Bases sur les schémas de conception

Vous trouverez quelques volumes de référence à propos des schéma de conception dans mes suggestions de lecture. Le classique est bien sûr celui du *Gang of Four*.

Quelques raccourcis :

- Général
- Idiomes
- Mauvaises pratiques

- Adaptateur
- Bâtisseur
- *Bytecode*
- Commande
- *Command Query Reponsibility Segregation* (CQRS)
- Décorateur
- Délégation
- *Dirty Flag*
- Enchaînement de méthodes
- Fabrique (*Factory*)
- Façade
- *Flyweight*
- Injection de dépendance
- Interface
- Intermédiaire (*Proxy*)
- Itérateur
- Modèle/ Vue/ Contrôleur (MVC)
  - Modèle/ Vue/ Présentation (MVP)
  - Modèle/ Vue/ Vue/ Modèle (MVVM)
- *Null Object*
- Observateur
- Ordonnanceur
- Regroupement (*Pooling*)
- Singleton
- *State*
- Stratégie
- Visiteur

- Principaux schémas de conception en parallélisme

Les schémas de conception, ou *Design Patterns* de leur nom anglais, sont des pratiques qui ont fait leurs preuves et qui sont suffisamment rodées et documentées pour être décrites par leur nom et enseignées comme telles. Ces pratiques sont estimées indépendantes du langage de programmation utilisé pour les implémenter, bien qu'on les associe habituellement aux langages de programmation orientée objet.

Un schéma de conception n'est pas du code, mais bien un descriptif d'une façon de faire récurrente et connue. Il y a donc plusieurs manières d'implémenter chacun d'entre eux. Les schémas de conception, pris individuellement, ne sont typiquement pas révolutionnaires. Au contraire, il est probable que la plupart de ces pratiques vous soient connues, parce que vous les avez déjà appliquées sans trop y penser, par réflexe ou parce que cela vous semblait être une bonne idée pour le problème que vous cherchiez à résoudre. C'est l'idée de codifier ces pratiques qui est brillante, en fait : nommer les pratiques et les décrire facilite la pédagogie, l'apprentissage et formalise le tout de manière à faire en sorte que nous évitions les écueils ou que nous fassions bien, mais pas tout à fait assez bien les choses.

> Certaines pratiques reconnues sont plutôt locales à certains langages ou groupes de langages. Dans ce cas, on tend à parler d'idiomes de programmation (*Programming Idioms*), laissant le terme schéma de conception (*Design Pattern*) pour les pratiques transférables d'un langage à l'autre. Ne considérez toutefois pas les idiomes comme des pratiques mineures : dans un cas comme dans l'autre, on parle de pratiques usitées, connues et documentées, qui ont fait leurs preuves et dont on connaît bien les avantages et les inconvénients.

Vous trouverez ci-dessous quelques liens et quelques articles, classés par catégorie, à propos des schémas de conception et de considérations connexes. Lorsque des articles de votre humble serviteur sont disponibles, vous trouverez un ou plusieurs liens vous y menant. Des articles d'autres auteurs sont aussi indiqués dans chaque cas, et il en sera de même pour des critiques du schéma de conception (lorsque j'aurai des liens appropriés à proposer).

Notez qu'il existe des schémas de conception dans plusieurs catégories de pratiques, pas seulement logicielles (on accorde habituellement le crédit de l'idée originale d'un langage de schémas de conception et de pratiques recommandables à Christopher Alexander, un architecte). Bien entendu, ici, c'est le créneau logiciel qui nous intéressera. En informatique, le livre clé sur le sujet est Design Patterns: Elements of Reusable Object-Oriented Software, un classique qui a étonnamment bien vieilli.

> Comme pour toute chose, il faut lire les divers articles et les diverses critiques ci-dessous avec discernement. Règle générale, en pratique, rien n'est complètement bon ou complètement mauvais, et il faut, face à des critiques, essayer de comprendre les raisons qui les ont provoquées et la manière par laquelle l'auteur a implémenté ou conçoit le schéma de conception.

Idées générales à propos des schémas de conception

Quelques liens et articles d'ordre général :

- Les idées de Christopher Alexander expliquées aux programmeuses et aux programmeurs OO, par Doug Lea en 1993 : http://www.patternlanguage.com/bios/douglea.htm
- Un Wiki général, servant de catalogue les schémas de conception logiciels : http://en.wikipedia.org/wiki/Category:Software_design_patterns
- Un autre Wiki général, décrivant les schémas de conception logiciels : http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29
- Un site quelque peu pédagogique sur le sujet : http://sourcemaking.com/
- Livre en ligne de pratiques du monde du jeu vidéo, colligées par Robert Nystrom : http://gameprogrammingpatterns.com/ (pour l'histoire derrière l'oeuvre, voir http://journal.stuffwithstuff.com/2014/04/22/zero-to-95688-how-i-wrote-game-programming-patterns/)
- Quelques rubriques du *Engineering Cookbook*, par Robert C. Martin, originalement publiées dans les années '90 dans *The C++ Report* :
  - une description du visiteur acyclique : http://www.objectmentor.com/resources/articles/acv.pdf
  - une réflexion sur le mauvais design et sur l'inversion de dépendances, qui discute entre autres l'importance des abstractions : http://www.objectmentor.com/resources/articles/dip.pdf
  - une réflexion sur le principe de ségrégation des interfaces : http://www.objectmentor.com/resources/articles/isp.pdf
  - une réflexion sur la question de la granularité des interfaces : http://www.objectmentor.com/resources/articles/granularity.pdf
  - une réflexion sur l'importance de la stabilité des interfaces : http://www.objectmentor.com/resources/articles/stability.pdf
- Un texte de Peter Norvig sur les schémas de conception dans les langages dynamiques : http://www.norvig.com/design-patterns/ (pour le même texte mais en format « présentation », voir http://norvig.com/design-patterns/design-patterns.pdf)
- Une réflexion avec exemples sur les schémas de conception, par Tony Marston : http://www.tonymarston.net/php-mysql/design-patterns.html
- Ce qui se produit quand un langage finit par absorber une pratique répandue à même ses propres idiomes : http://robey.lag.net/2011/04/30/dissolving-patterns.html
- De la pertinence d'apprendre les schémas de conception d'un domaine donné, par Tim Benke en 2012 : http://timbenke.de/?p=457
- Les schémas de conception seraient-ils en fait des concepts manquants dans nos langages de programmation? http://c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures
- Texte d'Adam Petersen en 2012 sur ce que sont (et ne sont pas) les schémas de conception : http://www.adampetersen.se/articles/signs/signs.htm
- Résumé graphique de certains des schémas de conception les plus connus : http://www.celinio.net/techblog/wp-content/uploads/2009/09/designpatterns1.jpg
- Rappel dialectique sur l'importance des schémas de conception, en particulier du livre séminal sur le sujet, par Robert C. Martin en 2014 : http://blog.cleancoder.com/uncle-bob/2014/06/30/ALittleAboutPatterns.html
- Robert Nystrom « revisite » certains schémas ce conception classiques dans le contexte du développement de jeux vidéos : http://gameprogrammingpatterns.com/design-patterns-revisited.html
- Marc Brooker propose, dans ce texte de 2015, ce qu'il qualifie de « défense tranquille » des schémas de conception : http://brooker.co.za/blog/2015/01/25/patterns.html
- À propos de l'adhérence (ou non) aux principes des schémas de conception, par Arne Mertz en 2015 : http://arne-mertz.de/2015/02/adherence-to-design-patterns/
- Une implémentation Java pour les schémas de conception du GoF, par Bauke Scholtz : https://gof-design-patterns.zeef.com/bauke.scholtz
- On a traité Robert C. Martin de *Pattern Pusher*, en 2015. Que veut-on dire par là? http://blog.cleancoder.com/uncle-bob/2015/07/05/PatternPushers.html
- En 2015, Egon Elbre propose de prendre le temps de réapprendre les schémas de conception : https://medium.com/@egonelbre/relearning-design-patterns-912f5094ffee

Des liens sur les schémas de conception pour les interfaces personne/ machine :

- Une bibliothèque de *Patterns* en développement d'interfaces personne/ machine : http://patterns.endeca.com/content/library/en/home.html
- Un registre général de pratiques dans ce domaine : http://ui-patterns.com/
- Un exemple, le *Blank Slate*, pour donner à un client une idée de ce qu'il obtiendra en tant que produit fini : http://ui-patterns.com/patterns/BlankSlate

> À propos de l'importance des abstractions et des risques de créer des dépendances malsaines, un exemple :
>
> - Supposons que vous développiez une interface personne machine dans laquelle il existe des méthodes pour réagir à des événements
> - Supposons que vous nommiez ces méthodes **`OnClick()`**, **`OnQuit()`**, **`OnMouseMove()`**, etc.
> - Vous constatez qu'il existe deux manières de fermer l'application, soit de passer par un menu tel que **`Fichier->Quitter`** ou par le bouton **`Quitter`**
>
> Évidemment, vous ne voulez pas dupliquer la fonctionnalité (peu de choses sont pires en développement logiciel que la réutilisation par copier/ coller).
>
> Le mauvais réflexe est de faire en sorte que l'une des deux méthodes appelle l'autre (que **`OnQuitter()`** appelle **`OnFichierQuitter()`**, disons). En effet, cela créerait une dépendance artificielle entre deux contrôles (et vous ne voulez pas ajouter de tels boulets à votre design).
>
> Le réflexe sain est de constater que la fonctionnalité de quitter l'application sera requise à plus d'un endroit et de l'extraire de ces deux contrôles pour la placer ailleurs, puis de faire en sorte que les contrôles sollicitent cette méthode tierce, plus générale. Voyez-vous pourquoi?

Des liens sur les schémas de conception pour le Web :

- Catalogue de schémas de conception pour produire des applications Web interactives, par Brad Frost : http://bradfrost.github.io/this-is-responsive/patterns.html
- Un essai de Michael Weiss sur les schémas de conception pour les applications Web : http://hillside.net/plop/plop2003/Papers/weiss-web.pdf
- Sur la place des schémas de conception dans la applications Web : http://www.e-gineer.com/v1/articles/design-patterns-in-web-programming.htm
- Des schémas de conception pour le développement de systèmes pour le Web avec Java : http://www.javaworld.com/channel_content/jw-patterns-index.html
- Sur les intermédiaires pour implémenter des messageries asynchrones : http://www1.cse.wustl.edu/~schmidt/PDF/proactor.pdf
- Une série d'exemples .NET (article du Code Project, alors lire avec discernement) : http://www.codeproject.com/KB/architecture/WebApplicationPatterns.aspx
- Des schémas de conception en JavaScript :
  - http://addyosmani.com/resources/essentialjsdesignpatterns/book/
  - http://shichuan.github.com/javascript-patterns/

Des liens sur les schémas de conception en programmation fonctionnelle :

- Textes de Neal Ford en 2012 :
  - http://www.ibm.com/developerworks/java/library/j-ft10/index.html?ca=drs-
  - http://www.ibm.com/developerworks/java/library/j-ft11/index.html?ca=drs-
- Réflexion de Mark Seemann en 2012, à l'effet que les schémas de conception sont intimement liés au paradigme qui leur donne naissance, et qui se base sur les différences entre approche fonctionnelle et approche OO pour ce faire : http://blog.ploeh.dk/2012/05/25/DesignPatternsAcrossParadigms.aspx
- Quelques schémas de conception avec Scala que Li Haoyi considère « vieux », en 2016 : http://www.lihaoyi.com/post/OldDesignPatternsinScala.html

Quelques schémas de conception pour les microservices, répertoriés par Chris Richardson : http://microservices.io/patterns/

Quelques critiques :

- Les schémas de conception logiciels ne seraient pas des *Patterns* au sens classique : http://perl.plover.com/yak/design/
- Il ne faut pas abuser des schémas de conception, pas plus qu'il ne faut les utiliser sans réflexion ni discernement :

    - http://coderoom.wordpress.com/2010/06/23/criminal-overengineering/
    - http://twitch.nervestaple.com/2011/12/04/java-patterns/
- Est-ce un idiome? Je ne le sais pas, mais en `2016`, Joe Wright discute de manières d'éliminer les alternatives (les « if ») dans les programmes : http://code.joejag.com/2016/anti-if-the-missing-patterns.html

Quelques liens pertinents vers des bibliothèques ou des catalogues de schémas de conception et de pratiques :

- Une bibliothèque de *Patterns* : http://hillside.net/patterns/
- Une bibliothèque de *Paterns* tenue à jour par Yahoo! http://developer.yahoo.com/ypatterns/
- Une bibliothèque de *Patterns* et de pratiques de Microsoft :http://msdn.microsoft.com/en-us/library/ff921345.aspx
    - en particulier, une dédiée à la technologie `.NET` : http://msdn.microsoft.com/en-us/library/ms998469
- Une bibliothèque de *Patterns* en développement d'interfaces personne/ machine : http://patterns.endeca.com/content/library/en/home.html
- Des pratiques dans le développement du noyau de Linux : http://lwn.net/Articles/336224/
- Un *Pattern* récurrent dans les noyaux, soit le noyau préemptif : http://www.bertos.org/blog/programming/embedded-programming-pattern-preemptive-kernel
- Des pratiques dans le développement de jeux : http://gameprogrammingpatterns.com/
- Des pratiques avec les interactions asynchrones dams le monde `.NET` (un article du Code Project) : http://www.codeproject.com/KB/dotnet/async_pattern.aspx
- Les pratiques pour intégration d'applications d'entreprises, selon Apache : http://camel.apache.org/enterprise-integration-patterns.html
- Quelques schémas de conception en infonuagique : http://www.cloudcomputingpatterns.org/
- Le schéma de conception *Circuit Breaker*, proposé en `2015` par Indu Alagarsamy dans une optique de sécurisation du code : http://particular.net/blog/protect-your-software-with-the-circuit-breaker-design-pattern

Idiomes de programmation

Là où les schémas de conception sont des pratiques généralement applicables dans l'ensemble des langages de programmation (typiquement ceux qui sont orientés objet), les idiomes sont des pratiques plus locales, qui dépendent de mécanismes que certains langages n'ont pas (par exemple l'idiome `RAII`, qui exige une finalisation déterministe) ou qui ont trait aux façons de faire d'un langage donné (par exemple, la création dynamique intempestive d'objets dans un langage offrant un mécanisme de collecte automatique des ordures).

