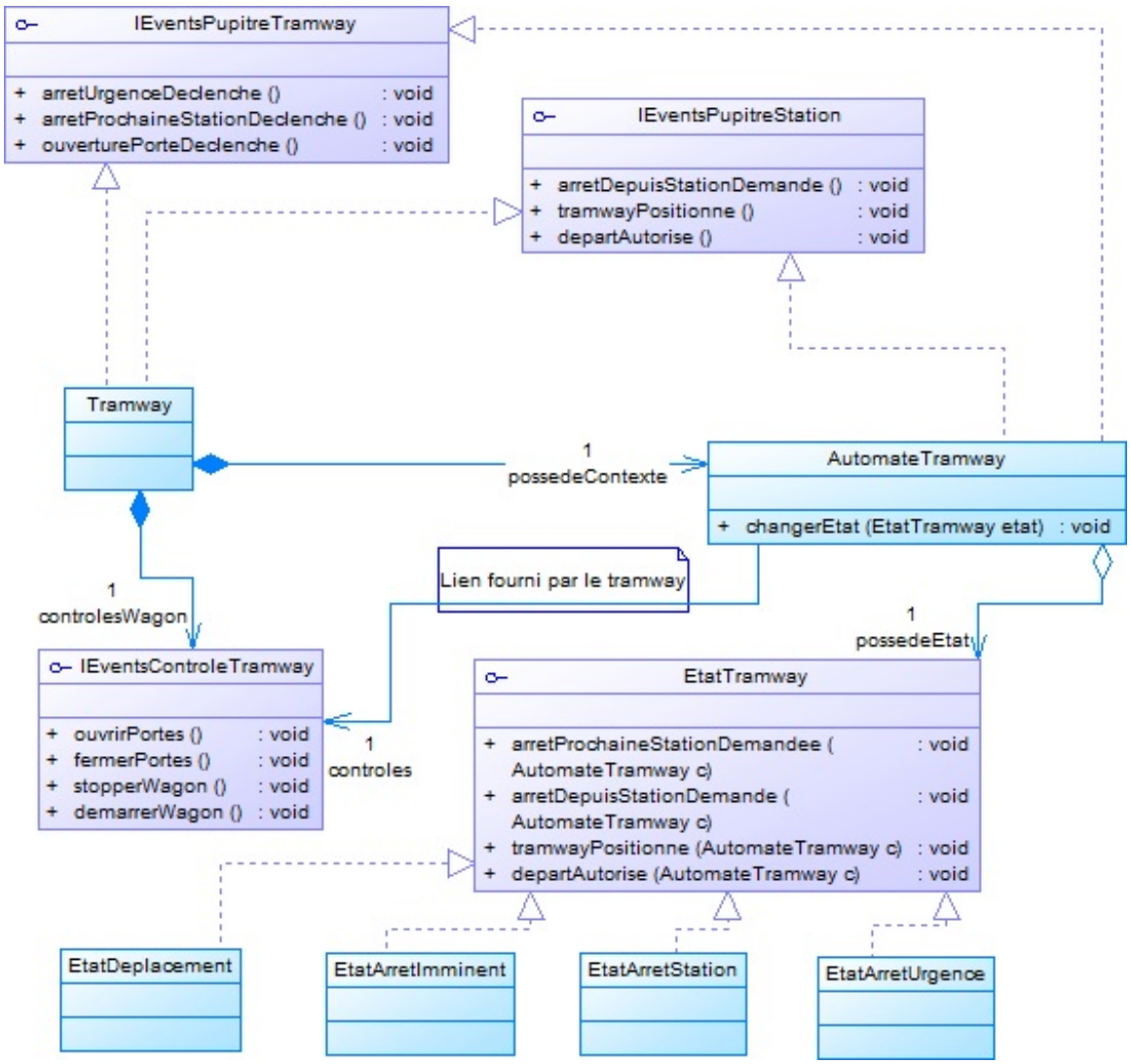
Si vous vous attardez un petit peu sur le diagramme vous comprendrez très vite le principe. Le tramway va référencer l'automate (contexte) qui lui même va manipuler des états, peu importe lesquels. Lorsque le tramway reçoit un évènement, il en informe l'automate. Cet automate qui possède un seul état à tout instant va :

- Appeler sur l'état courant l'évènement associé. L'état effectuera alors des opérations sur le tramway.
- Éventuellement changer d'état courant si l'évènement le prévoit. L'opération changerEtat de la classe AutomateTramway permet de définir le nouvel état à prendre.