Quelques raccourcis vers des idiomes connus :

- Affectation sécuritaire
- `CRTP`
- Immuabilité
- Incopiable
- *Nifty Counter*
- *Null-State*
- `NVI`
- Paramètres nommés
- `pImpl`
- `RAII`

Quelques liens d'ordre général :

- Le site https://www.programming-idioms.org/ se veut un répertoire d'idiomes de programmation dans divers langages
- Une critique du terme « idiome » au sens entendu ici, par Zed A. Shaw qui base son intervention sur le sens du mot *Idiom* en anglais : http://learncodethehardway.org/blog/AUG_19_2012.html
- Un éloge de la « programmation idiomatique », ou programmation dans le respect des idiomes, par Joel E. Kemp en `2013` : http://mrjoelkemp.com/2013/05/what-is-idiomatic-programming/
- L'impact des idiomes sur notre perception du code, par Mark Seemann en `2015` : http://blog.ploeh.dk/2015/08/03/idiomatic-or-idiosyncratic/
- Des idiomes propres à APL :
    - ensemble d'idiomes avec APL, colligés en `1987` par Alan J. Perlis et Spencer Rugaber : http://cpsc.yale.edu/sites/default/files/files/tr87.pdf
    - pratiqyes avec APL, présentation d'Aaron Hsu en `2017` : https://sway.com/b1pRwmzuGjqB30On
- Des idiomes propres à C++ :
    - http://en.wikibooks.org/wiki/C++_Programming/Idioms
    - http://en.wikibooks.org/wiki/Category:More_C%2B%2B_Idioms
    - Walter E. Brown cherche à faire intégrer à `C++ 17` une pratique qu'il nomme l'« idiome de détection de C++ » :
    - http://users.rcn.com/jcoplien/Patterns/C++Idioms/EuroPLoP98.html
    - https://github.com/lefticus/cppbestpractices/blob/master/00-Table_of_Contents.md
    - un répertoire bien fait de brefs exemples idiomatiques : http://www.cppsamples.com/
    - le polymorphisme externe, qui permet de compenser par la généricité certains défauts du polymorphisme classique : http://www1.cse.wustl.edu/~schmidt/PDF/External-Polymorphism.pdf
    - compter les pointés (la base des pointeurs intelligents) : http://www.boost.org/community/counted_body.html
    - l'idiome du booléen sécuritaire (le *Safe Bool Idiom*) : http://www.artima.com/cppsource/safebool.html
        - cet idiome est rendu caduque avec le traitement réservé à l'opérateur explicite de conversion en **`bool`** de `C++ 11`, comme le relate ce texte de Chris Sharpe en `2013` : http://chris-sharpe.blogspot.co.uk/2013/07/contextually-converted-to-bool.html
    - texte de `2014` dans lequel Paul Cechner décrit quelques idiomes de `C++ 11` et de `C++ 14` dont il se sert au quotidien : http://seshbot.com/blog/2014/08/16/modern-c-plus-plus-idioms-i-use-every-day/
    - Jason Turner collecte en un même lieu des bonnes pratiques avec C++ : https://github.com/lefticus/cppbestpractices/blob/master/00-Table_of_Contents.md
    - la relation avocat-client, qui repose sur le concept de **`friend`** :
        - https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Friendship_and_the_Attorney-Client
        - texte d'Alan R. Bolton en `2006` : http://www.drdobbs.com/friendship-and-the-attorney-client-idiom/184402053
    - le « gestionnaire de ressources universel », un idiome décrit en détail par ce texte de `2015` : http://cpp.indi.frih.net/blog/2015/07/the-universal-resource-class-pattern/
    - un idiome pour permettre une suspension temporaire et contrôlée du respect des invariants d'une classe, par Joaquín M López Muñoz en `2015` : http://bannalia.blogspot.ca/2015/06/design-patterns-for-invariant-suspension.html
- Des idiomes propres à D :
    - http://p0nce.github.io/d-idioms/
- Des idiomes propres à Go :
    - générer des identifiants uniques : http://nf.id.au/concurrency-patterns-a-source-of-unique-numbe
    - des pratiques en lien avec la concurrence : http://blog.golang.org/2010/09/go-concurrency-patterns-timing-out-and.html
- Idiomes propres à Haskell :
    - la catégorie, décrite par Gabriel Gonzales en `2012` : http://www.haskellforall.com/2012/08/the-category-design-pattern.html
- Idiome propres à J, essentiellement représentés par la fonction **`under`** de ce langage, et analogue direct de l'idiome `RAII`, mais présenté non pas comme un inverse mais bien comme un *obverse*. Textes de James Hague en `2012` :
    - http://prog21.dadgum.com/121.html
    - http://prog21.dadgum.com/122.html
    - http://prog21.dadgum.com/141.html
- Des idiomes propres à Java :
    - une liste dénotant des idiomes pris au sens de « pratiques types » : http://www.c2.com/cgi/wiki?JavaIdioms
- Des idiomes propres à Kotlin :
    - inventaire de bonnes pratiques avec Kotlin, colligé par Philipp Hauer : https://blog.philipphauer.de/idiomatic-kotlin-best-practices/
- Idiomes avec Python :
    - d'une perspective basée sur la réalisation de tests, par Rahul Verma et Chetan Giridhar : http://dpip.testingperspective.com/
    - décrits par Ajay Kumar N : http://pypix.com/tools-and-tips/design-patterns-beginners/
    - texte de `2014` par lequel David Taylor expose dix idiomes qu'il aurait aimé connaître plus tôt : http://prooffreaderplus.blogspot.ca/2014/11/top-10-python-idioms-i-wished-id.html
    - diaporama sur les idiomes de programmation dans ce langage : http://www.scribd.com/doc/39946630/Python-Idioms
- Des idiomes propres à Ruby :
    - des objets pour représenter des cas atypiques, par Matheus Lima en `2014` : http://www.matheuslima.com/ruby-special-case-objects/
- Des idiomes propres à Scala :
    - http://www.artima.com/scalazine/articles/selfless_trait_pattern.html
    - http://www.artima.com/weblogs/viewpost.jsp?thread=267933
    - http://www.artima.com/scalazine/articles/stackable_trait_pattern.html
    - http://debasishg.blogspot.com/2010/07/refactoring-into-scala-type-classes.html
    - http://www.artima.com/weblogs/viewpost.jsp?thread=275135

Affectation sécuritaire

L'affectation en C++ est une opération qui peut être complexe à implémenter. Heureusement, Herb Sutter (sur la base de techniques mises de l'avant par Jon Kalb si je ne m'abuse) nous a montré que, si nous avons implémenté le constructeur de copie, la destruction et la méthode **`swap()`**, alors l'affectation peut être implémentée de manière simple et sécuritaire.

Supposons une classe **`X`** dont la structure est, à la base, telle que proposé à droite.

Remarquez qu'un **`X`** est responsable de son attribut **`tab`**, ce qui est rendu visible par le constructeur paramétrique et par le destructeur de cette classe.

```
#include <algorithm>
class X {
   int *tab;
   std::size_t n;
public:
   X(std::size_t n) : tab{ new int[n] }, n{ n } {
   }
   X(const X &autre) : tab{ new int[autre.n] }, n{ n } {
      using std::copy;
      copy(autre.tab, autre.tab + autre.n, tab);
   }
   ~X() {
      delete [] tab;
   }
   // etc.
};
```

Une **mauvaise** implémentation de l'affectation pour un **`X`** serait celle à droite.

En effet, cette implémentation ne fonctionne pas dans le cas d'un programme tel que :

```
X a(3);
a = a;
```

```
#include <algorithm>
class X {
   int *tab;
   std::size_t n;
public:
   X(std::size_t n) : tab{ new int[n] }, n{ n } {
   }
   X(const X &autre) : tab{ new int[autre.n] }, n{ autre.n } {
      using std::copy;
      copy(autre.tab, autre.tab + autre.n, tab);
```

car la destination (**`*this`**) détruit, en éliminant ses propres états, la source. De plus, si le **`new[]`** échoue, nous avons ici un vilain problème de sécurité, la destination étant détruite mais jamais remplacée par de nouveaux états – on a alors un objet de destination dans un état incorrect, un bris patent d'encapsulation..

```
   }
   ~X() {
      delete [] tab;
   }
   X& operator=(const X &autre) {
      using std::copy;
      delete [] tab;
      tab = new int[autre.n];
      n = autre.n;
      copy(autre.tab, autre.tab + autre.n, tab);
      return *this;
   }
   // etc.
};
```

Une variante correcte, mais beaucoup plus lourde, serait celle proposée à droite. Dans ce cas, si une exception survient lors du **`new[]`**, alors celle-ci filtre jusqu'au code client et l'objet demeure cohérent.

Cela demeure une approche quelque peu artisanale, à repenser pour chaque opérateur d'affectation.

```
#include <algorithm>
class X {
   int *tab;
   std::size_t n;
public:
   X(std::size_t n) : tab{ new int[n] }, n{ n } {
   }
   X(const X &autre) : tab{ new int[autre.n] }, n{ n } {
      using std::copy;
      copy(autre.tab, autre.tab + autre.n, tab);
   }
   ~X() {
      delete [] tab;
   }
   X& operator=(const X &autre) {
      int *p = new int[autre.n];
      using std::copy;
      copy(autre.tab, autre.tab + autre.n, p);
      delete [] tab;
      tab = p;
      n = autre.n;
      return *this;
   }
   // etc.
};
```

Une « amélioration » à ce schème serait de ne pas réaliser l'allocation de mémoire et les copies de contenu dans le cas où un objet est affecté à un autre.

Cependant, l'irritant de cette « amélioration » est que chaque appel à cet opérateur implique un test (un **`if`**), donc que tous paient pour éviter un problème relativement rare, un cas pathologique.

```
#include <algorithm>
class X {
   int *tab;
   std::size_t n;
public:
   X(std::size_t n) : tab{ new int[n] }, n{ n } {
   }
   X(const X &autre) : tab{ new int[autre.n] }, n{ n } {
      using std::copy;
      copy(autre.tab, autre.tab + autre.n, tab);
   }
   ~X() {
      delete [] tab;
   }
   X& operator=(const X &autre) {
      if (*this != &autre) {
         int *p = new int[autre.n];
         using std::copy;
         copy(autre.tab, autre.tab + autre.n, p);
         delete [] tab;
         tab = p;
         n = autre.n;
      }
      return *this;
   }
   // etc.
};
```

L'idiome d'affectation sécuritaire a plusieurs avantages sur les implémentations aritsanales :

- Il est simple, au sens où il s'exprime simplement, en peu de mots, et de manière homogène peu importe le type
- Il est sécuritaire en tout temps, même si la source et la destination sont un seul et même objet
- Il n'implique pas de test supplémentaire pour éviter le cas pathologique de l'affectation sur soi
- Il réduit le couplage dans le code, en réduisant la quantité de code redondant, du fait qu'il exploite des opérations qui seront déjà implémentées (constructeur de copie, destructeur) et une autre opération, **`swap()`**, qui peut pratiquement toujours être de complexité constante, $O(1)$, et se faire sans risque de lever d'exceptions

Voici comment cet idiome se présente en pratique :

- Une méthode **`swap()`** doit être implémentée pour permuter les états de deux instances du même type. En général, cette méthode sera $O(1)$ et se fera sans lever d'exceptions, du fait que la plupart des objets pour lesquels on souhaite implémenter la Sainte-Trinité sont responsables de ressources, typiquement (mais pas nécessairement) de pointeurs, et que permuter des pointeurs (qui sont des primitifs) ne lève pas d'exceptions
- L'affectation en tant que telle s'implémente :
  - en construisant une copie anonyme du paramètre reçu par l'opération d'affectation
  - en permutant les états de cette temporaire anonyme avec ceux de l'objet de destination, et
  - en détruisant implicitement les états de l'objet anonyme (celui-ci n'ayant pas de nom, il expirera suite à l'exécution de cette expression.

```
#include <algorithm>
class X {
   int *tab;
   std::size_t n;
public:
   X(std::size_t n) : tab{ new int[n] }, n{ n } {
   }
   X(const X &autre) : tab{ new int[autre.n] }, n{ n } {
      using std::copy;
      copy(autre.tab, autre.tab + autre.n, tab);
   }
   ~X() {
      delete [] tab;
   }
   void swap(X &autre) noexcept {
      using std::swap;
      swap(tab, autre.tab);
      swap(n, autre.n);
   }
   X& operator=(const X &autre) {
      X{ autre }.swap(*this);
      return *this;
   }
   // etc.
};
```

Ce faisant, le coût d'une affectation est égal à la somme du coût d'une copie (ce qui est normal, puisqu'il faut copier les états de la source à la destination), du coût d'un nettoyage (ce qui est normal, puisqu'il faut nettoyer les états de la destination avant affectation) et d'un coût constant, celui des permutations d'états.

Dans une brillante présentation de 2014, Sean Parent a mis de l'avant qu'on peut faire encore mieux pour un type déplaçable, donc implémentant la sémantique de mouvement. Dans un tel cas, il est possible d'exprimer l'affectation comme suit :

```
class X {
   // ...
public:
   X(const X&); // constructeur de copie
   X(X&&); // constructeur de mouvement
   X& operator=(const X &autre) {
      X temp = autre; // copie
      *this = std::move(temp); // mouvement
      return *this;
   }
   // ...
};
```

Pour les types déplaçables, il est possible de tirer un (léger) gain de vitesse d'une copie suivie d'un mouvement (le mouvement impliquant typiquement deux opérations machine par état à déplacer) en comparaison avec une copie suivie d'une permutation (qui implique typiquement trois opérations machine par état à déplacer).

Quelques textes d'autres sources :

- L'idiome ramené à l'essentiel : http://www.cppsamples.com/common-tasks/copy-and-swap.html

- À propos de la permutation de valeurs : http://www.cppsamples.com/common-tasks/swap-values.html
- À propos de la permutation de conteneurs : http://www.cppsamples.com/common-tasks/swap-containers.html

CRTP

L'idiome CRTP (pour *Curiously Recurring Template Pattern*), qu'on devrait à James O. Coplien dans une édition de 1995 du *C++ Report*, est une utilisation surprenante de la généricité, par laquelle le nom d'un enfant est utilisé dans la définition du nom de son parent (générique). Concrètement, pour une classe générique **B<T>**, l'idiome CRTP définit des enfants **D<B<D>>**, donc utilise le nom de l'enfant **D** en lieu et place du type générique du parent (le type **T** dans **B<T>**).

Cet idiome a un nombre étonnant d'applications pertinentes.

Quelques textes de votre humble serviteur :

- Une application bien connue de l'idiome CRTP, nommée le Truc de Barton-Nackman : ../Divers--cplusplus/Truc-Barton-Nackman.html
- Application connexe de l'idiome, la sélection de parent : ../Divers--cplusplus/Selecteur-classe-parent.html
- autre application connexe de l'idiome, l'enchaînement de parents : ../Divers--cplusplus/Enchainement-parents.html
- Ce texte sur les singletons montre aussi des applications de l'idiome CRTP : ../Divers--cplusplus/CPP--Singletons.html
- Ce texte sur les propriétés contient aussi des applications de l'idiome CRTP : ../Divers--cplusplus/Proprietes.html
- Ce texte sur l'automatisation de la génération de tests unitaires contient, lui aussi, des applications de l'idiome CRTP : Auto-Tests-Unitaires.html
- Ce texte sur montrant comment on peut presque itérer sur des bits à l'aide d'un intermédiaire contient des applications de l'idiome CRTP : ../Divers--cplusplus/Sequence-bits.html
- Appliquer CRTP avec des *templates* variadiques? ../Divers--cplusplus/templates_variadiques_CRTP.html

Quelques textes d'autres sources :

- Un Wiki au sujet de cet idiome : http://en.wikipedia.org/wiki/Curiously_Recurring_Template_Pattern
- Une application de CRTP pour réaliser une forme d'enchaînement polymorphique statique, par Marco Arena en 2012 : http://marcoarena.wordpress.com/2012/04/29/use-crtp-for-polymorphic-chaining/
- Ce texte de 2013 : http://talesofcpp.fusionfenix.com/post-12/episode-eight-the-curious-case-of-the-recurring-template-pattern
- Une autre perspective sur cet idiome, vu comme une délégation de traitement aux classes dérivées : http://www.cppsamples.com/common-tasks/delegate-behavior-to-derived-classes.html
- Utiliser CRTP comme levier pour une forme d'évaluation paresseuse, texte de 2017 par Rainer Grimm : http://www.modernescpp.com/index.php/c-is-still-lazy
- Série d'articles sur CRTP proposée par Jonathan Boccara en 2017 :
  - http://www.fluentcpp.com/2017/05/12/curiously-recursive-template-pattern/
  - http://www.fluentcpp.com/2017/05/16/what-the-crtp-brings-to-code/
  - http://www.fluentcpp.com/2017/05/19/crtp-helper/
  - réaliser des Mixin avec CRTP : https://www.fluentcpp.com/2017/12/12/mixin-classes-yang-crtp/
- En 2018, Jonathan Boccara examine la transformation d'une hiérarchie de classes polymorphiques en une application de l'idiome CRTP : https://www.fluentcpp.com/2018/05/22/how-to-transform-a-hierarchy-of-virtual-methods-into-a-crtp/
- Utiliser CRTP pour choisir des segments de code par plateforme dès la compilation, technique proposée par Michael Klimenko en 2018 : https://mklimenko.github.io/english/2018/07/02/platform-dependent-crtp/

Critiques :

- En 2018, Fei Teng fait part de ses difficultés lors de l'exportation de classes avec parents appliquant CRTP à partir d'une DLL : https://dreamdota.com/dll-exporting-a-crtp-base/

Immuabilité

L'immuabilité est une manière de concevoir des classes de manière à rendre leurs instances impossibles à modifier une fois construites.

On utilise souvent cet idiome dans les langages où il est impossible de définir des instances constantes (comme Java et C#, par exemple). En effet, prenant l'exemple de Java, si un programme définit un **`final X x = new X();`** pour une classe **X** donnée, c'est la référence **x** qui est constante (elle ne peut mener vers une autre instance de **X** que celle qui lui est donnée à l'initialisation), pas le référé (celui-ci est pleinement modifiable).

Plusieurs types clés des modèles Java et .NET sont immuables pour cette raison (le cas canonique dans chaque cas est la classe **String**).

Quelques textes de votre humble serviteur :

- Une discussion de l'importance des constantes dans les langages orientés objet, et de l'immuabilité comme palliatif aux langages qui montrent une déficience en ce sens : ../Divers--cplusplus/Importance-constantes.html

Quelques textes d'autres sources :

- Un Wiki sur le sujet : http://en.wikipedia.org/wiki/Immutable_object

Incopiable

L'idiome des classes incopiables est un idiome important du langage C++, du fait que la Sainte-Trinité (le constructeur de copie, l'affectation et la destruction) est générée automatiquement pour toute classe. Dans les cas où un objet est responsable d'une ressource, la question de la gestion de sa copie entre en ligne de compte : partagera-t-on cette ressource? La copiera-t-on? La clonera-t-on?