Pour bien comprendre le fonctionnement, rien de mieux qu'un petit exemple. Imaginons que le tramway se situe dans l'état EtatDeplacement. Il reçoit l'évènement arretUrgenceDeclenche(). Le tramway va donc rediriger cet évènement sur son automate. L'automate va à son tour indiquer l'évènement sur son état en passant par l'interface EtatTramway. C'est l'état EtatDeplacement qui va traiter cette requête en effectuant l'arrêt du train et en transformant son état en EtatArretUrgence.

Mais pourquoi l'interface EtatTramway a besoin d'un paramètre AutomateTramway dans chacune de ses opérations ?

En fait, c'est la seule possibilité pour un état de de dire qu'il va laisser place à un autre. Il est obligé de remonter vers l'automate pour que ce dernier puisse changer vers le nouvel état à pointer. Si vous avez compris le principe, alors c'est gagné ! Voila ce que donne le diagramme UML complet :



Un lien a été rajouté à l'automate permettant à un état de pouvoir agir sur les contrôles du tramway.

Les plus attentionnés auront remarqué qu'il y a un problème sur le diagramme. L'opération changerEtat() propose un service qui ne **doit pas** être accessibles en dehors du cadre de l'automate et de ses états. Le Tramway ne **doit pas** pouvoir changer un état de l'automate, **seuls ces états sont habilités** à faire ce changement.

Sans rentrer dans les détails, il existe des moyens plus ou moins efficaces pour résoudre cet aléa. Voici une liste des possibilités :

- Laisser le code tel quel et rajouter un commentaire pour indiquer que l'opération est réservée pour un usage interne. **Cette solution est à éviter autant que possible**.
- On peut utiliser l'amitié pour donner des accès privilégiés à certaines classes. Cet artifice existe en C++ mais pas en Java par exemple.
- On peut découper l'application en packages ou namespaces et indiquer que certaines opérations sont isolées dans ces sous-couches et ne sont pas connues ailleurs. Cette fonctionnalité n'est pas présente dans tous les langages non plus.
- Enfin, on peut se servir du design pattern Visiteur qui peut s'utiliser dans tous les langages mais engendre une complexité supplémentaire. Cependant, ce pattern n'est pas prévu pour résoudre ce problème à la base.

Pour éviter une surcharge du code, je me contenterai de la première solution mais elle est à proscrire autant que possible. Si je devais choisir une solution parmi les suivantes, mon choix se porterait sur l'amitié pour le code C++ et un découpage en packages pour la solution Java (avec l'automate et ses états dans un même package isolés du reste et dont l'opération changerEtat() serait en visibilité package).

Nous voilà enfin arrivés à l'implémentation. Je place le code dans une balise secret pour éviter de rallonger la page. En effet, la quantité de code augmente significativement avec cette solution.

Code C++ : Pour des raisons de déclarations anticipées obligatoires, je n'ai pas pu regrouper le code dans un seul fichier. J'ai donc donné le code découpé pour chaque fichier d'entête et du corps correspondant :

Code header C++ :

Secret (cliquez pour afficher)