Quand on n'est pas en mesure de trancher, ou quand la copie de l'objet responsable de la ressource serait un problème, l'idiome de la classe incopiable entre en jeu.

Quelques textes de votre humble serviteur :

- L'implémentation que je fais de ce concept : ../Divers--cplusplus/Incopiable.html
- Le site h-deb est rempli d'exemples montrant comment appliquer cet idiome.

Quelques textes d'autres sources :

- L'idiome, à ma connaissance, a été développé par les gens de *Boost* : http://www.boost.org/doc/libs/1_46_1/libs/utility/utility.htm#Class_noncopyable

Nifty Counter (aussi connu sous le nom de *Schwarz Counter*)

Cet idiome décrit la tenue à jour d'un compteur de références partagé sur un objet global, de manière à ce que cet objet soit créé lorsqu'il est demandé une première fois, détruit lorsqu'il est relâché la dernière fois, et partagé entre-temps. Ceci peut entre autres être utile pour des objets globaux tels que **std::cout** en C++. Cette technique va comme suit.

Un fichier d'en-tête déclare l'objet global à partager (ici : **Serveur**), qui a des attributs de classe qu'il importe d'initialiser une seule fois de manière non-triviale, et définit une variable globale statique (invisible à l'édition des liens) à même le fichier d'en-tête. Cette dernière, nommée ici Initialiseur, servira à gérer le compteur de références en question.

```
#ifndef SERVEUR_H
#define SERVEUR_H
class Serveur {
   friend struct Initialiseur;
   // ... attributs de classe ...
public:
   Serveur();
   // services ...
};
struct Initialiseur {
   Initialiseur();
   ~Initialiseur();
} __ze_initialiseur; // la variable globale
#endif
```

Un fichier source, où le constructeur d'un **Initialiseur** incrémentera (avec prudence) un compteur global interne au fichier et invisible à l'édition des liens, et où le destructeur d'**Initialiseur** décrémentera (avec tout autant de prudence) le même compteur.

Si, à la construction, un **Initialiseur** fait le constat que c'est lui qui a fait passer le compteur de 0 à 1, alors il assurera l'initialisation des attributs globaux de **Serveur**.

Si, à la destruction, un **Initialiseur** fait le constat que c'est lui qui a fait passer le compteur de 1 à 0, alors il assurera le nettoyage des attributs globaux de **Serveur**.

> Notez que, pour être *Thread-Safe*, une implémentation doit aussi s'assurer que les états globaux ne soient pas utilisés avant que leur initialisation n'ait été complétée, une tâche qui n'est pas banale; le *Nifty Counter* est un outil qui prédate les ordinateurs multi-coeurs.

```
#include "Serveur.h"
#include <atomic>
static atomic<long> ze_nifty_counter { 0L }; // le compteur en question
Initialiseur::Initialiseur() {
   long avant = ze_nifty_counter.load();
   while (ze_nifty_counter.compare_exchange_weak(avant, avant + 1))
      ;
   if (avant == 0) { // si c'est moi qui ai changé le compteur de 0 à 1
      // initialiser les états globaux de Serveur...
   }
}
Initialiseur::~Initialiseur() {
   long avant = ze_nifty_counter.load();
   while (ze_nifty_counter.compare_exchange_weak(avant, avant - 1))
      ;
   if (avant == 1) { // si c'est moi qui ai changé le compteur de 1 à 0
      // nettoyer les états globaux de Serveur...
   }
}
// ...
```

Ainsi, chaque fichier source qui inclura **Serveur.h** inclura aussi une variable globale gérant un même *Nifty Counter*. Le problème de « qui est responsable de l'initialisation des états » est résolu de par la gestion du *Nifty Counter*, elle-même implémentée par les variables globales implicitement ajoutées à chaque unité de traduction.

Quelques textes d'autres sources :

- https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Nifty_Counter

État nul, ou *Null-State*

L'idiome d'état nul représente la capacité de représenter un état par défaut pour un type donné, et de ramener un objet de ce type à cet état. L'état nul est un état reconnaissable en tant que tel, et deux instances d'un état nul pour un même type devraient typiquement être égales au sens du contenu.

Quelques textes d'autres sources :

- Andrzej Krzemieński a écrit des textes sur ce sujet en lien avec la syntaxe d'initialisation uniforme de `C++ 11` :
  - http://akrzemi1.wordpress.com/2011/12/06/null-state-part-i/
  - http://akrzemi1.wordpress.com/2011/12/07/null-state-part-ii/

Interface non-virtuelle, ou `NVI`

Aussi appelé *Template Method*, cet idiome implique :

- Une classe parent offrant des services polymorphiques; et
- Au moins une classe enfant implémentant des services.

Dans un tel cas, l'idiome `NVI` recommande que :

- L'interface publique du parent soit faite de méthodes qui ne sont pas polymorphiques;
- Les services polymorphiques, abstraits ou non, soient protégés;
- Que les services du parent encadrent, lors d'un appel, ceux des enfants.

Par `NVI`, il est possible de centraliser en un même lieu (les méthodes du parent) des services `(i)` de saisie de statistiques d'utilisation (nombre d'appels, durée d'exécution des appels des services implémentés par les enfants, ce genre de truc), `(ii)` de validation des préconditions des fonctions (p. ex. : ce paramètre sera non-nul), `(iii)` de validation des postconditions (p. ex. : l'état de la variable globale **`x`** sera identique avant et après l'appel à la fonction), *etc*.

L'exemple à droite met ceci en relief : le parent encadre l'appel aux services de l'enfant à partir d'une interface concrète, bien que les services de l'enfant soient eux-mêmes sollicités par polymorphisme.

```cpp
#include <chrono>
using namespace std; // bof
using namespace std::chrono; // re-bof
struct TimedOperation {
   system_clock::duration proceder() {
      auto avant = system_clock::now();
      proceder_impl();
      return system_clock::now() - avant;
   }
   virtual ~TimedOperation() = default;
protected:
   virtual void proceder_impl() = 0;
};
class GrosCalcul : public TimedOperation {
   void proceder_impl(); // quelque chose qui prend du temps
   // ...
};
```

Un exemple semblable avec C# serait :

```cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace z
{
   abstract class OpérationMinutée
   {
      public abstract string Nom
      {
         get;
      }
      public void Exécuter()
      {
         System.Diagnostics.Stopwatch minuterie = new System.Diagnostics.Stopwatch();
         minuterie.Start();
         ExécuterImpl();
         minuterie.Stop();
         Console.WriteLine("Exécution de {0} en {1} ms", Nom, minuterie.ElapsedMilliseconds);
      }
      protected abstract void ExécuterImpl();
   }
   class OpérationÀTester : OpérationMinutée
   {
      int tempsExécution;
      public override string Nom
      {
         get { return "Petit test"; }
      }
      protected override void ExécuterImpl()
      {
         System.Threading.Thread.Sleep(tempsExécution);
      }
      public OpérationÀTester(int dodo)
      {
         tempsExécution = dodo;
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         OpérationMinutée op = new OpérationÀTester(500);
         op.Exécuter();
      }
   }
}
```

Quelques textes :

- Un Wiki sur le sujet : http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Non-Virtual_Interface
- Texte de Herb Sutter sur le sujet, en `2001` : http://www.gotw.ca/publications/mill18.htm
- Une réflexion de Sumant Tambe sur cet idiome, texte de `2007` : http://cpptruths.blogspot.com/2007/04/non-virtual-interface-nvi-idiom-and.html
- En `2016`, Edaqa Mortoray fait une critique de l'idiome `NVI` : https://mortoray.com/2016/08/29/a-pattern-bandage-for-messy-virtual-functions/

Paramètres nommés

Il arrive que les paramètres d'une fonction, par exemple ceux d'un constructeur, entraînent une confusion chez les clients. Pensez par exemple à une classe **`Rectangle`** comme la suivante (ébauche) :

```cpp
class Rectangle {
   // ...
public:
   constexpr Rectangle(int,int);
   // ...
};
```

Dans le constructeur paramétrique, les noms des paramètres sont documentaires, mais l'utilisation d'une telle classe est une source d'erreurs. Par exemple, dans le programme ci-dessous, sur la seule base du code source, il n'est pas clair que le **`Rectangle`** nouvellement créé soit de largeur `7` et de hauteur `3` ou de largeur `3` et de hauteur `7` :

```cpp
int main() {
   Rectangle rect{ 3,7 };
   // ...
}
```

Certains langages évitent ces ambiguïtés en permettant au code client de nommer les paramètres. Ceci introduit une forme de distance entre l'ordre des paramètres dans le code appelant et dans la signature de la fonction appelée. En C++, il est possible d'en arriver à un résultat semblable d'au moins deux manières.

Une des approches est de permettre l'enchaînement d'initialisations nommées, comme le montre l'exemple à droite.

Le principal défaut de cette approche est qu'il brise l'encapsulation, au sens où il est possible d'avoir un **`Rectangle`** qui ne soit que partiellement initialisé, mais l'approche peut être convenable dans les cas où les états par défaut de l'objet nouvellement construit sont adéquats pour la classe.

```cpp
class Rectangle {
   // ...
   int hauteur,
       largeur;
```

Un irritant accessoire (quoique...) de cette approche est qu'elle est inefficace, au sens où l'objet doit être construit dans sa version par défaut, puis ses états sont remplacés par des versions plus proches des attentes du code client. Ce coût est banal ici, avec une classe dont les états sont constitués d'une paire d'entiers, mais peut être nettement plus douloureux à absorber lorsque les états ne sont pas triviaux (une **`std::string`**, un vecteur, une autre sorte d'objet complexe à initialiser, ...)

```
public:
   Rectangle();
   Rectangle& Largeur(int valeur) {
      // ... valider? ...
      largeur = valeur;
   }
   Rectangle& Hauteur(int valeur) {
      // ... valider? ...
      hauteur = valeur;
   }
};
// ...
int main() {
   auto rect = Rectangle{}.Largeur(3).Hauteur(7);
   // ...
}
```

L'autre est de définir un type par paramètre, comme le montre l'exemple à droite.

Cette approche a plusieurs avantages :

- Elle est explicite
- Elle n'entraîne aucun coût à l'exécution
- Elle permet de localiser les règles d'affaires d'un type à même ce type. Ici, c'est **`Largeur`** qui définira les règles de validité d'une largeur de **`Rectangle`**, et qui garantira le respect de ces règles
- Elle permet d'offrir plusieurs signatures pour une même fonction, si cela semble opportun. Ici, le code client peut instancier un **`Rectangle`** avec une **`Largeur`** puis une **`Hauteur`**, ou encore avec une **`Hauteur`** puis une **`Largeur`**, au choix

Son inconvénient principal est qu'il peut être fastidieux de définir plusieurs types : si cette approche est utilisée de manière abusive, le nombre de types dans un programme risque d'exploser; de même, les types peuvent avoir des particularités contextuelles (peut-être la largeur d'un **`Rectangle`** et celle d'un **`Cercle`** sont-elles soumises à des règles distinctes), ce qui implique de réfléchir au design des classes (recours à des espaces nommés, à des classes internes, à des dépôts de classes auxiliaires, à des fabriques, etc.)

Dans les classes **`Hauteur`** et **`Largeur`**, il aurait aussi été possible de remplacer l'accesseur **`valeur()`** par un opérateur de conversion au type du substrat, par exemple :

```
class Hauteur {
   int valeur;
public:
   constexpr explicit Hauteur(int valeur) noexcept : valeur{ valeur }{
   }
   constexpr explicit operator int() const noexcept {
      return valeur;
   }
};
```

C'est une question de préférence, en fait (appeler **`static_cast<int>`** sur une **`Hauteur`** ou appeler sa méthode **`valeur()`**).

```
class Largeur {
   int valeur_;
public:
   constexpr explicit Largeur(int valeur)
      : valeur_{ valeur } // ... valider?
   {
   }
   constexpr int valeur() const {
      return valeur_;
   }
};
class Hauteur {
   int valeur_;
public:
   constexpr explicit Hauteur(int valeur)
      : valeur_{ valeur } // ... valider?
   {
   }
   constexpr int valeur() const {
      return valeur_;
   }
};
class Rectangle {
   // ...
public:
   // ...
   constexpr Rectangle(Largeur, Hauteur);
   constexpr Rectangle(Hauteur, Largeur);
   // ...
};
// ...
int main() {
   Rectangle rect{ Largeur{ 3 }, Hauteur{ 7 } };
   // ...
}
```

Quelques textes d'autres sources :

- Dans ce texte de 2014, Marco Arena formule le souhait que cet idiome soit supporté à même le langage, plutôt que par une technique de programmation : http://marcoarena.wordpress.com/2014/12/16/bring-named-parameters-in-modern-cpp/

pImpl

L'idiome pImpl (pour *Private Implementation*) est une pratique par laquelle il est possible, en C++, de concevoir des objets dont l'implémentation est véritablement opaque. Le compilateur et la classe travaillent de concert pour isoler de manière stricte l'implémentation de l'interface. Par conséquent, le code client est découplé de l'implémentation de l'objet, et devient strictement portable (au sens de la compilation, à tout le moins).

Quelques textes de votre humble serviteur :

- Un exemple d'implémentation de délégué à l'aide de cet idiome : ../../Sources/Exemple-delegue.html
- Quelques exemples de mise en application de cet idiome, pour l'un de mes cours : ../../Sources/pImpl-exemples.html
- Un exemple d'application de cet idiome pour concevoir des mutex portables : ../Client-Serveur/Mutex-Portables.html
- Un exemple d'application de cet idiome pour concevoir des *threads* portables : ../Client-Serveur/Threads-Portables.html

Quelques textes d'autres sources :

- https://en.cppreference.com/w/cpp/language/pimpl
- Discussions de cet idiome par Herb Sutter :
  - http://www.gotw.ca/publications/mill05.htm
  - http://herbsutter.com/gotw/_100/
  - http://herbsutter.com/gotw/_101/
- Présentation de pImpl et des déclarations *a priori* dans une optique de réduction du couplage, par Anton Khodakivskiy en 2012 : http://akhodakivskiy.github.com/2012/12/11/private-implementation-forward-declaration.html
- Texte de 2013 suggérant que toute personne programmant avec C++ devrait être familière avec cet idiome : http://tonka2013.wordpress.com/2013/08/31/why-every-c-developer-should-know-about-the-pimpl-idiom-pattern/
- L'idiome ramené à l'essentiel : http://www.cppsamples.com/common-tasks/pimpl.html
- L'idiome pImpl expliqué par Manu Sánchez en 2016 : https://manu343726.github.io/2016/03/07/c++11-opaque-pointer-idiom.html
- Implémentation de pImpl suivant les principes de la règle de zéro, par Andrey Upadyshev en 2015 : http://oliora.github.io/2015/12/29/pimpl-and-rule-of-zero.html
- Réaliser un pImpl avec un **`unique_ptr`**, par Jonathan Boccara en 2017 : https://www.fluentcpp.com/2017/09/22/make-pimpl-using-unique_ptr/

RAII

L'idiome RAII (pour *Resource Acquisition is Initialization*) est fortement répandu en C++, du fait que ce langage propose un modèle permettant la finalisation déterministe (grâce aux destructeurs) des objets automatiques. Le langage C#, avec les blocs **`using`**, et Java 7, avec les blocs **`try-with`** (inspirés de cette proposition de Joshua Bloch) offrent d'ailleurs maintenant des mécanismes similaires.

Prenons par exemple la fonction **`g()`** à droite, qui alloue (pour des raisons obscures) dynamiquement une instance de **`X`**, puis la passe à la fonction **`f()`** qui n'est pas **`noexcept`**.

Ici, si **`f()`** devait lever une exception, la finalisation de **`*p`** (réalisée par **`delete p`** dans ce code plus-que-douteux) ne serait pas atteinte, et le programme subirait une fuite de mémoire (problème souvent moins grave qu'il n'y paraît), mais surtout une possible fuite de ressources dû à la non-finalisation de **`*p`** (imaginez si **`X::~X()`** devait être responsable de compléter une transaction bancaire...)