Code : C++

```
// ====================================
// FICHIER Interfaces.h
// ====================================
#ifndef INTERFACES_H_INCLUDED
#define INTERFACES_H_INCLUDED

class IEventsPupitreTramway
{
   public :
    virtual void arretUrgenceDeclenche() = 0;
    virtual void arretProchaineStationDeclenche() = 0;
    virtual void ouverturePorteDeclenche() = 0;
};

class IEventsPupitreStation
{
   public :
    virtual void arretDepuisStationDemande() = 0;
    virtual void tramwayPositionne() = 0;
    virtual void departAutorise() = 0;
};

class IEventsControleTramway
{
   public :
    virtual void ouvrirPortes() = 0;
    virtual void fermerPortes() = 0;
    virtual void stopperWagon() = 0;
    virtual void demarrerWagon() = 0;
};

class AutomateTramway; // Déclaration anticipée !
class EtatTramway
{
   public :
    virtual void arretProchaineStationDemandee(AutomateTramway &c)
= 0;
    virtual void arretDepuisStationDemande(AutomateTramway &c) =
0;
    virtual void tramwayPositionne(AutomateTramway &c) = 0;
    virtual void departAutorise(AutomateTramway &c) = 0;
    virtual void arretUrgenceDeclenche(AutomateTramway &c) = 0;
    virtual void ouverturePorteDeclenche(AutomateTramway &c) = 0;
};
#endif // INTERFACES_H_INCLUDED

#ifndef TRAMWAY_H_INCLUDED
#define TRAMWAY_H_INCLUDED

// ====================================
// FICHIER Tramway.h
// ====================================
#include "Interfaces.h"

class Tramway : public IEventsPupitreTramway, public
IEventsPupitreStation
{
   public :
    Tramway();

    virtual ~Tramway();

    IEventsControleTramway& controles();
```

```
        virtual void arretDepuisStationDemande() ;
        virtual void tramwayPositionne();
        virtual void departAutorise();
        virtual void arretUrgenceDeclenche();
        virtual void arretProchaineStationDeclenche();
        virtual void ouverturePorteDeclenche();

    private :
        AutomateTramway *m_contexte;
        IEventsControleTramway *m_controlesWagon;
};
#endif // TRAMWAY_H_INCLUDED

// ====================================
// FICHIER AutomateTramway.h
// ====================================
#ifndef AUTOMATETRAMWAY_H_INCLUDED
#define AUTOMATETRAMWAY_H_INCLUDED

#include "Interfaces.h"

class AutomateTramway : public IEventsPupitreStation, public
IEventsPupitreTramway
{
    public :
        AutomateTramway(IEventsControleTramway *referenceTram);

        IEventsControleTramway& getControlesTramway();

        void changerEtat(EtatTramway *etat);

        virtual void arretUrgenceDeclenche();
        virtual void arretProchaineStationDeclenche() ;
        virtual void ouverturePorteDeclenche();
        virtual void arretDepuisStationDemande();
        virtual void tramwayPositionne();
        virtual void departAutorise();

    private :
        IEventsControleTramway *m_refCtrlTramway;
        EtatTramway *m_possedeEtat;
};
#endif // AUTOMATETRAMWAY_H_INCLUDED

// ====================================
// FICHIER EtatDeplacement.h
// ====================================
#ifndef ETATDEPLACEMENT_H_INCLUDED
#define ETATDEPLACEMENT_H_INCLUDED

#include "Interfaces.h"

class EtatDeplacement : public EtatTramway
{
    public :
        virtual void arretProchaineStationDemandee(AutomateTramway
&c);
        virtual void arretDepuisStationDemande(AutomateTramway &c);
        virtual void arretUrgenceDeclenche(AutomateTramway &c);
        virtual void tramwayPositionne(AutomateTramway &c);
        virtual void ouverturePorteDeclenche(AutomateTramway &c);
        virtual void departAutorise(AutomateTramway &c);
};
#endif // ETATDEPLACEMENT_H_INCLUDED

// ====================================
// FICHIER EtatArretStation.h
// ====================================
#ifndef ETATARRETSTATION_H_INCLUDED
#define ETATARRETSTATION_H_INCLUDED
```

```
#include "Interfaces.h"

class EtatArretStation : public EtatTramway
{
   public :
    EtatArretStation();

    virtual void arretProchaineStationDemandee(AutomateTramway
&c);
    virtual void arretDepuisStationDemande(AutomateTramway &c);
    virtual void arretUrgenceDeclenche(AutomateTramway &c);
    virtual void departAutorise(AutomateTramway &c);
    virtual void ouverturePorteDeclenche(AutomateTramway &c);

    virtual void tramwayPositionne(AutomateTramway &c);

    private :
     bool m_demandeProchainArret;
};
#endif // ETATARRETSTATION_H_INCLUDED

// ====================================
// FICHIER EtatArretImminent.h
// ====================================
#ifndef ETATARRETIMMINENT_H_INCLUDED
#define ETATARRETIMMINENT_H_INCLUDED

#include "Interfaces.h"


class EtatArretImminent : public EtatTramway
{
   public :
    virtual void arretUrgenceDeclenche(AutomateTramway &c);
    virtual void tramwayPositionne(AutomateTramway &c);
    virtual void ouverturePorteDeclenche(AutomateTramway &c);
    virtual void arretProchaineStationDemandee(AutomateTramway
&c);
    virtual void departAutorise(AutomateTramway &c);
    virtual void arretDepuisStationDemande(AutomateTramway &c);
};
#endif // ETATARRETIMMINENT_H_INCLUDED

// ====================================
// FICHIER EtatArretUrgence.h
// ====================================
#ifndef ETATARRETURGENCE_H_INCLUDED
#define ETATARRETURGENCE_H_INCLUDED

#include "Interfaces.h"

class EtatArretUrgence : public EtatTramway
{
   public :
    EtatArretUrgence(EtatTramway *etatPrecedent) ;

    virtual void arretUrgenceDeclenche(AutomateTramway &c);
    virtual void tramwayPositionne(AutomateTramway &c);
    virtual void departAutorise(AutomateTramway &c);
    virtual void arretProchaineStationDemandee(AutomateTramway
&c);
    virtual void arretDepuisStationDemande(AutomateTramway &c);
    virtual void ouverturePorteDeclenche(AutomateTramway &c);

   private :
    EtatTramway *m_etatPrecedent;
    bool m_portesOUvertes;
};
#endif // ETATARRETURGENCE_H_INCLUDED
```