Allouer dynamiquement des ressources est utile... quand il y a un besoin. Sur la base du code de droite, ce besoin n'est pas évident, et il est probable qu'ici, une allocation automatique aurait été plus adéquate :

```
void g() {
   X x;
   // ...
   f(&x);
   // ...
} // aucune fuite, plus simple, plus rapide
```

Faisons donc comme si l'allocation dynamique était pertinente ici, pour les besoins de l'illustration.

```
class X { /* ... */ };
int f(const X*);
void g() {
   auto p = new X;
   // ...
   f(p); // si f() lève une exception, *p ne sera pas détruit
   // ...
   delete p;
}
```

Si nous souhaitons automatiser la finalisation du pointé, il suffit de confier cette responsabilité à une variable locale (ici, un **`std::unique_ptr<X>`**) et de laisser son destructeur s'en charger.

```
#include <memory>
class X { /* ... */ };
int f(const X*);
void g() {
   std::unique_ptr<X> p { new X };
   // ...
   f(p.get()); // si f() lève une exception, *p ne sera pas détruit
   // ...
} // destruction implicite du pointé
```

Quelques textes de votre humble serviteur :

- Un exemple d'application de l'idiome `RAII` à l'aide de verrous testables : ../Client-Serveur/Verrous-testables.html
- Petit texte sur les objets `RAII` et la copie de contenu : ../Client-Serveur/Copie-et-synchro.html
- Quelques exemples d'utilisation d'objets `RAII` : ../Client-Serveur/Objets-autonomes.html
- Une discussion comparative (et sujette à débats) de l'impact de cet idiome : ../Divers--cplusplus/CPP--Exploiter-Symetrie.html

Quelques textes d'autres sources :

- https://en.cppreference.com/w/cpp/language/raii
- Une description de cet idiome : http://www.hackcraft.net/raii/
- Une autre description de l'idiome, cette fois sous un autre nom (`AC/ DC`, pour *Acquire in Constructor, Destructor Cleanup*) : http://blog.brush.co.nz/2009/02/raii-acdc/
- Les blocs **`using`** de C# :
  - http://msdn.microsoft.com/en-us/library/yh598w02.aspx
  - texte d'Eric Lippert en `2014` : http://blog.coverity.com/2014/02/26/resources-vs-exceptions/
- Les blocs **`try-with`** de Java `7` :
  - http://download.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html
  - http://jdevelopment.nl/trywithresources-jdk7-scoped-declarations/
- Une réflexion sur le modèle de types de Java, en lien avec cet idiome : http://www.artima.com/lejava/articles/javaone2009_gwyn_fisher.html
- Une variante reposant sur une macro : http://blogs.msdn.com/b/twistylittlepassagesallalike/archive/2011/07/21/the-final-macro.aspx
- Une réflexion sur le nom de l'idiome, par Andrzej Krzemieński en `2012` : http://akrzemi1.wordpress.com/2012/03/09/resource-or-session/
- En `2012`, Martin C. Martin se lance dans une défense de cet idiome : http://blog.martincmartin.com/2012/07/06/in-defense-of-implicit-code-in-c/
- Dans cette présentation de `2012`, Jon Kalb recommande qu'on utilise plutôt le vocable *Responsibility Acquisition Is Initialization* : http://exceptionsafecode.com/slides/esc.pdf
- Il est possible, en C++, de spécialiser une méthode sur la base du traitement de **`*this`** en tant que référence ou en tant que référence sur un *rvalue*. Cette possibilité peut améliorer certaines manoeuvres associées à l'idiome `RAII`. À cet effet, article de `2014` : http://kukuruku.co/hub/cpp/ref-qualified-member-functions
- En `2015`, Pavel Frolov propose une technique `RAII` pour couvrir des cas qui seraient déplaisants à traiter avec un pointeur intelligent : http://accu.org/index.php/journals/2086
- Appliquer `RAII` avec D, texte de `2015` : https://w0rp.com/blog/post/an-raii-constructor-by-another-name-is-just-as-sweet/
- Même dans les outils de tests, l'approche `RAII` est saine, comme le rappelle Andrzej Krzemieński en `2015` : https://akrzemi1.wordpress.com/2015/06/25/set-up-and-tear-down/
- Comme le rappelle avec justesse Eli Bendersky en `2016`, `RAII` n'a pas besoin d'exceptions pour être un idiome pertinent : http://eli.thegreenplace.net/2016/c-raii-without-exceptions/
- Texte de Malte Skarupke en `2016` expliquant en quoi `RAII` peut simplifier la composition : https://probablydance.com/2016/06/03/c11-completed-raii-making-composition-easier/ (prudence : les idées sont intéressantes mais certains exemples de code sont incorrects)

Critiques :

- Je dois avouer avoir été surpris, mais cette présentation de Jonathan Blow en `2014` contient une critique de cet idiome. Le lien qui suit mène non pas au début de la présentation mais bien à la section qui émet la critique en question : https://www.youtube.com/watch?v=TH9VCN6UkyQ&feature=youtu.be&t=29m56s
- En `2018`, Jonathan Boccara identifie des cas où `RAII` n'est pas, selon lui, la meilleure approche : https://www.fluentcpp.com/2018/02/13/to-raii-or-not-to-raii/

Mauvaises pratiques

Il existe aussi de mauvaises pratiques, que ce soit des erreurs commises de bonne foi et qui se perpétuent pour quelque raison que ce soit (souvent parce que les gens n'ont pas le temps de se poser la question à savoir « est-ce une bonne idée? »), qu'on nomme souvent des *Anti-patterns* (un texte qu'on doit apparemment à Andrew Koenig, du moins selon Martin Fowler dans http://martinfowler.com/bliki/AntiPattern.html), ou des gestes carrément hostiles, commis par des gens malveillants sur une base délibérée, qu'on nomme alors des *Dark Patterns*.

Quelques liens pertinents sur ces sujets :

- Un Wiki servant de catalogue aux *Dark Patterns* : http://wiki.darkpatterns.org/Home
- Un Wiki servant de catalogue aux *Anti-Patterns* : http://en.wikipedia.org/wiki/Anti-pattern
- Un autre catalogue d'*Anti-Patterns* : http://c2.com/cgi/wiki?AntiPatternsCatalog
- Des « pratiques » malsaines qui existent souvent dans du logiciel écrit avant la vague structurante ayant découlé de la publication du volume du *Gang of Four* : http://fuzz-box.blogspot.com/2011/05/resign-patterns.html
- Un exemple « amusant » d'*anti-pattern*, nommé « action à distance » : http://en.wikipedia.org/wiki/Action_at_a_distance_pattern
- Un exemple humoristique, le *Front-Ahead Design*, où *tout* est placé dans l'interface personne/ machine : http://thedailywtf.com/Articles/FrontAhead-Design.aspx
- En `2012`, Mike Hadlow décrit le fait d'utiliser une base de données en tant que file d'attente comme étant un *anti-pattern* : http://mikehadlow.blogspot.se/2012/04/database-as-queue-anti-pattern.html
- Des *Dark Patterns* pour les interfaces personne/ machine, texte de `2013` par Harry Brignull : http://www.theverge.com/2013/8/29/4640308/dark-patterns-inside-the-interfaces-designed-to-trick-you
- En `2011`, Raymond Chen décrit ce qu'il nomme le *for-if antipattern*, manière très inefficace de réaliser une tâche qui devrait être simple : http://blogs.msdn.com/b/oldnewthing/archive/2011/12/27/10251210.aspx
- Un *antipattern* bien connu est le *Crunch Mode*, ou livraison de dernière minute qui demande beaucoup de temps supplémentaire. Texte de Chad Fowler en `2014` expliquant comment se débarrasser de cette vilaine pratique : http://chadfowler.com/blog/2014/01/22/the-crunch-mode-antipattern/
- Liste de mauvaises pratiques avec Python, colligées par Constantine Lignos en `2014` : http://lignos.org/py_antipatterns/
- En `2014`, Edaqa Mortoray disserte sur ce qu'il nomme *False Abstraction* : http://mortoray.com/2014/08/01/the-false-abstraction-antipattern/
- Texte de Mike Hadlow en `2014` sur l'*antipattern* qu'il nomme *Lava Layer* : http://mikehadlow.blogspot.co.uk/2014/12/the-lava-layer-anti-pattern.html
- Ce que ce texte de `2015` par Matthew Jones nomme *Inner Platform*, où ce qui se produit quand un logiciel, en abusant de flexibilité, finit par devenir une pâle émulation d'un autre : http://www.exceptionnotfound.net/the-inner-platform-effect-anti-pattern-primers/
- En `2015`, Sahand Saba rapporte neuf mauvaises pratiques qui, selon lui, devraient être connues des programmeuses et des programmeurs : http://sahandsaba.com/nine-anti-patterns-every-programmer-should-be-aware-of-with-examples.html
- Selon ce texte de `2015`, l'*anti-pattern* le plus coûteux serait... le recours malvenu à **`printf()`**, particulièrement dans une page Web : http://m1el.github.io/printf-antipattern/
- Selon Piotr Solnica en `2015`, et je suis plutôt d'accord avec lui, le recours à des objets placés dans un état invalide est un *antipattern* : http://solnic.eu/2015/12/28/invalid-object-is-an-anti-pattern.html
- Le **`for...case`**, expliqué en `2017` par Remy Porter : http://thedailywtf.com/articles/a-foursome-of-arrays (et pour une suite : http://thedailywtf.com/articles/your-private-foursome)
- Une mauvaise pratique, au sens de pratique foncièrement malhonnête, dans le cadre d'interfaces personne / machine, rapportée par Raymond Chen en `2018` : https://blogs.msdn.microsoft.com/oldnewthing/20180730-00/?p=99365
- Ensemble de mauvaises pratiques en C++, colligées par Jonathan Wakely : http://kayari.org/cxx/antipatterns.html
- En `2017`, Michael Nygard prétend que les *Entity Services* sont une mauvaise pratique et ajoutent une complexité inutile aux systèmes sans que le retour sur l'investissement ne soit suffisant : http://www.michaelnygard.com/blog/2017/12/the-entity-service-antipattern/
- Ces mauvaises pratiques qui deviennent des pratiques, tout simplement, texte de Jasper Sprengers en `2017` : https://blog.codecentric.de/en/2017/09/anti-patterns-become-pattern/
- Mauvaises pratiques de programmation en binômes, par Siddarth en `2017` : https://sidstechcafe.com/pair-programming-antipatterns-xperience-792fe0112aa1
- Mauvaises pratiques de tests logiciels, identifiées et expliquées par Kostis Kapelonis en `2018` : http://blog.codepipes.com/testing/software-testing-antipatterns.html
- Les alternatives (les **`if`**) présentés comme une mauvaise pratique, du moins dans bien des cas, selon Joe Wright en `2016` : https://code.jocjag.com/2016/anti-if-the-missing-patterns.html

Adaptateur

Aussi appelé *Adapter* (anglais), cette pratique correspond à insérer le code nécessaire entre deux interfaces qui ne se correspondent pas en tout point mais pour lesquelles les différences sont suffisamment mineures pour qu'il soit possible d'adapter l'une à l'autre :

- Parfois, on parle du même concept exprimé différemment (par exemple, une classe C++ qui exposerait des itérateurs avec des méthodes **`First()`** et **`PastEnd()`** alors que la métaphore standard de ce langage demande **`begin()`** et **`end()`**)
- Parfois, on parle d'ajouter des éléments manquants mais triviaux à déduire à partir d'une interface existante (ceci peut se faire à l'aide de traits par exemple)
- Enfin, il faut parfois construire une mécanique complète à partir d'une autre mécanique, comme dans le cas des **`reverse_iterator`** qui peuvent être élaborés sur la base d'itérateurs bidirectionnels existants

Textes d'autres sources :

- Explications de Zaheer Ahmed en `2016` : http://conceptf1.blogspot.ca/2016/02/adapter-design-pattern.html

Bâtisseur

Aussi appelé *Builder*, ce schéma décrit un objet qui connaît une séquence d'opérations lui permettant d'assembler des objets. Petit cousin de Fabrique.

Textes d'autres sources :

- Explication proposée par Nikolas Fränkel en `2013` : http://blog.frankel.ch/a-dive-into-the-builder-pattern
- Exemple idiomatique de C++, proposé par Joseph Mansfield : http://www.cppsamples.com/patterns/builder.html

Bytecode

Ce schéma survient naturellement lorsque le besoin de flexibilité d'un système est tel qu'il vaut mieux encoder les actions sous forme de données destinées à une machine virtuelle.

À ce sujet :

- Explication de la pratique par Robert Nystrom : http://gameprogrammingpatterns.com/bytecode.html

Commande

Ce schéma correspond à représenter des actions posées dans un programme sous la forme d'entités logicielles, par exemple pour être en mesure de préparer une séquence d'instructions à exécuter en bloc (des macros) ou pour mettre en place un mécanisme d'annulation ou de réexécution de commandes (*Undo*/ *Redo*).

Un exemple complet implémentant un outil simpliste d'édition de texte avec annulation et réexécution suit.

Les inclusions standards sont en petit nombre. J'utiliserai :

- L'en-tête **<memory>** pour **unique_ptr**
- Les en-têtes **<string>** et **<string_view>**, puisque nous manipulerons du texte
- L'en-tête **<cassert>** pour valider quelques cas critiques, et
- Je préférerai **<vector>** à **<stack>** du fait que je ne serai pas toujours puriste dans mon utilisation des piles permettant d'annuler et de refaire

En particulier, je permettrai d'afficher le contenu d'une pile, pour faciliter le débogage, ce qui est plus simple à réaliser avec un conteneur généraliste comme **std::vector** qu'avec un « conteneur » dont l'interface est limitée comme **std::stack**.

```
#include <memory>
#include <string>
#include <string_view>
#include <iostream>
#include <cassert>
#include <algorithm>
#include <vector>
using namespace std;
```

Toutes les commandes seront des dérivés de l'interface **CommandeImpl** montrée à droite. Une commande devra savoir s'exécuter, s'annuler et (pour fins de débogage) se décrire sur un flux. Notez qu'en pratique, une commande s'exécutera ou s'an nulera sur l'état du programme; dans le cas simple décrit ici, l'état du programme n'est que la chaîne de caractères en cours d'édition.

```
struct CommandeImpl {
   virtual void executer(string&) = 0;
   virtual void annuler(string&) = 0;
   virtual ostream& decrire(ostream&) const = 0;
   virtual ~CommandeImpl() = default;
};
```

Le code client ne sera pas invité à utiliser des **CommandeImpl*** ou des pointeurs intelligents de **CommandeImpl**. Il ne s'agirait pas d'une interface naturelle d'utilisation. En C++, il est d'usage de proposer au code client des objets simples à manipuler et qui se comportent un peu comme des int (suivant une maxime proposée par Scott Meyers : *Do as the* **int***s do!*).

La classe **Commande** ici a les caractéristiques suivantes :

- Elle est incopiable car son attribut **p** est lui-même incopiable
- Elle est déplaçable, implémentant la sémantique de mouvement, ce qui permet de l'entreposer dans un conteneur
- Elle englobe une sorte de **CommandeImpl** qui ne peut être nulle
- Ses services concrets (**executer()**, **annuler()** et **decrire()**) sont des relais vers le **CommandeImpl** qu'elle englobe

Remarquez que le nom **Commande** (le nom « évident ») a été réservé à la classe qui devrait être utilisée. C'est un facteur parmi tant d'autres pour inviter le code client à se restreindre aux meilleures pratiques de programmation.

Remarquez aussi que je n'en ai pas fait un pImpl strict, du fait que le souhait ici est que **CommandeImpl** soit implémentée par autant que classes spécialisées que jugé nécessaire par le code client.

```
class Commande {
   unique_ptr<CommandeImpl> p;
public:
   Commande(unique_ptr<CommandeImpl> p) : p{ std::move(p) } {
      assert(p && "Une implementation non-nulle de commande est requise");
   }
   Commande(Commande&&) = default;
   Commande(nullptr_t) {
      assert(false && "Une implementation non-nulle de commande est requise");
   }
   Commande& operator=(Commande&&) = default;
   void executer(string &s) {
      p->executer(s);
   }
   void annuler(string &s) {
      p->annuler(s);
   }
   ostream& decrire(ostream &os) const {
      return p->decrire(os);
   }
};
```

Une **Commande** s'affiche sans peine sur un flux, mais le fruit de cette action est de réaliser une projection polymorphique (variant selon le type effectif de la **CommandeImpl**) sur le flux en question. Flexible pour la commande, simple pour le code client.

```
ostream& operator<<(ostream &os, const Commande &cmd) {
   return cmd.decrire(os);
}
```

Pour notre programme de démonstration, les seules commandes seront des entrées de texte, qui pourront être exécutées (ajoutant le texte à la chaîne de caractères qui représente l'état de l'édition en cours) ou annulées (supprimant le texte de la fin de la chaîne de caractères qui représente l'état de l'édition en cours).

Remarquez que tous ses services sont privés, outre son constructeur paramétrique.

Le texte conservé dans une **EntreeTexte** sera une ligne entrée au clavier. Le saut de ligne ne sera pas conservé dans l'attribut **texte**, pour simplifier le code de **decrire()**, mais les opérations **annuler()** et **executer()** en tiendront compte.

```
class EntreeTexte : public CommandeImpl {
   string texte;
   void executer(string &dest) {
      dest += texte;
      dest += '\n';
   }
   void annuler(string &dest) {
      assert(dest.size() >= texte.size() + 1); // +1 pour le '\n'
      dest = dest.substr(0, dest.size() - (texte.size() + 1));
   }
   ostream& decrire(ostream &os) const {
      return os << "Entree du texte \"" << texte << '\"';
   }
public:
   EntreeTexte(string_view s) : texte{ s } {
   }
};
```

Passons maintenant au code client de ces quelques classes.

Pour fins décoratives, deux fonctions banales englobent l'exécution du programme de test :

- **presenter()**, qui affiche un petit mot de bienvenue, et
- **au_revoir()**, qui salue l'usager et affiche le fruit de la séance d'édition

```
void presenter() {
   cout << "Petit editeur magique\n"  << endl;
}
void au_revoir(string_view s) {
   cout << "Le resultat de votre seance d'edition va comme suit:\n" << s
        << "\n\nA la prochaine!"  << endl;
}
```

La séance d'édition sera interactive et permettra à l'usager de réaliser des actions. La gamme des actions possibles est décrite par l'énumération forte **Action**.

```
enum class Action : short {
   MONTRER, AJOUTER, ANNULER, REEXECUTER, VISUALISER, QUITTER
};
```

Pour réduire le recours à des constantes directement dans le code, et pour garantir un peu de souplesse si la gamme des actions s'enrichit, un prédicat **est_quitter()** permet de savoir si une **Action** donnée implique de quitter le programme.

```
bool est_quitter(Action action) noexcept {
   return action == Action::QUITTER;
}
```

La fonction interactive **choisir_action()** offre une gamme d'options à l'usager et retourne un choix valide pour une **Action**.

Pour alléger l'écriture, j'ai traité les entrées incorrectes (qui ne constituent pas un entier valide) comme des erreurs graves. Cela dit, il serait possible d'enrichir le traitement d'erreurs pour que ce cas soit considéré comme moins critique.

```
class ActionInvalide {};
Action choisir_action() {
   static const char * NOMS[] {
      "Montrer les options",
      "Ajouter texte",
      "Annuler plus recent",
      "Reexecuter plus recent",
      "Visualiser edition courante",
      "Quitter"
   };
   enum { N = sizeof(NOMS) / sizeof(NOMS[0]) };
   short choix;
   do {
      for (int i = 0; i < N; ++i)
         cout << "Entrez " << i << " pour l'option \"" << NOMS[i] << "\"\n";
      cout << "\nVotre choix? ";
      if (!(cin >> choix))
         throw ActionInvalide{};
   } while (choix < 0 && N <= choix);
   return static_cast<Action>(choix);
}
```

Le véritable client de la classe **Commande** dans ce programme est la classe **Executer**, un foncteur jouant le rôle d'un automate. C'est lui qui tient à jour les commandes qu'il est possible d'annuler ou de réexécuter, qui applique l'annulation ou la réexécution, qui réalise les actions que l'usager souhaite faire, etc.

Les éléments clés de cette classe sont :

- L'attribut **canevas**, qui est la chaîne que le programme éditera (l'état du programme)
- La méthode **presenter_piles()**, qui présente le contenu des piles **undo** (commandes qu'il est possible d'annuler) et **redo** (commandes annulées mais qu'il est possible de réexécuter) à partir de la tête (la prochaine **Commande** à annuler ou à réexécuter, selon le cas). Cette méthode, bien que non-essentielle, est utile pour déboguer et comprendre la mécanique du programme
- La méthode **annuler()**, qui annule la plus récente **Commande** exécutée et la déplace dans la pile des

```
class AnnulationVide {};
class ErreurLecture {};
class ReexecutionVide {};
class Executer {
   string canevas;
   vector<Commande> undo, redo;
   void presenter_piles() {
      if (undo.empty())
         cout << "\nAucune commande a annuler\n";
      else {
         cout << "\nCommandes a annuler:\n";
         for (auto i = undo.rbegin(); i != undo.rend(); ++i)
            cout << '\t' << *i << '\n';
      }
```

commandes qu'il est possible de réexécuter
- La méthode **`reexecuter()`**, qui réexécute la plus récente **`Commande`** annulée et la déplace dans la pile des commandes qu'il est possible d'annuler
- La méthode **`entrer_ligne()`**, qui lit une ligne à la console (en évitant un problème avec **`getline()`**), ajoute cette action dans la pile des commandes qu'il est possible d'annuler, et vide la pile des commandes qu'il est possible de réexécuter
- La méthode **`visualiser()`**, qui ne fait qe projeter à la sortie standard le texte en cours d'édition, et
- La méthode **`resultat()`**, qui retourne le fruit de l'édition

La méthode clé dans la gestion des actions est **`operator()`**, qui permet d'utiliser un **`Executer`** comme une fonction unaire. Cette méthode dirige l'exécution de l'automate en sollicitant les services appropriés selon les circonstances.

Notez que j'ai fait le choix de passer canevas en paramètre aux méthodes **`annuler()`**, **`reexecuter()`** et **`visualiser()`**. Ce choix est politique : je ne suis pas convaincu qu'**`Executer`** devrait être responsable de gérer **`canevas`**, et il se pourrait que je modifie le tout éventuellement pour faire en sorte que cet état clé du programme soit plutôt passé à **`Executer::operator()`** par son client.

En limitant le couplage entre les méthodes d'instance et **`canevas`**, je me garde un peu de latitude.

```
      if (redo.empty())
         cout << "\nAucune commande a reexecuter\n";
      else {
         cout << "\nCommandes a reexecuter:\n";
         for (auto i = redo.rbegin(); i != redo.rend(); ++i)
            cout << '\t' << *i << '\n';
      }
      cout << endl;
   }
   void annuler(string &s) {
      if (undo.empty()) throw AnnulationVide{};
      auto cmd = move(undo.back());
      undo.pop_back();
      cmd.annuler(s);
      redo.emplace_back(move(cmd));
   }
   void reexecuter(string &s) {
      if (redo.empty()) throw ReexecutionVide{};
      auto cmd = std::move(redo.back());
      redo.pop_back();
      cmd.executer(s);
      undo.emplace_back(std::move(cmd));
   }
   void entrer_ligne() {
      string ligne;
      cout << "\nEntrez du texte puis <enter>: ";
      cin.ignore();
      if (!getline(cin, ligne))
         throw ErreurLecture{};
      undo.emplace_back(make_unique<EntreeTexte>(ligne));
      redo.clear();
      canevas += ligne;
      canevas += '\n';
      cout << '\n';
   }
   void visualiser(string_view s) {
      cout << "\nEtat de l'edition en cours:\n\n" << s << '\n';
   }
public:
   void operator()(Action action) {
      try {
         switch (action) {
         case Action::MONTRER:
            presenter_piles();
            break;
         case Action::ANNULER:
            annuler(canevas);
            break;
         case Action::AJOUTER:
            entrer_ligne();
            break;
         case Action::REEXECUTER:
            reexecuter(canevas);
            break;
         case Action::VISUALISER:
            visualiser(canevas);
            break;
         default:
            assert("Ne devrait jamais arriver ici" && false);
         }
      } catch (AnnulationVide&) {
         cerr << "\nRien a annuler\n" << endl;
      } catch (ReexecutionVide&) {
         cerr << "\nRien a reexecuter\n" << endl;
      }
   }
   const string& resultat() const {
      return canevas;
   }
};
```

Enfin, le programme de test est tout simple, et s'exprime par la séquence suivante :

- Souhaiter la bienvenue à l'usager
- Permettre à l'usager de choisir une action
- Tant que l'action n'est pas de quitter, exécuter cette action et permettre d'en choisir une autre
- À la toute fin, afficher le fruit de l'édition

```
int main() {
   Executer executer;
   presenter();
   try {
      for (auto action = choisir_action(); !est_quitter(action); action = choisir_action())
         executer(action);
   } catch (ActionInvalide&) {
      cerr << "Action invalide choisie; fin du programme" << endl;
   } catch (ErreurLecture&) {
      cerr << "Erreur de lecture; fin du programme" << endl;
   }
   au_revoir(executer.resultat());
}
```

La beauté du schéma de conception Commande est qu'il permet d'abstraire sous forme d'objets polymorphiques les actions d'un programme pour les exécuter (ou les annuler) aux moments jugés opportuns.

Textes d'autres sources :

- Un Wiki sur le sujet : http://en.wikipedia.org/wiki/Command_pattern
- Présentations relativement classiques, incluant quelques schémas UML, surtout axées sur les pratiques typiques de langages tels que Java ou C# :
  - http://www.oodesign.com/command-pattern.html
  - http://sourcemaking.com/design_patterns/command
  - http://www.blackwasp.co.uk/Command.aspx
  - http://java.dzone.com/articles/design-patterns-command
- Implémentation plus près du monde du jeu : http://gameprogrammingpatterns.com/command.html

Command Query Responsibility Segregation (CQRS)

Ce schéma tient au fait que les modèles utilisés pour lire et pour modifier de l'information peuvent être distincts. Ceci correspond bien au principe de faible couplage, forte cohésion des designs orientés objets.

À ce sujet :

- Explication de Martin Fowler en 2011 : http://martinfowler.com/bliki/CQRS.html
- Discussion par Stephen Haunts en 2013 : http://stephenhaunts.com/2013/07/04/command-query-responsibility-segregation/

Décorateur

Ce schéma sert à enrichir ou à modifier (« décorer », d'où le nom du schéma de conception) une implémentation existante d'une interface.

À ce sujet :

- Un Wiki sur le sujet : http://en.wikipedia.org/wiki/Decorator_pattern
- Décorateurs avec Python :
  - un texte de 2014 : http://www.codingismycraft.com/2014/05/23/python-decorators-basics/
- Utilisations courantes du schéma de conception Décorateur avec Ruby :
  - par Jake Douglas en 2011 : http://jakedouglas.wordpress.com/2011/10/16/every-day-use-of-the-decorator-pattern/
  - par Kresimir Bojcic en 2011 : http://kresimirbojcic.com/2011/12/01/decorators-in-ruby.html
  - par Arlandis Lawrence en 2015 : http://blog.8thlight.com/arlandis-lawrence/2015/05/18/decorators-in-ruby.html
- Exemples du schéma de conception Décorateur avec Java : http://javapapers.com/design-patterns/decorator-pattern/
- Le Décorateur sans douleur, un texte d'Alex Schepanovski en 2013 : http://hackflow.com/blog/2013/11/03/painless-decorators/
- Exemple idiomatique de C++, proposé par Joseph Mansfield : http://www.cppsamples.com/patterns/decorator.html
- Explications de Zaheer Ahmed en 2016 : http://conceptf1.blogspot.ca/2016/01/decorator-design-pattern.html

Délégation

Par cette pratique, un objet affiche qu'il est en mesure de réaliser certaines opérations mais, à l'interne, les délègue à un autre objet. Dans certains cas, comme celui de l'agrégation telle qu'elle se présente sous COM, la délégation peut s'inscrire dans une pratique idiomatique d'optimisation. De nombreuses variantes existent, comme par exemple la programmation par politiques.

À propos de ce schéma de conception :

- Deux Wikis abordant le sujet selon des angles connexes :
  - http://en.wikipedia.org/wiki/Delegation_pattern
  - http://en.wikipedia.org/wiki/Delegation_%28programming%29
- Texte de `2012` par Jim Gay expliquant qu'à son avis, bien des gens confondent passage de messages et délégation : http://www.saturnflyer.com/blog/jim/2012/07/06/the-gang-of-four-is-wrong-and-you-dont-understand-delegation

Dirty Flag

Cette pratique se résume à ne pas prendre action tant qu'une action n'est pas requise.

À ce sujet :

- Explication de Robert Nystrom : http://gameprogrammingpatterns.com/dirty-flag.html

Enchaînement de méthodes

Cette pratique consiste à faire en sorte que les méthodes d'un objet puissent être littéralement « enchaînées », l'une à la suite de l'autre, dans une structure plus complexe. L'optique sous-jacente est de créer des `API` plus fluides.

Prenons par exemple la classe **`Rectangle`** très simple à droite. Outre ses services de base (p. ex. : des accesseurs **`const`** nommés respectivement **`largeur()`** et **`hauteur()`**), on y trouve deux services, aussi nommés **`largeur()`** et **`hauteur()`** dans cet exemple (aucune obligation de leur donner ces noms, évidemment) mais prenant cette fois un paramètre et retournant chaque fois une référence sur l'instance propriétaire de la méthode (sur **`*this`**). C'est par ces services (des mutateurs) que s'articule ici l'enchaînement de méthodes.

Si nous examinons le programme de test, la construction du **`Rectangle`** nommé **`r0`** demande de connaître le sens associé à chacun des paramètres qui lui sont passés. Plus concrètement, dans cet exemple, est-ce que la valeur `3` représente une hauteur ou une largeur? Évidemment, les deux choix sont légitimes, et le programmeur en charge de rédiger le code client doit consulter la documentation de la classe pour l'instancier convenablement.

L'instanciation de **`r1`**, par contre, est beaucoup plus explicite. Elle repose sur :

- La construction d'un **`Rectangle`** par défaut
- Sur ce **`Rectangle`** par défaut anonyme, on appelle la méthode **`hauteur(3)`** ce qui en modifie la hauteur et retourne une référence sur le **`Rectangle`** suivant la modification
- Sur ce **`Rectangle`** encore, on appelle la méthode **`largeur(5)`** ce qui en modifie la largeur et retourne une référence sur le **`Rectangle`** suivant la modification
- Enfin, **`r1`** est initialisé par copie du **`Rectangle`** anonyme ainsi construit, de manière explicite et étapiste

Cet exemple montre comment profiter de l'enchaînement de méthodes à la construction, mais il est évidemment possible d'appliquer cette pratique à d'autres cas d'espèces.

```cpp
#include <iostream>
class Rectangle {
public:
   class LargeurInvalide{};
   class HauteurInvalide{};
private:
   int largeur_ = 1, hauteur_ = 1;
   static bool est_hauteur_valide(int valeur) {
      return valeur > 0;
   }
   static bool est_largeur_valide(int valeur) {
      return valeur > 0;
   }
   static int valider_hauteur(int valeur) {
      if (!est_hauteur_valide(valeur))
         throw HauteurIncorrecte{};
      return valeur;
   }
   static int valider_largeur(int valeur) {
      if (!est_largeur_valide(valeur))
         throw LargeurIncorrecte{};
      return valeur;
   }
public:
   int largeur() const {
      return largeur_;
   }
   int hauteur() const {
      return hauteur_;
   }
   Rectangle& hauteur(int valeur) {
      hauteur_ = valider_hauteur(valeur);
      return *this;
   }
   Rectangle& largeur(int valeur) {
      largeur_ = valider_largeur(valeur);
      return *this;
   }
   // etc.
   Rectangle(int largeur, int hauteur)
      : largeur_{ valider_largeur(largeur) },
        hauteur_{ valider_hauteur(hauteur) }
   {
   }
   Rectangle() = default;
   // ... etc.
};
int main() {
   //
   // ici, r est-il haut de 3 et large de 5 ou l'inverse?
   // pas clair à partir de la signature...
   //
   Rectangle r0{ 3, 5 };
   //
   // ici, c'est limpide
   //
   auto r1 = Rectangle{}.hauteur(3).largeur(5);

}
```

Quelques textes d'autres sources :

- Comment Martin Fowler présente cette pratique, qu'il qualifie de *FluentInterface*, en `2005` : http://martinfowler.com/bliki/FluentInterface.html
- Comment implémenter cette pratique et en tirer profit en JavaScript, en particulier avec jQuery, par Eric Feminella en `2013` : http://flippinawesome.org/2013/05/20/fluent-apis-and-method-chaining/

Fabrique (*Factory*)

L'idée derrière le schéma de conception Fabrique est de donner à une entité la responsabilité d'en créer une autre. Ce faisant, il est par exemple possible de coupler la construction d'un objet avec des opérations la précédant ou lui succédant, comme dans le cas d'un objet Autonome qui doit représenter un thread mais, le comportement qu'il représente étant polymorphique, ne peut être démarré avant d'avoir été pleinement construit.

Une autre application du schéma de conception Fabrique et de l'initialisation en deux temps est de réduire la quantité de code dans les constructeurs, préservant ainsi le rôle d'initialisation des états qui est traditionnellement dévolu à ces fonctions bien spéciales et facilitant l'entretien du code par la suite; il est plus facile de spécialiser une classe dont les constructeurs vont à l'essentiel. Voir https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rnr-two-phase-init pour des détails.

Jointes aux interfaces, les fabriques sont extrêmement utiles dans une optique de mise à jour dynamique du code. Il est en effet possible de cacher complètement les types réellement instanciés à l'intérieur du code de la fabrique, ce qui permet de modifier le code d'instanciation sans recompiler le code client.