Code Corps C++ :

Secret (cliquez pour afficher)

Code : C++

```
// ====================================
// FICHIER Tramway.cpp
// ====================================
#include "Tramway.h"

#include "AutomateTramway.h"

#include <iostream>

/**
*  Implémentation  de  l'interface  IEventsControleTramway  pour
pouvoir faire des tests.
*  On  se  contente  juste  de  redéfinir  les  opérations  pour
l'utiliser.
*/
class ControlesTramway : public IEventsControleTramway
{
   public:
    virtual void ouvrirPortes()  {std::cout << "Portes  ouvertes."
<< std::endl;}
    virtual void fermerPortes()  {std::cout << "Portes   fermées."
<< std::endl;}
    virtual void stopperWagon()  {std::cout <<  "Le tramway freine
jusqu'à son arrêt." << std::endl;}
    virtual void demarrerWagon()  {std::cout << "Le   tramway   se
lance en prenant de la vitesse." << std::endl;}
};

Tramway::Tramway()
{
   m_controlesWagon = new ControlesTramway(); //    Le    pointeur
devrait être testé !
   m_contexte = new AutomateTramway(m_controlesWagon);
}

Tramway::~Tramway()
{
   delete m_controlesWagon;
   delete m_contexte;
}

IEventsControleTramway& Tramway::controles() { return
*m_controlesWagon; }

void Tramway::arretDepuisStationDemande()
{
   m_contexte->arretDepuisStationDemande();
   std::cout << "Appel  utilisateur  d'un  arrêt  à  la  prochaine
station depuis la station." << std::endl;
}

void Tramway::tramwayPositionne() { m_contexte-
>tramwayPositionne(); }
void Tramway::departAutorise() { m_contexte->departAutorise(); }
void Tramway::arretUrgenceDeclenche() {m_contexte-
>arretUrgenceDeclenche(); }
void Tramway::arretProchaineStationDeclenche()
{
     m_contexte->arretProchaineStationDeclenche();
     std::cout << "Appel  utilisateur  d'un  arrêt  à  la  prochaine
```

```
        station depuis le wagon." << std::endl;
        }
        void Tramway::ouverturePorteDeclenche() { m_contexte-
        >ouverturePorteDeclenche(); }

        // ====================================
        // FICHIER AutomateTramway.cpp
        // ====================================
        #include "AutomateTramway.h"

        #include "EtatArretStation.h"

        AutomateTramway::AutomateTramway(IEventsControleTramway
        *referenceTram) : m_refCtrlTramway(referenceTram)
        {
           m_possedeEtat = new EtatArretStation();
        }

        AutomateTramway::~AutomateTramway()
        {
           delete m_possedeEtat;
        }

        IEventsControleTramway &AutomateTramway::getControlesTramway() {
        return *m_refCtrlTramway; }

        void AutomateTramway::changerEtat(EtatTramway *etat)
        {
            delete m_possedeEtat;
            m_possedeEtat = etat;
        }

        void AutomateTramway::arretUrgenceDeclenche() { m_possedeEtat-
        >arretUrgenceDeclenche(*this);   }
        void AutomateTramway::arretProchaineStationDeclenche() {
        m_possedeEtat->arretProchaineStationDemandee(*this);}
        void AutomateTramway::ouverturePorteDeclenche() { m_possedeEtat-
        >ouverturePorteDeclenche(*this);   }
        void AutomateTramway::arretDepuisStationDemande() { m_possedeEtat-
        >arretDepuisStationDemande(*this);   }
        void AutomateTramway::tramwayPositionne() { m_possedeEtat-
        >tramwayPositionne(*this); }
        void AutomateTramway::departAutorise() { m_possedeEtat-
        >departAutorise(*this); }

        // ====================================
        // FICHIER EtatDeplacement.cpp
        // ====================================
        #include "EtatDeplacement.h"

        #include "AutomateTramway.h"
        #include "EtatArretImminent.h"
        #include "EtatArretUrgence.h"

        #include <iostream>

        void
        EtatDeplacement::arretProchaineStationDemandee(AutomateTramway &c)
        {
           c.changerEtat(new EtatArretImminent());
        }

        void EtatDeplacement::arretDepuisStationDemande(AutomateTramway
        &c)
        {
           c.changerEtat(new EtatArretImminent());
        }

        void EtatDeplacement::arretUrgenceDeclenche(AutomateTramway &c)
        {
```

```
    c.getControlesTramway().stopperWagon();
    c.changerEtat(new EtatArretUrgence(this));