Quelques textes de votre humble serviteur :

- Une application du schéma de conception fabrique pour mettre en place des mutex portables : ../Client-Serveur/Mutex-Portables.html
- Une application du schéma de conception fabrique pour mettre en place des sections critiques portables : ../Client-Serveur/Sections-Critiques-Portables.html
- Au bas de ce texte sur les singletons, une variante décrit une vision d'instanciation par voie de fabrique : ../Divers--cplusplus/CPP--Singletons.html
- Ce texte sur la sérialisation propose au passage quelques applications de ce schéma de conception : ../Divers--cplusplus/Serialiser.html

Quelques textes d'autres sources :

- Tout comme il y a des classes fabriques, il y a des méthodes fabriques. À leur sujet, un Wiki : http://en.wikipedia.org/wiki/Factory_method_pattern
- Un texte de Guy Peleg décrivant une mécanique pour enregistrer des types d'objets dans une fabrique générique en C++ : http://www.artima.com/cppsource/subscription_problem.html
- Une critique des abus de ce schéma de conception, par Mark Seeman en `2011`. Notez que l'auteur utilise des langages `.NET` et a les avantages et les inconvénients de ce modèle. En C++, j'aurai réglé son problème en le découplant à l'aide de traits, et le polymorphisme n'aurait pas été nécessaire. Voyez-vous comment? http://blog.ploeh.dk/2011/12/19/FactoryOverload.aspx
- En `2014`, Kenneth Parker explique pourquoi il utilise des fabriques : http://startingdotneprogramming.blogspot.ca/2014/04/factory-methods-are-better.html
- Implémenter une fabrique avec Go, par Matthew Brown en `2016` : http://matthewbrown.io/2016/01/23/factory-pattern-in-golang/
- En `2015` et en `2016`, Bartlomiej Filipek relate ses efforts pour implémenter des fabriques :
  - http://www.bfilipek.com/2015/01/errata-and-nice-c-factory-implementation.html
  - http://www.bfilipek.com/2016/03/nice-c-factory-implementation-2.html

Façade

Ce schéma de conception a pour vocation d'offrir une interface simplifiée pour une entité plus complexe. Ceci permet entre autres de corriger des défauts de design dans une `API`, et d'unifier l'utilisation de plusieurs composants interreliés.

Quelques textes d'autres sources :

Un Wiki sur le sujet : http://en.wikipedia.org/wiki/Facade_pattern

Implémenter ce schéma de conception en C# pour offrir une interface homogène à plusieurs conteneurs, texte d'Eric Vogel en `2014` : http://visualstudiomagazine.com/articles/2013/06/18/the-facade-pattern-in-net.aspx

Flyweight

L'idée derrière le schéma de conception *Flyweight* est de partager les états (immuables, dans l'immense majorité des cas) des objets entre eux, pour faciliter la création d'une vaste quantité d'objets sans consommer une quantité excessive de mémoire.

Quelques textes d'autres sources :

- un Wiki sur le sujet : http://en.wikipedia.org/wiki/Flyweight_pattern
- une explication avec schémas : http://sourcemaking.com/design_patterns/flyweight
- description dans un contexte de jeu : http://gameprogrammingpatterns.com/flyweight.html

Injection de dépendance

Ce schéma de conception permet de définir des classes capables de réaliser des tâches à partir de la combinaison de fonctionnalités définies en partie par des tiers. Plusieurs versions de ce schéma de conception existent, certaines étant statiques alors que d'autres sont dynamiques. Une variante de cette approche est la programmation par politiques.

Un exemple polymorphique, basé sur des interfaces, serait celui visible à droite. Nous y voyons :

- Une interface **`Decodeur`**, qui dicte comment une opération du décodage d'un flux doit se faire (ici, prendre le flux et en consommer une chaîne de caractères)
- Quelques implémentations de cette interface, soit une qui procède une ligne à la fois, une qui procède un mot à la fois, et une qui s'arrête à la rencontre d'un délimiteur, celui-ci inclus... Cette liste n'épuise bien sûr pas les possibilités
- Une implémentation, **`ZeDecodeur`**, qui saura comment consommer le texte d'un flux sur la base d'un **`Decodeur`** qui lui sera éventuellement fourni. Notez que cette implémentation exige un **`Decodeur`** non-nul, mais qu'on aurait aussi pu implémenteur une version par défaut, sujette à être remplacée par un **`Decodeur`** fourni par le code client si cela s'avère pertinent, et
- Du code de test

```
#include <string>
#include <istream>
#include <memory>
#include <cassert>
#include <sstream>
#include <fstream>
#include <iostream>
using namespace std;
//
// Interface à implémenter
//
class FluxEpuise {};
struct Decodeur {
   virtual string consommer(istream &) = 0;
   virtual ~Decodeur() = default;
};
//
// Quelques exemples d'implémentation
//
struct DecodeurLigneParLigne : Decodeur {
   string consommer(istream &is) {
      string ligne;
      if (!getline(is, ligne)) throw FluxEpuise{};
      return ligne;
   }
};
struct DecodeurMotParMot : Decodeur {
   string consommer(istream &is) {
      string mot;
      if (!(is >> mot)) throw FluxEpuise{};
      return mot;
   }
};
class DecodeurViaDelimiteur : public Decodeur {
   char delim;
public:
   DecodeurViaDelimiteur(char delim) : delim{delim} {
   }
   string consommer(istream &is) {
      if (!is) throw FluxEpuise{};
      is >> std::noskipws;
      string s;
      char c;
      for(; is.get(c) && c != delim; s.push_back(c))
         ;
      if (is) s.push_back(c);
      is >> std::skipws;
      return s;
   }
};
//
/ La classe dans laquelle se fera l'injection
//
class ZeDecodeur {
   unique_ptr<Decodeur> decodeur_;
public:
   ZeDecodeur(unique_ptr<Decodeur> &&decodeur)
      : decodeur{ std::move(decodeur) }
   {
      assert(decodeur);
   }
   string consommer(istream &is) {
      return decodeur->consommer(is);
   }
};
//
// Exemple d'utilisation
//
void decoder(ZeDecodeur zd, istream &is, ostream &os) {
   using std::endl;
   try {
      for(;;)
         os << zd.consommer(is) << endl;
   } catch (...) {
   }
}
int main() {
   decoder(
      ZeDecodeur{
         make_unique<DecodeurLigneParLigne>()
      },
      ifstream{ "in.txt" }, cout
   );
   decoder(
      ZeDecodeur{
         make_unique<DecodeurMotParMot>()
      },
      ifstream{"in.txt"}, cout
   );
   decoder(
      ZeDecodeur{
         make_unique<DecodeurViaDelimiteur>('}')
      },
      ifstream{ "in.txt" }, cout
   );
}
```

Un exemple générique, moins fortement couplé, serait celui visible à droite.

La mécanique est essentiellement la même, à quelques nuances près :

- Nous évitons l'allocation dynamique de mémoire
- Nous évitons aussi l'imposition d'un parent commun unique tel que l'interface **`Decodeur`** de la version polymorphique
- Nous n'avons pas la possibilité de changer d'implémentation dynamiquement si le coeur nous en dit, le type de décodeur étant inscrit dans le type de **`ZeDecodeur`**

```
#include <string>
#include <istream>
#include <sstream>
#include <fstream>
#include <iostream>
using namespace std;
//
// Quelques exemples d'implémentation
//
class FluxEpuise {};
struct DecodeurLigneParLigne {
   string consommer(istream &is) {
      string ligne;
      if (!getline(is, ligne)) throw FluxEpuise{};
      return ligne;
   }
};
struct DecodeurMotParMot {
   string consommer(istream &is)    {
      string mot;
      if (!(is >> mot)) throw FluxEpuise{};
      return mot;
   }
};
class DecodeurViaDelimiteur {
   char delim;
public:
   DecodeurViaDelimiteur(char delim) : delim{delim} {
```

```
   }
   string consommer(istream &is) {
      if (!is) throw FluxEpuise();
      is >> std::noskipws;
      char c;
      string s;
      for(; is.get(c) && c != delim_; s.push_back(c))
         ;
      if (is) s.push_back(c);
      is >> std::skipws;
      return s;
   }
};
//
// La classe dans laquelle se fera l'injection
//
template <class D>
   class ZeDecodeur {
      D decodeur;
   public:
      ZeDecodeur(D decodeur) : decodeur{decodeur} {
      }
      string consommer(istream &is) {
         return decodeur.consommer(is);
      }
   };
template <class D>
   ZeDecodeur<D> CreerDecodeur(D &&decodeur) {
      return ZeDecodeur<D>{std::forward<D>(decodeur)};
   }
//
// Exemple d'utilisation
//
template <class D>
   void decoder(ZeDecodeur<D> zd, istream &is, ostream &os) {
      try {
         for(;;)
            os << zd.consommer(is) << endl;
      } catch (...) {
      }
   }
int main() {
   decoder(
      CreerDecodeur(DecodeurLigneParLigne{}),
      ifstream{ "in.txt" }, cout
   );
   decoder(
      CreerDecodeur(DecodeurMotParMot{}),
      ifstream{ "in.txt" }, cout
   );
   decoder(
      CreerDecodeur(DecodeurViaDelimiteur{'}'}),
      ifstream{ "in.txt" }, cout
   );
}
```

Quelques textes d'autres sources :

- Article de Griffin Caprio, en 2005, montrant des exemples d'application de ce schéma de conception pour la plateforme .NET : http://msdn.microsoft.com/fr-fr/magazine/cc163739%28en-us%29.aspx
- Texte de Martin Fowler sur le sujet : http://www.martinfowler.com/articles/injection.html
- Variante connue, l'inversion de contrôle, décrite par Martin Fowler en 2005 : http://martinfowler.com/bliki/InversionOfControl.html
- Texte de Dhananjay Nene, en 2005, montrant des exemples d'application de ce schéma de conception pour la plateforme Java : http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection
- Tutoriel pour introduire à cette pratique, par Rick Hightower en 2011 : http://code.google.com/p/jee6-cdi/wiki/DependencyInjectionAnIntroductoryTutorial_Part1
- Un texte de 2012 à propos du passage de code où se produit une injection de legs à code où se produit une injection de dépendances : http://blogs.telerik.com/blogs/posts/12-04-04/from-legacy-to-dependency-injection.aspx
- Comparer l'injection de dépendance et la programmation orientée aspect :
  - pourquoi préférer la POA? Texte de Gael Fraiteur en 2010 : http://www.postsharp.net/blog/post/Aspect-Oriented-Programming-vs-Dependency-Injection.aspx
  - dans ce texte de 2013, Kenneth Truyers recommande de préférer l'injection de dépendance à la POA : http://www.kenneth-truyers.net/2013/05/16/why-choose-di-interception-over-aspect-oriented-programming/
- Refactoriser le code pour favoriser l'injection de dépendances, un texte d'Ondrej Balas en 2014 :
  - http://visualstudiomagazine.com/articles/2014/05/01/how-to-refactor-for-dependency-injection.aspx
  - http://visualstudiomagazine.com/articles/2014/06/01/how-to-refactor-for-dependency-injection.aspx
- Injection de dépendances en JavaScript en passant par des propriétés plutôt que par des constructeurs, un texte de Stefan Isele en 2014 : http://stefan-isele.logdown.com/posts/200710-javascript-dependency-injection-into-properties
- Injection de dépendances avec Go, texte de 2014 : http://blog.parse.com/2014/05/13/dependency-injection-with-go/
- En 2015, Derek Comartin recommande de jeter à la poubelle nos conteneurs d'injection de dépendances : http://codeopinion.com/throw-out-your-dependency-injection-container/
- Injection de dépendances avec Java, par Hany Ahmed en 2015 : https://dzone.com/articles/javaee-contexts-and-dependency-injection

Critiques de cette pratique :

- En 2011, Tony Marston explique que, selon lui, l'injection de dépendances est quelque chose de mal : http://www.tonymarston.net/php-mysql/dependency-injection-is-evil.html
- En 2013, Jason Lewis se dit d'avis qu'il s'agit d'une bonne idée, mais appliquée au mauvais paradigme de programmation : http://decomplecting.org/blog/2013/01/03/dci-the-right-idea-for-the-wrong-paradigm/
- Sans critiques l'injection de dépendances en tant que telle, ce texte de 2014 par Yegor Bugayenko présente les conteneurs provoquant de l'injection de dépendances comme étant une sorte d'*antipattern* : http://www.yegor256.com/2014/10/03/di-containers-are-evil.html

Interface

L'idée derrière le schéma de conception Interface est de déterminer, souvent par une abstraction polymorphique, une strate de services opaque qui seront implémentés par d'autres entités. Ceci permet d'exprimer les algorithmes sur une base plus abstraite et plus générale, et tend à mener vers du code plus réutilisable.

Ce schéma de conception est si répandu que plusieurs langages (souvent orientés objets), et pas les moins connus, en ont fait un concept de niveau langage plus qu'une pratique. Pour cette raison, nombreux sont ceux qui ne le voient plus comme un schéma de conception. Et pourtant...

À propos des interfaces en C# et de certaines subtilités propres à ce langage, voir : ../Divers--cdiese/Interfaces.html

Quelques textes d'autres sources :

- En 2012, Zed A. Shaw nous rappelle que dans une API comme de manière générale, l'abstraction et l'indirection sont deux choses différentes l'une de l'autre : http://zedshaw.com/essays/indirection_is_not_abstraction.html
- Texte de Robert C. Martin en 2015 sur le fait que le mot interface, pris au sens contraignant de C# et de Java, est une pratique malsaine (je suis plutôt d'accord avec lui, je dois l'avouer; c'est un truc qui m'a toujours beaucoup déçu de ces langages) : http://blog.cleancoder.com/uncle-bob/2015/01/08/InterfaceConsideredHarmful.html
- Réflexion de 2015 par Vladimir Khorikov portant sur la nuance entre interface en tant que concept et interface en tant qu'entité d'un langage de programmation : http://enterprisecraftsmanship.com/2015/06/02/interfaces-vs-interfaces/

Intermédiaire (*Proxy*)

L'idée derrière le schéma de conception Intermédiaire (on utilise souvent le nom anglais *Proxy*) est de placer une entité tierce entre deux entités, pour ajouter la couche d'abstraction proverbiale dont fait mention le théorème fondamental de l'informatique.

Ce schéma de conception est très répandu, en particulier dans les systèmes répartis où il facilite la mise en place d'approches comme la communication RPC par exemple.

Certains concepts ne peuvent se représenter par des intermédiaires (en C++, certains échecs bien connus comme celui de la spécialisation du type `vector<bool>`, qui ont essayé de représenter des booléens par des bits, en attestent). Cela n'empêche pas des expérimentations amusantes pour qui n'est pas trop puriste (comme ceci).

Quelques textes d'autres sources :

- Un Wiki sur le sujet : http://en.wikipedia.org/wiki/Proxy_pattern
- En Java, décorer un objet dynamiquement avec un *Proxy* : http://www.ibm.com/developerworks/java/library/j-jtp08305/index.html

Itérateur

L'idée derrière les itérateurs est d'offrir une abstraction du concept de parcours d'une séquence. Ceci permet la rédaction d'algorithmes applicables à une séquence d'éléments, et ce de manière indépendante du conteneur dans lequel les éléments sont entreposés (arbre, vecteur, liste simplement ou doublement chaînée, *etc.*).

Certaines bibliothèques, dont STL pour C++, exploitent beaucoup cette pratique, ce qui permet d'approcher l'orthogonalité entre algorithmes et conteneurs, accroissant du même coup l'éventail d'outils disponibles pour fins de développement logiciel.

Quelques textes de votre humble serviteur :

- Une introduction aux conteneurs et aux itérateurs de C++

- Un petit truc pour définir des itérateurs de points de vue en C++
- Comment réaliser une itération statique en C++

Quelques textes d'autres sources :

- Le Wiki sur le sujet : http://en.wikipedia.org/wiki/Iterator
- Réflexions sur les concepts d'itérateur interne et d'itérateur externe, par Robert Nystrom en 2013 : http://journal.stuffwithstuff.com/2013/01/13/iteration-inside-and-out/
- Les itérateurs avec Go, par Ewen Cheslack-Postava en 2013 : http://ewencp.org/blog/golang-iterators/

Critiques du schéma de conception Itérateur

- Un texte d'Andrei Alexandrescu qui prétend que les itérateurs de C++ sont brisés, et qu'on devrait les remplacer par des intervalles : http://www.uop.edu.jo/download/PdfCourses/Cplus/iterators-must-go.pdf

Modèle/ Vue/ Contrôleur

L'approche Modèle/ Vue/ Contrôleur (MVC pour les intimes) est un schéma de conception servant au développement d'applications, le mot « application » étant pris au sens de systèmes informatiques offrant une interface personne/ machine, ce qui sied bien au développement d'interfaces Web placées par-dessus un ou plusieurs générateurs de contenu (ASP, JSP, Servlet, etc.).

L'idée derrière MVC est de formaliser la séparation entre une interface personne/ machine et ses mécanismes sous-jacents. Règle générale, le contrôleur est associé aux événements en entrée, le modèle est associé aux traitements sous-jacents, et la vue est associée à ce qui est proposé à l'humain, souvent dans une interface visible à l'écran. Ces trois éléments constitutifs sont en interaction mutuelle et forment un triangle.

Selon la vision MVC :

- Le **modèle** contient l'essentiel des données nécessaires au bon fonctionnement de l'application – ses états. Pensez à une classe pourvue principalement d'accesseurs et de manipulateurs. Le modèle ne sait rien de la vue ou du contrôleur;
- La **vue** présente l'apparence de l'application. Règle générale, la vue peut consulter les états du modèle (faire des **`get`**) mais ne peut les modifier (faire des **`set`**). La vue ne devrait pas connaître le contrôleur. La vue doit par contre être avertie lorsque le modèle est modifié, de manière à présenter une information à jour – le modèle est en général chargé de cette tâche;
- Le **contrôleur** est chargé de réagir aux entrées faites par l'usager. Il est chargé de créer et d'initialiser le modèle.

On remarquera qu'une des qualités du découpage MVC est qu'il permet d'avoir plusieurs vues sur un même modèle. Une variante, le « Modèle Document Vue », groupe la vue et le contrôleur de plus près; c'est l'approche de la bibliothèque MFC de Microsoft. Ce couplage serré a le gros défaut de rendre le code difficile à segmenter, et tend à rendre les interfaces MFC difficiles à réutiliser ou à faire évoluer dans le temps. Le Modèle Document Vue se prête surtout à des tâches pouvant être représentées par des classes terminales.

L'approche MVC a ceci d'intéressant qu'elle permet de formaliser en bonne partie un découpage et des comportements logiciels typiques. Cette formalisation entraîne une systématisation et permet de développer des infrastructures de prise en charge de bonnes parties de ces modèles. Avec Java, la technologie Struts est un bon exemple d'une infrastructure de ce genre.

Comprendre le modèle MVC permet de bien utiliser la plupart des infrastructures commerciales d'interfaces personne/ machine contemporaines.

Il existe des *Frameworks* spécifiquement dédiés au développement d'applications selon MVC, par exemple Angular.js : ../Web/JavaScript-Outils.html#angularjs

Quelques textes d'autres sources :

- Descriptifs de ce schéma de conception :
  - http://ootips.org/mvc-pattern.html
  - explication informelle de Mike Pack en 2012 : http://mikepackdev.com/blog_posts/34-a-review-of-mvc
- Un Wiki sur ce sujet : http://en.wikipedia.org/wiki/Model-view-controller
- Une explication bien faite sur MSDN : http://msdn.microsoft.com/en-us/library/ff649643.aspx
- Un texte de Thomas Davis s'attardant sur la partie modèle du schéma de conception MVC : http://backbonetutorials.com/what-is-a-model/
- Un texte d'Edward Z. Yang discutant de MVC dans une optique de pureté, ce mot étant pris ici au sens de pureté dans la programmation fonctionnelle : http://blog.ezyang.com/2010/07/mvc-and-purity/
- Une variante de Martin Fowler, la présentation séparée : http://www.martinfowler.com/eaaDev/SeparatedPresentation.html
- En 2012, James Wrightson relate qu'à son avis, pour un petit jeu, MVC est *Overkill* mais peut tout de même s'avérer utile pour contrôler un personnage avec souris et clavier : http://boxhacker.com/blog/2012/03/11/model-view-controller/
- De l'avis de Conrad Irwin en 2012, MVC est mort et il faut passer à autre chose. Il met de l'avant ce qu'il nomme MOVE, pour *Models, Operations, Views and Events*, donc une extension à MVC tenant compte des événements : http://cirw.in/blog/time-to-move-on
- De son côté, Ingo Schramm explique, en 2012, qu'il n'est pas clair à ses yeux que MOVE soit préférable à MVC : http://ingoschramm.tumblr.com/post/26409997578/mvc-move-or-simply-a-state-machine
- Saines pratiques pour MVC avec JavaScript, selon Alex McCaw en 2014 : http://blog.sourcing.io/mvc-style-guide
- Selon l'auteur de ce texte de 2013, MVC est vraiment un schéma de conception du monde des interfaces personne/ machine : http://ajdotnet.wordpress.com/2013/11/03/mvc-is-a-ui-pattern/
- Appliquer MVC en Haskell, un texte de Gabriel Gonzalez en 2014 : http://www.haskellforall.com/2014/04/model-view-controller-haskell-style.html
- Le passé, le présent et l'avenir de MVC, d'un point de vue Java, par Gastón I. Silva en 2014 : http://givan.se/p/00000010
- Gérer des exceptions dans un design de type MVC, texte du Code Project par Marla Sukesh en 2014 : http://www.codeproject.com/Articles/731913/Exception-Handling-in-MVC
- Selon Alexander Jung en 2013, MVC est un schéma de conception d'interfaces personne/ machine et ne devrait pas être confondu avec une approche ayant des visées plus larges : https://ajdotnet.wordpress.com/2013/11/03/mvc-is-a-ui-pattern/
- Le schéma MVC avec Go, par Jon Calhoun en 2015 : http://calhoun.io/creating-controllers-views-in-go/
- Mieux comprendre MVC, par Andreas Söderlund en 2015 : https://github.com/ciscoheat/mithril-hx/wiki/Rediscovering-MVC
- En 2016, Jean-Jacques Dubray explique pourquoi il a abandonné ce schéma de conception : http://www.infoq.com/articles/no-more-mvc-frameworks
- Comprendre MVC, livre en ligne par Stefano Borini : https://stefanoborini.gitbooks.io/modelviewcontroller/content/
- En 2016, Oskar Sjöberg explique qu'il cesse d'utiliser MVC pour privilégier MVA, au sens de Modèle/ Vue/ Action : http://modulit.se/blog/no-more-mvc-for-me-i-will-use-mva/

Modèle/ Vue/ Présentation (MVP)

Variante de MVC axée sur les interfaces personne/ machine, MVP place le volet présentation entre la vue et le modèle, et délègue à la présentation la responsabilité sur la logique applicative.

À ce sujet :

- https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93presenter
- Cette explication de Microsoft met de l'avant que MVP facilite les tests et découple la logique de l'interface : https://msdn.microsoft.com/en-us/library/ff649571.aspx
- Texte de 2007 sur les différences entre MVC et MVP : http://www.infragistics.com/community/blogs/todd_snyder/archive/2007/10/17/mvc-or-mvp-pattern-whats-the-difference.aspx

Modèle/ Vue/ Vue/ Modèle (MVVM)

Le format Web tend vers une variation de MVC par laquelle la vue est encore plus découplée du modèle qu'à l'habitude. En particulier, c'est souvent le fureteur qui doit consulter le modèle pour vérifier si des changements y ont été apportés. Cette approche, MVVM, est populaire chez les gens qui utilisent beaucoup les outils Microsoft ou des Frameworks Web tels qu'Angular.js.

Avec MVVM, le contrôleur est remplacé par un simple lien entre la vue et le modèle, et ce lien se limite souvent à une transformation des données.

À ce sujet :

- http://en.wikipedia.org/wiki/Model_View_ViewModel
- Explication du modèle par Josh Smith en 2009 : http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
- La variante MVVM-C, expliquée par Susann Proszak en 2016 : http://tech.trivago.com/2016/08/26/mvvm-c-a-simple-way-to-navigate/

Null Object

L'idée derrière le *Null Object* est d'avoir une implémentation par défaut (ne faisant rien) d'un comportement polymorphique donné, dans le but d'éviter les cas particuliers ou dégénérés dans le code, et ainsi de réduire les tests et les risques d'erreurs. Un *Null Object* peut souvent prendre la place d'une référence nulle ou d'un pointeur nul dans un programme; puisque le *Null Object* est un objet valide, nul besoin de vérifier s'il est là ou non – s'il est là, alors son comportement est essentiellement de ne rien faire.

Un exemple (simple) est présenté à droite, à la fois sans *Null Object* et avec *Null Object*. Dans les deux cas, l'exemple présente un combat fort inégal à trois mettant en scène les monstres Bob, Joe et Bill. Bob est armé d'un bâton, Joe est armé d'une scie à chaîne alorsque Bill est désarmé. C'est le cas d'un monstre désarmé qui nous intéresse ici.

Le premier cas n'applique pas le schéma de conception *Null Object*. On y représente donc les armes possédées par un monstre donné à l'aide de pointeurs, pour fins d'indirection polymorphique, et le fait d'être désarmé est représenté par un pointeur nul vers une arme.

Puisque le pointeur peut être nul, il doit être testé à chaque fois qu'on souhaite utiliser ce vers quoi il pointe. Cela signifie que chaque appel à **`Monstre::frapper()`** pour un **`Monstre`** donné implique un test (un **`if`**) et, si le test réussit, un appel polymorphique à la méthode **`frapper()`** de l'arme vers laquelle mène le pointeur.

Si les monstres désarmés sont rares, alors la majorité des tests seront superflus, gaspillant des ressources; le fait qu'un monstre désarmé soit possible impose cependant la présence de ce test (l'oublier pourrait faire planter le programme).

```
#include <string>
#include <random>
#include <algorithm>
#include <iostream>
using namespace std;
//
// Quelques globales pour alléger l'exemple...
// (ne faites pas ça à la maison!)
//
random_engine rd;
mt19937 prng{ rd() };
uniform_int_distribution<int> distrib{ 10,50 };
//
//
//
class Monstre;
struct Arme {
   virtual void frapper(Monstre &) = 0;
   virtual ~Monstre() = default;
};
class Monstre {
   int vie;
   Arme *arme;
```

```
   string nom_;
public:
   Monstre(const string &nom, Arme *arme)
      : nom_{ nom }, vie{ 100 }, arme{ arme }
   {
   }
   string nom() const {
      return nom_;
   }
   void bobo(int degats) {
      vie -= degats;
   }
   bool mort() const noexcept {
      return vie <= 0;
   }
   void frapper(Monstre &m) {
      if(arme)
         arme->frapper(m);
   }
};
struct Baton : Arme {
   void frapper(Monstre &m) {
      m.bobo(distrib(prng));
   }
};
struct Chainsaw : Arme {
   void frapper(Monstre &m) {
      m.bobo(distrib(prng));
   }
};
bool est_vivant(const Monstre &m) {
   return !m.mort();
}
int main() {
   Chainsaw chainsaw;
   Baton baton;
   Monstre monstres[] {
      Monstre("Bob", &chainsaw),
      Monstre("Joe", &baton),
      Monstre("Bill", nullptr) // désarmé!
   };
   enum { N = std::size(monstres) };
   uniform_int_distribution<int> monstre_distrib{ 0, N-1 };
   while(count(begin(monstres), end(monstres), est_vivant) > 1) {
      // un monstre peut s'auto-mutiler ou s'acharner
      // sur un cadavre (ils sont bêtes)
      if (auto &m = monstres[monstre_distrib(prng)]; !m.mort())
         m.frapper(monstres[monstre_distrib(prng)];
   }
   cout << "Le gagnant: "
        << find_if(begin(monstres), end(monstres), est_vivant)->nom()
        << endl;
}
```

En appliquant le schéma de conception *Null Object*, le test devient redondant, comme le montre l'exemple à droite.

Notez que, pour mieux illustrer le principe, ce nouvel exemple utilise des références plutôt que des pointeurs à titre d'indirection polymorphique. En C++, outre quelques perversions techniques, une référence ne peut être nulle.

Le code est plus simple, en général plus rapide (à moins que le cas particulier que représente le *Null Object* ne soit en fait un cas fréquent, car dans ce cas, les tests avec **`if`** *pourraient* coûter moins cher que des appels polymorphiques répétés (et encore, si nous considérons les risques accrus de sécurité qu'entraîne la nécessité de tester le pointeur chaque fois, il n'est pas clair que ce soit un réel gain).

Enfin, nous avons un gain documentaire : le concept d'être désarmé est représenté par un type, dûment nommé, et ne requiert plus vraiment qu'on l'accompagne de commentaires.

```
#include <string>
#include <random>
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;
//
// Quelques globales pour alléger l'exemple...
// (ne faites pas ça à la maison!)
//
random_engine rd;
mt19937 prng{ rd() };
uniform_int_distribution<int> distrib{ 10,50 };
//
//
//
class Monstre;
struct Arme {
   virtual void frapper(Monstre &) = 0;
   virtual ~Monstre() = default;
};
class Monstre {
   int vie;
   Arme &arme;
   string nom_;
public:
   Monstre(const string &nom, Arme &arme)
      : nom_{ nom }, vie{ 100 }, arme{ arme }
   {
   }
   string nom() const {
      return nom_;
   }
   void bobo(int degats) {
      vie -= degats;
   }
   bool mort() const noexcept {
      return vie <= 0;
   }
   void frapper(Monstre &m) {
      arme.frapper(m);
   }
};
struct Baton : Arme {
   void frapper(Monstre &m) {
      m.bobo(distrib(prng));
   }
};
struct Chainsaw : Arme {
   void frapper(Monstre &m) {
      m.bobo(distrib(prng));
   }
};
struct Desarme : Arme {
   void frapper(Monstre &) {
   }
};
bool est_vivant(const Monstre &m) {
   return !m.mort();
}
int main() {
   Chainsaw chainsaw;
   Baton baton;
   Desarme desarme;
   Monstre monstres[] {
      Monstre{ "Bob", chainsaw },
      Monstre{ "Joe", baton },
      Monstre{ "Bill", desarme }
   };
   enum { N = std::size(monstres) };
   uniform_int_distribution<int> monstre_distrib{0, N-1};
   while(count(begin(monstres), end(monstres), est_vivant) > 1) {
      // un monstre peut s'auto-mutiler ou s'acharner
      // sur un cadavre (ils sont bêtes)
      if (auto &m = monstres[monstre_distrib(prng)]; !m.mort())
         m.frapper(monstres[monstre_distrib(prng)];
   }
   cout << "Le gagnant: "
        << find_if(begin(monstres), end(monstres), est_vivant)->nom()
        << endl;
}
```

Quelques textes d'autres sources :

- Texte de `2002` par Kevlin Henney, présentant ce schéma de conception : http://www.researchgate.net/publication/242408118_Null_Object_Something_for_Nothing
- Un texte de Martin Fowler décrivant ce schéma de conception, sous le nom assez bien choisi de *Special Case* (ou « cas particulier ») : http://martinfowler.com/eaaCatalog/specialCase.html
- Un texte en format `PDF` de Kevlin Henney décrivant aussi ce schéma de conception : http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf
- Le recours au « code confiant », une approche en partie controversée qui repose partiellement sur ce schéma de conception : http://avdi.org/talks/confident-code-rubymidwest-2011/

- Utiliser des *Null Objects* pour faciliter la composition de fonctionnalités tout en évitant de s'enfouir sous les alternatives (les **`if`**), texte de 2014 : http://www.italiancpp.org/2014/11/23/anti-if-idioms-in-cpp/

Observateur

L'idée derrière le schéma de conception Observateur est de formaliser l'idée d'un service d'abonnement, par exemple pour réagir à des événements dans une interface personne/ machine ou lors de l'arrivée de données sur un flux.

> Le code à droite est un petit exemple écrit en réponse à une question de Zinedine Bedrani, cohorte 07 du DDJV et qui utilise quelques trucs chouettes de C++ 11 comme **`auto`**, les λ et les **`shared_ptr`**.