}

void EtatDeplacement::tramwayPositionne(AutomateTramway &c)
{ std::cout << "Station passée sans s'arrêter." << std::endl; }

void EtatDeplacement::ouverturePorteDeclenche(AutomateTramway &c)
{ std::cout << "Ouverture porte impossible." << std::endl; }

void EtatDeplacement::departAutorise(AutomateTramway &c) { /* RIEN
*/ }


// ====================================
// FICHIER EtatArretStation.cpp
// ====================================
#include "EtatArretStation.h"

#include "AutomateTramway.h"
#include "EtatArretUrgence.h"
#include "EtatArretImminent.h"
#include "EtatDeplacement.h"

EtatArretStation::EtatArretStation() :
m_demandeProchainArret(false)
{

}

void
EtatArretStation::arretProchaineStationDemandee(AutomateTramway
&c)
{
    m_demandeProchainArret = true;
}

void EtatArretStation::arretDepuisStationDemande(AutomateTramway
&c)
{
    m_demandeProchainArret = true;
}

void EtatArretStation::arretUrgenceDeclenche(AutomateTramway &c)
{
    c.getControlesTramway().stopperWagon();
    c.changerEtat(new EtatArretUrgence(this));
}

void EtatArretStation::departAutorise(AutomateTramway &c)
{
    c.getControlesTramway().fermerPortes();
    c.getControlesTramway().demarrerWagon();
    if(m_demandeProchainArret)
        c.changerEtat(new EtatArretImminent());
    else
        c.changerEtat(new EtatDeplacement());
}

void EtatArretStation::ouverturePorteDeclenche(AutomateTramway &c)
{
     c.getControlesTramway().ouvrirPortes();
}

void EtatArretStation::tramwayPositionne(AutomateTramway &c) { /*
RIEN */ }

// ====================================
// FICHIER EtatArretImminent.cpp
// ====================================
```

```
#include "EtatArretImminent.h"

#include "AutomateTramway.h"
#include "EtatArretStation.h"
#include "EtatArretUrgence.h"

#include <iostream>

void EtatArretImminent::arretUrgenceDeclenche(AutomateTramway &c)
{
   c.getControlesTramway().stopperWagon();
   c.changerEtat(new EtatArretUrgence(this));
}

void EtatArretImminent::tramwayPositionne(AutomateTramway &c)
{
   c.getControlesTramway().stopperWagon();
   c.changerEtat(new EtatArretStation());
}

void EtatArretImminent::ouverturePorteDeclenche(AutomateTramway
&c)
{ std::cout << "Ouverture porte impossible." << std::endl; }

void
EtatArretImminent::arretProchaineStationDemandee(AutomateTramway
&c)  { /* RIEN */ }
void EtatArretImminent::departAutorise(AutomateTramway &c) { /*
RIEN */ }
void EtatArretImminent::arretDepuisStationDemande(AutomateTramway
&c)  { /* RIEN */ }

// ====================================
// FICHIER EtatArretUrgence.cpp
// ====================================
#include "EtatArretUrgence.h"

#include "AutomateTramway.h"

EtatArretUrgence::EtatArretUrgence(EtatTramway *etatPrecedent) :
m_etatPrecedent(etatPrecedent),
   m_portesOUvertes(false)
{

}

void EtatArretUrgence::arretUrgenceDeclenche(AutomateTramway &c)
{
   if(m_portesOUvertes)
      c.getControlesTramway().fermerPortes();
   c.getControlesTramway().demarrerWagon();
   c.changerEtat(m_etatPrecedent);
}

void EtatArretUrgence::tramwayPositionne(AutomateTramway &c) { /*
RIEN */ }
void EtatArretUrgence::departAutorise(AutomateTramway &c) { /*
RIEN */ }
void
EtatArretUrgence::arretProchaineStationDemandee(AutomateTramway
&c) { /* RIEN */ }
void EtatArretUrgence::arretDepuisStationDemande(AutomateTramway
&c) { /* RIEN */ }

void EtatArretUrgence::ouverturePorteDeclenche(AutomateTramway &c)
{
   c.getControlesTramway().ouvrirPortes();
   m_portesOUvertes = true;
}
```

```
// ====================================
// FICHIER main.cpp
// ====================================
#include "Tramway.h"

using namespace std;

int main()
{
   Tramway tram;
   // On va simuler les évènements en les appelant manuellement.

   tram.arretDepuisStationDemande();
   tram.departAutorise();
   tram.tramwayPositionne();
   tram.ouverturePorteDeclenche();
   tram.departAutorise();
   tram.tramwayPositionne();
   tram.ouverturePorteDeclenche(); // Portes impossibles à ouvrir
   tram.tramwayPositionne();
   tram.arretUrgenceDeclenche();
   tram.ouverturePorteDeclenche();
   tram.arretUrgenceDeclenche();

   tram.arretProchaineStationDeclenche();
   tram.tramwayPositionne();

   return 0;
}
```

Code Java :

Secret (cliquez pour afficher)

Code : Java

```
interface IEventsPupitreTramway
{
   public void arretUrgenceDeclenche();
   public void arretProchaineStationDeclenche();
   public void ouverturePorteDeclenche();
}

interface IEventsPupitreStation
{
   public void arretDepuisStationDemande();
   public void tramwayPositionne();
   public void departAutorise();
}

interface IEventsControleTramway
{
   public void ouvrirPortes();
   public void fermerPortes();
   public void stopperWagon();
   public void demarrerWagon();
}


interface EtatTramway
{
   public void arretProchaineStationDemandee(AutomateTramway c);
   public void arretDepuisStationDemande(AutomateTramway c);
   public void tramwayPositionne(AutomateTramway c);
   public void departAutorise(AutomateTramway c);
   public void arretUrgenceDeclenche(AutomateTramway c);
```

```
    public void ouverturePorteDeclenche(AutomateTramway c);
}



class ControlesTramway implements IEventsControleTramway
{
    public void ouvrirPortes() {System.out.println("Portes
ouvertes.");}
    public void fermerPortes() {System.out.println("Portes
fermées.");}
    public void stopperWagon() {System.out.println("Le     tramway
freine jusqu'à son arrêt.");}
    public void demarrerWagon() {System.out.println("Le tramway se
lance en prenant de la vitesse.");}
}

class Tramway implements IEventsPupitreTramway,
IEventsPupitreStation
{
   public Tramway()
   {
      m_controlesWagon = new ControlesTramway();
      m_contexte = new AutomateTramway(m_controlesWagon);
   }

   public IEventsControleTramway controles() { return
m_controlesWagon; }

   public void arretDepuisStationDemande()
   {
      m_contexte.arretDepuisStationDemande();
      System.out.println("Appel  utilisateur  d'un  arrêt  à  la
prochaine station depuis la station.");
   }
   public void tramwayPositionne() {
m_contexte.tramwayPositionne(); }
   public void departAutorise() { m_contexte.departAutorise(); }
   public void arretUrgenceDeclenche()
{m_contexte.arretUrgenceDeclenche(); }
   public void arretProchaineStationDeclenche()
   {
      m_contexte.arretProchaineStationDeclenche();
      System.out.println("Appel  utilisateur  d'un  arrêt  à  la
prochaine station depuis le wagon.");
   }
   public void ouverturePorteDeclenche() {
m_contexte.ouverturePorteDeclenche(); }

   private AutomateTramway m_contexte;
   private IEventsControleTramway m_controlesWagon;
}

class AutomateTramway implements IEventsPupitreStation,
IEventsPupitreTramway
{
   public AutomateTramway(IEventsControleTramway referenceTram)
   {
      m_refCtrlTramway = referenceTram;
      m_possedeEtat = new EtatArretStation();
   }

   public IEventsControleTramway getControlesTramway() { return
m_refCtrlTramway; }

   public void changerEtat(EtatTramway etat) { m_possedeEtat =
etat; }
   public void arretUrgenceDeclenche() {
m_possedeEtat.arretUrgenceDeclenche(this);   }
   public void arretProchaineStationDeclenche() {
```

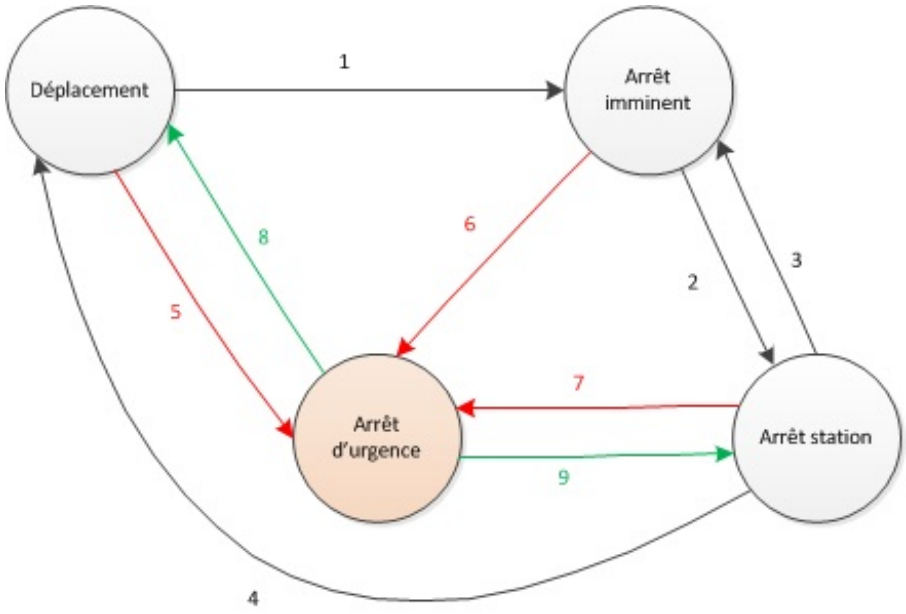
```
        m_possedeEtat.arretProchaineStationDemandee(this);}
           public void ouverturePorteDeclenche() {
        m_possedeEtat.ouverturePorteDeclenche(this);    }
           public void arretDepuisStationDemande() {
        m_possedeEtat.arretDepuisStationDemande(this);    }
           public void tramwayPositionne() {
        m_possedeEtat.tramwayPositionne(this); }
           public void departAutorise() {
        m_possedeEtat.departAutorise(this); }

           private IEventsControleTramway m_refCtrlTramway;
           private EtatTramway m_possedeEtat;
        }


        class EtatDeplacement implements EtatTramway
        {
           public void arretProchaineStationDemandee(AutomateTramway c)
           {
              c.changerEtat(new EtatArretImminent());
           }

           public void arretDepuisStationDemande(AutomateTramway c)
           {
              c.changerEtat(new EtatArretImminent());
           }

           public void arretUrgenceDeclenche(AutomateTramway c)
           {
              c.getControlesTramway().stopperWagon();
              c.changerEtat(new EtatArretUrgence(this));
           }

           public void tramwayPositionne(AutomateTramway c)
           {
              System.out.println("Station passée sans s'arrêter.");
           }

           public void ouverturePorteDeclenche(AutomateTramway c)
           {
              System.out.println("Ouverture porte impossible.");
           }

           public void departAutorise(AutomateTramway c) { /* RIEN */ }
        }

        class EtatArretUrgence implements EtatTramway
        {
           public EtatArretUrgence(EtatTramway etatPrecedent)
           {
              m_etatPrecedent = etatPrecedent;
              m_portesOUvertes = false;
           }

           public void arretUrgenceDeclenche(AutomateTramway c)
           {
              if(m_portesOUvertes)
                 c.getControlesTramway().fermerPortes();
              c.getControlesTramway().demarrerWagon();
              c.changerEtat(m_etatPrecedent);
           }

           public void tramwayPositionne(AutomateTramway c) { /* RIEN */ }
           public void departAutorise(AutomateTramway c) { /* RIEN */ }
           public void arretProchaineStationDemandee(AutomateTramway c) {
        /* RIEN */ }
           public void arretDepuisStationDemande(AutomateTramway c) { /*
        RIEN */ }

           public void ouverturePorteDeclenche(AutomateTramway c)
```

```
        {
            c.getControlesTramway().ouvrirPortes();
            m_portesOUvertes = true;
        }

        private EtatTramway m_etatPrecedent;
        private boolean m_portesOUvertes;
    }

    class EtatArretImminent implements EtatTramway
    {
        public void arretUrgenceDeclenche(AutomateTramway c)
        {
            c.getControlesTramway().stopperWagon();
            c.changerEtat(new EtatArretUrgence(this));
        }

        public void tramwayPositionne(AutomateTramway c)
        {
            c.getControlesTramway().stopperWagon();
            c.changerEtat(new EtatArretStation());
        }

        public void ouverturePorteDeclenche(AutomateTramway c)
        {
            System.out.println("Ouverture porte impossible.");
        }

        public void arretProchaineStationDemandee(AutomateTramway c)  {
    /* RIEN */ }
        public void departAutorise(AutomateTramway c) { /* RIEN */ }
        public void arretDepuisStationDemande(AutomateTramway c)  { /*
    RIEN */ }
    }

    class EtatArretStation implements EtatTramway
    {
        public EtatArretStation()
        {
            m_demandeProchainArret = false;
        }

        public void arretProchaineStationDemandee(AutomateTramway c)
        {
            m_demandeProchainArret = true;
        }

        public void arretDepuisStationDemande(AutomateTramway c)
        {
            m_demandeProchainArret = true;
        }

        public void arretUrgenceDeclenche(AutomateTramway c)
        {
            c.getControlesTramway().stopperWagon();
            c.changerEtat(new EtatArretUrgence(this));
        }

        public void departAutorise(AutomateTramway c)
        {
            c.getControlesTramway().fermerPortes();
            c.getControlesTramway().demarrerWagon();
            if(m_demandeProchainArret)
                c.changerEtat(new EtatArretImminent());
            else
                c.changerEtat(new EtatDeplacement());
        }

        public void ouverturePorteDeclenche(AutomateTramway c)
        {
```

```
            c.getControlesTramway().ouvrirPortes();
        }

        public void tramwayPositionne(AutomateTramway c) { /* RIEN */ }

        private boolean m_demandeProchainArret;
    }

    public class Run {
        public static void main(String[] args) {
            Tramway tram = new Tramway();
            // On va simuler les évènements en les appelant
manuellement.
            tram.arretDepuisStationDemande();
            tram.departAutorise();
            tram.tramwayPositionne();
            tram.ouverturePorteDeclenche();
            tram.departAutorise();
            tram.tramwayPositionne();
            tram.ouverturePorteDeclenche(); // Portes impossibles à
ouvrir
            tram.tramwayPositionne();
            tram.arretUrgenceDeclenche();
            tram.ouverturePorteDeclenche();
            tram.arretUrgenceDeclenche();

            tram.arretProchaineStationDeclenche();
            tram.tramwayPositionne();
        }
    }
```

La quantité de code a vraiment grimpé. Première constatation : La mise en place du design pattern a souvent pour conséquence d'engendrer des classes en plus, complexifiant le code. L'avantage principal ici est qu'en connaissant comment s'utilise le patron de conception État, on peut facilement trouver et corriger les problèmes car les classes ont des rôles très spécifiques et bien déterminés. L'autre aspect déterminant que l'on peut soulever, est le fait qu'on puisse traduire de manière fidèle un automate à états ou un diagramme d'états-transitions. Si une nouvelle transition ou un nouvel état apparait, il est simple de le rajouter.

Si vous trouvez que le code du dessus vous parait difficile, voilà une petite aide sur la procédure. Vous verrez qu'en fait c'est très simple à partir de mon diagramme d'états-transitions de retranscrire le modèle. Si on reprend le croquis :



Rappel du diagramme.

Il suffit de se placer sur un état particulier. Prenons par exemple l'état "Arrêt station" qui possède les origines des arcs "3", "4", et "7". En regardant le tableau correspondant dans la première partie, on peut voir les évènements déclencheurs sur chacun de

ces arcs. Chaque arc implique d'implémenter une opération pour un évènement donné. Si on suit notre exemple, l'arc "3" possède comme évènement déclencheur departAutorise(), on devra donc écrire le comportement de cette opération dans l'état "Arrêt station". Dans cette opération, le changement d'état vers "Arrêt imminent" sera à effectuer si les conditions sont respectées. En regardant le code, la retranscription apparait très distinctement. 

Pour conclure sur le design pattern État, retenez que c'est un modèle intéressant pour représenter un problème spécifique. C'est un bon exemple qui vous montre que les patrons de conception ne sont pas tous synonymes de réponses à des problèmes quotidiens. Maintenant, si vous tombez sur ce genre de situation, son implémentation ne sera plus un casse tête. N'apprenez pas la solution par cœur, sachez à quelle problématique répond ce pattern et une recherche sur internet vous indiquera la marche à suivre. Le but est justement de pouvoir rechercher facilement ces modèles. D'ailleurs, c'est ce que moi même j'ai fait pour la rédaction de ce chapitre. 

Ouf, en voilà un chapitre plutôt chargé ! Si la lecture vous a été profitable je n'en serai que plus heureux. Nous allons revenir sur des patrons de conception à utilisation plus courants.

La suite pour bientôt...

Pas de conclusion pour l'instant