Quelques textes de votre humble serviteur :

- Un exemple d'implémentation simpliste du schéma de conception Observateur avec C# : ../Divers--cdiese/Observateur.html
- Un exemple d'implémentation du schéma de conception Observateur pour réagir à des fuites de mémoire, en C++ : ../../Sources/Detection-Fuites--Observateur.html

Quelques textes d'autres sources :

- Un Wiki sur ce sujet : http://en.wikipedia.org/wiki/Observer_pattern
- Un exemple relativement détaillé d'observateur, par IBM : http://researchweb.watson.ibm.com/designpatterns/example.htm
- Implémenter correctement l'interface **`INotifyPropertyChanged`** de la plateforme .NET, application concrète de ce schéma de conception, par Karol Waledzik en 2012 : http://www.sharpcrafters.com/blog/post/The-Definitive-INotifyPropertyChanged.aspx
- Réduire le couplage du code dans les interfaces personne/ machine à l'aide de ce schéma de conception, texte de Craig Gidney en 2012 : http://twistedoakstudios.com/blog/Post1694_decoupling-inlined-ui-code
- Texte de Herb Sutter en 2003 portant sur la généralisation de cette pratique : http://www.drdobbs.com/cpp/generalizing-observer/184403873?queryText=sutter%2B%2522generalizing%2Bobserver%2522
- Explication proposée par Robert Nystrom : http://gameprogrammingpatterns.com/observer.html
- Exemple idiomatique de C++, proposé par Joseph Mansfield : http://www.cppsamples.com/patterns/observer.html
- Alléger l'écriture pour implémenter ce schéma de conception, proposition de Brett Hall en 2015 : https://backwardsincompatibilities.wordpress.com/2015/04/21/to-much-code-signalled/

```cpp
#include <locale>
#include <vector>
#include <memory>
#include <algorithm>
#include <iostream>
using namespace std;
struct ILecteurTouches {
   virtual void reagir(char) = 0;
   virtual ~ILecteurTouches() = default;
};
class DejaAbonne {};
class PasAbonne {};
class serveur_touches {
   vector<shared_ptr<ILecteurTouches>> abonnes;
public:
   void abonner(shared_ptr<ILecteurTouches> p) {
      if (!p) return;
      if (find(begin(abonnes), end(abonnes), p) != end(abonnes))
         throw DejaAbonne{};
      abonnes.push_back(p);
   }
   void desabonner(shared_ptr<ILecteurTouches> p) {
      if (!p) return;
      auto it = find(begin(abonnes), end(abonnes), p);
      if (it == end(abonnes))
         throw PasAbonne{};
      abonnes.erase(it);
   }
   void agir() {
      if (char c; cin >> c)
         for(auto & p : abonnes_)
           p->reagir(c); // <-- observateur!
   }
};
//
// supposons une classe qui gère la logique du jeu
// (simplifiée à l'ultra-extrême, bien entendu)
//
class Jeu {
   bool fini = false;
public:
   void quitter() {
      fini = true; }
   }
   bool fin() const {
      return fini;
   }
};
class afficheur_touche : public ILecteurTouches {
   void reagir(char c) {
      cout << c;
   }
};
class evenement_quitter : public ILecteurTouches {
   Jeu &jeu;
   locale &loc;
public:
   evenement_quitter(Jeu &jeu, const locale &loc = locale{""})
      : jeu{ jeu }, loc{ loc }
   {
   }
   void reagir(char c) {
      if (toupper(c, loc) == 'Q') // bof
         jeu.quitter(); // par exemple
   }
};
int main() {
   Jeu jeu;
   serveur_touches svr;
   svr.abonner(make_shared<evenement_quitter>(jeu)));
   svr.abonner(make_shared<afficheur_touche>()));
   while (!jeu.fin())
      svr.agir();
}
```

Ordonnanceur

L'idée derrière ce schéma de conception est de formaliser un mécanisme décrivant l'ordre dans lequel des tâches seront réalisées, et de mettre en place les requis pour assurer leur synchronisation.

Quelques textes d'autres sources :

- un Wiki sur le sujet : http://en.wikipedia.org/wiki/Scheduler_pattern

Regroupement (*Pooling*)

L'idée derrière ce schéma de conception est de réduire le coût de la création dynamique de ressources (souvent des ressources lourdes comme des *threads*, des outils de synchronisation ou des connexions à des bases de données) en créant ces ressources *a priori* puis en les distribuant au besoin à ceux qui en ont besoin.

Quelques textes de votre humble serviteur :

- Un exemple simple de regroupement de *threads* reposant sur les mécanismes de C++ 11 : ../Parallelisme/thread_pool.html
- Un exemple complet de regroupement de *threads* avec programmation par promesses (*Futures*), le tout « maison » (donc sans recours à des bibliothèques tierces ou aux mécanismes de C++ 11) : ../../Sources/Thread-Pool--Futures.html

Quelques textes d'autres sources :

- Les groupements de *threads* :
  - http://en.wikipedia.org/wiki/Thread_pool_pattern
  - http://java.sun.com/docs/books/tutorial/essential/concurrency/pools.html
- Au sujet des regroupements de connexions à des bases de données (article du Code Project) : http://www.codeproject.com/KB/cpp/StaticConnectionPool.aspx
- Les regroupements de *threads* de Microsoft Windows, par Kenny Kerr en 2011 :
  - http://msdn.microsoft.com/en-ca/magazine/hh335066.aspx
  - http://msdn.microsoft.com/en-us/magazine/hh394144.aspx
- Quoi faire si vous avez besoin d'un regroupement de *threads* et s'il s'avère que le mécanisme offert par votre système d'exploitation ne vous convient pas? Un texte de Larry Osterman, en 2004 : http://blogs.msdn.com/b/larryosterman/archive/2004/03/29/101329.aspx
- Description d'un regroupement de connexions, par Vladimir Mihalcea en 2014 : http://vladmihalcea.com/2014/04/17/the-anatomy-of-connection-pooling/
- Groupement de *threads* avec Java, texte de 2014 : http://www.sourcetricks.com/2014/04/thread-pools-in-java.html
- Groupement de *threads* et *Thread-Safety* avec C#, par Jon Skeet en 2014 : http://codeblog.jonskeet.uk/2014/08/01/object-pooling-and-thread-safety/
- Bien gérer la taille d'un regroupement de connexions, texte de Brett Woolridge en 2014 : https://github.com/brettwooldridge/HikariCP/wiki/About-Pool-Sizing
- Regroupement d'objets avec Unity, texte de 2016 par Emilia Szymańska : http://emi.gd/blog/object-pooling/
- Regroupement d'objets et *templates* variadiques, texte de Rebecca Fernandez en 2016 : https://rebeccafernandezdotcom.wordpress.com/2016/11/05/object-pools-variadic-templates-and-reference-forwarding/

Singleton

L'idée derrière les singletons est d'avoir une classe telle qu'elle ne puisse être instanciée qu'une seule fois par programme, pas plus, tout en évitant que cette contrainte ne dépende de la gentillesse et de la discipline des programmeuses et des programmeurs.

Quelques textes de votre humble serviteur :

- Les singletons en C++
- Les singletons en C#
- Les singletons en JavaScript
- Le problème de la gestion de l'ordonnancement de la construction et de la destruction des singletons statiques en C++

Quelques textes d'autres sources :

- Le Wiki sur le sujet : http://en.wikipedia.org/wiki/Singleton_pattern
- Un texte de Peter Norlund, qui propose ce qu'il décrit comme une classe de base générique en C++ pour des singletons qui soit telle que l'ordre de destruction des singletons soit contrôlé (son approche est différente de la mienne) : http://www.nada.kth.se/cvap/abstracts/cvap246.html
- Un texte qui postule que le schéma de conception singleton pose le problème de la mauvaise manière, soit celui de l'identité, alors que selon son auteur, l'idée clé est la localisation des états. L'auteur présente son alternative, le *Borg* (le code proposé est en Python) : http://code.activestate.com/recipes/66531/
- Un texte sur les méthodes singleton en Smalltalk et en Ruby : http://talklikeaduck.denhaven2.com/2009/05/30/singleton-methods-in-smalltalk-and-ruby
- Un vieux (1996, donc précédant le standard ISO de C++, qui n'est même pas la dernière version du standard) texte de Douglas C. Schmidt sur le *Double-Checked Locking*, une pratique (prudence! Voir ci-dessous...) qui vise à éviter certaines conditions de course associées aux singletons : http://www1.cse.wustl.edu/~schmidt/editorial-3.html
- Un texte plus récent (2004) des bien connus Scott Meyers et Andrei Alexandrescu portant sur les périls du *Double-Checked Locking* : http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- Google fournit un outil de détection de singletons et d'autres états globaux : http://code.google.com/p/google-singleton-detector/
- Application de ce schéma de conception dans Chrome, pour optimiser certains comportements du fureteur : http://www.igvita.com/posa/high-performance-networking-in-google-chrome/#predictor
- Des singletons en D, un texte de 2013 : http://davesdprogramming.wordpress.com/2013/05/06/low-lock-singletons/
- Des singletons en Perl, un texte de 2013 par David Farrell : http://perltricks.com/article/52/2013/12/11/Implementing-the-singleton-pattern-in-Perl
- Une alternative aux singletons et aux variables globales en C++, proposée par Bob Schmidt en 2015 : http://accu.org/index.php/journals/2085
- En 2015, Arne Mertz s'interroge à savoir si le schéma de conception Singleton est une bonne ou une mauvaise pratique : http://arne-mertz.de/2015/04/singletons-whats-the-deal/
- Dans ce texte de 2015, Robert C. Martin prêche pour une application non-dogmatique du singleton, surtout en lien avec les tests : http://blog.cleancoder.com/uncle-bob/2015/07/01/TheLittleSingleton.html
- Plusieurs variantes d'implémentation d'une singleton en C++, proposées par Joe Ruether en 2015 : http://jrruethe.github.io/blog/2015/08/02/singletons/
- Explications de Zaheer Ahmed en 2015 : http://conceptf1.blogspot.ca/2015/04/singleton-design-pattern.html
- Texte de 2016 par Paul M. Watt, qui explique avoir eu une épiphanie au contact de ce schéma de conception : http://codeofthedamned.com/index.php/the-singleton
- Initialisation concurrente d'un singleton en C++ et rapidité d'exécution, texte de 2016 par Rainer Grimm : http://www.modernescpp.com/index.php/thread-safe-initialization-of-a-singleton
- Deux manières simples d'implémenter un singleton en C++, par Alon Gonen en 2017 : http://cppisland.com/?p=501
- Revisiter le singleton en C++, texte de 2017 par Giuseppe (nom de famille inconnu) : http://www.italiancpp.org/2017/03/19/singleton-revisited-eng/
- Texte de Marc Gregoire en 2017, qui propose d'implémenter un singleton en C++ à l'aide de variables **static** magiques : http://www.nuonsoft.com/blog/2017/08/10/implementing-a-thread-safe-singleton-with-c11-using-magic-statics/
- Discussion de manières d'implémenter un singleton en C#, par Jon Skeet : http://csharpindepth.com/Articles/General/Singleton.aspx (merci à Alexandre Leblanc pour le lien)
- Des singletons en Java à l'aide d'énumérations, par Dulaj Atapattu en 2017 : https://dzone.com/articles/java-singletons-using-enum
- En 2017, Robert Shenk propose une approche en Java qu'il nomme les « singletons malléables » : https://dzone.com/articles/the-malleable-singleton-pattern-bend-dont-break
- Singletons et systèmes embarqués, texte de 2014 par Jason Sachs : https://www.embeddedrelated.com/showarticle/691.php

Critiques du schéma de conception Singleton

- Une critique de ce schéma de conception et de son application dans le domaine du jeu vidéo : http://gameprogrammingpatterns.com/singleton.html
- Un texte qui prétend que les singletons sont le mal : http://blogs.msdn.com/b/scottdensmore/archive/2004/05/25/140827.aspx
- En 2016, Umer Mansoor recommande d'éviter le schéma de conception Singleton pour en avoir du code plus facile à tester : http://codeahoy.com/2016/05/27/avoid-singletons-to-write-testable-code/
- Critique proposée par Mihai Sebea en 2018 : https://www.fluentcpp.com/2018/03/06/issues-singletons-signals/
- Selon Umer Mansoor en 2016, les singletons compliquent les tests : https://codeahoy.com/2016/05/27/avoid-singletons-to-write-testable-code/

State

Le schéma de conception *State* tient à la représentation des états d'un automate par des objets, et à la navigation d'un objet à l'autre en fonction des circonstances en tant que flux d'exécution du programme résultant. En général, on utilise ce schéma de conception pour éviter de recourir à une masse d'alternatives ou à de très longues sélectives.

Un exemple viendra quand j'aurai quelques minutes.

Quelques textes d'autres sources :

- http://en.wikipedia.org/wiki/State_pattern
- Remplacer les alternatives par une implémentation du schéma de conception *State*, par Emiliano Mancuso en 2015 : http://bits.citrusbyte.com/state-design-pattern-with-ruby/

Stratégie

Le schéma de conception Stratégie tient à offrir plusieurs implémentations pour une même interface, tout en laissant le code client faire le choix de l'implémentation en fonction du contexte. Il ressemble en ceci aux idiomes NVI et pImpl, qui vont tous deux plus en détail dans les modalités. Ce schéma de conception permet entre autres à un même client de se construire à partir de plusieurs stratégies comportementales distinctes.

Quelques textes d'autres sources :

- Un Wiki sur le sujet : hhttp://en.wikipedia.org/wiki/Strategy_pattern
- En 2015, texte de Veronica S. Zotali, sur le Code Project, présentant une implémentation du schéma de conception Stratégie avec C# pour faire varier les implémentations d'une même interface de dictionnaire : http://www.codeproject.com/Tips/875620/Strategy-Design-Pattern-Csharp
- Appliquer le schéma de conception Stratégie au redimensionnement d'objets graphiques, par Bartlomiej Filipek en 2015 : http://www.bfilipek.com/2015/05/applying-strategy-pattern.html
- Explications de Yianna Kokalas en 2017 : http://www.manyiterations.com/2017/01/07/design-patterns-strategy.html

Visiteur

> Le visiteur est un drôle d'oiseau, qui est difficile d'entretien lorsqu'il est mis en application de manière classique mais dont on ne voudrait pas se passer pour la navigation de structures complexes. Certains (comme Vincent Thériault, un de mes anciens étudiants à la cohorte 02 du DDJV) font des miracles avec ce schéma de conception. D'autres (comme Andrei Alexandrescu, dans son livre *Modern C++ Design*) essaient d'en atténuer la lourdeur.
>
> Les pratiques avec ce schéma de conception se raffinent encore aujourd'hui, en couplant polymorphisme et généricité. Le fin mot reste à venir...

L'idée derrière le visiteur est de permettre à un objet de naviguer la structure d'une autre objet de l'intérieur. Ceci permet entre autres de distinguer la navigation de structures complexes, par exemple des arbres et des graphes, des opérations faites lors de cette navigation (modification de certaines noeuds, affichage de leur contenu).

L'exemple donné à droite est celui d'un arbre binaire générique minimaliste. Un prédicat donné à la compilation permet à l'arbre de décider chaque fois si un élément en cours d'ajout doit être placé à gauche ou à droite d'un noeud donné.

Les méthodes **visiter()**, déclinées en version **const** et non-**const**, permettent à un foncteur de s'inviter dans une instance de cette classe pour y appliquer des opérations sur les valeurs de chaque noeud.

Le programme principal montre deux exemples de tels visiteurs, soit l'un qui affichera chaque noeud (traversée en profondeur, de gauche à droite) et l'autre qui doublera la valeur de chaque noeud.

> Le schéma de conception Visiteur couple les objets capables de visiter avec les méthodes qui permettent de les accueillir et de les faire naviguer dans la structure interne d'un objet. On pourrait les qualifier d'itérateurs intrusifs, en quelque sorte.

Suite à une séance en classe avec les chics étudiants de la cohorte 07 du DDJV, j'ai ajouté un exemple permettant d'injecter un foncteur capable d'accumuler de l'information sur les noeuds visités. Pour ce faire, j'ai fait passer les fonctions **visiter()** du type **void** au type **F**, donc au type du paramètre représentant l'opération en cours de visite. Cette sémantique est connexe à celle utilisée pour **std::for_each()** dans STL, mais demande que les opérations visiteuses puissent être copiées. Ceci explique le recours à une sémantique de mouvement dans le foncteur **aff** dans le programme principal – l'affectation n'est pas implémentée sur un flux tel qu'un **std::ostream**.

```
struct PlusPetitQue {
   template <class T>
      bool operator()(const T &a, const T &b) {
         return a < b;
      }
};

template <class T, class Pred = PlusPetitQue>
   class arbre_binaire {
   public:
      using value_type = T;
   private:
      struct Noeud {
         value_type valeur;
         Noeud *gauche {}, *droite{};
         Noeud(const value_type &valeur) : valeur{ valeur } {
         }
      };
      Noeud *racine {};
      Pred pred;
      void ajouter(const value_type &valeur, Noeud *p) {
         if (pred(valeur, p->valeur))
            if (p->gauche)
               ajouter(valeur, p->gauche);
            else
               p->gauche = new Noeud(valeur);
         else
            if (p->droite)
               ajouter(valeur, p->droite);
            else
               p->droite = new Noeud(valeur);
      }
   public:
      arbre_binaire(Pred pred = {}) : pred{pred} {
      }
      bool empty() const noexcept {
         return !racine;
      }
      void ajouter(const value_type &valeur) {
         if (empty())
            racine = new Noeud{valeur};
         else
            ajouter(valeur, racine);
      }
   private:
      void clear_from(Noeud *p) {
         if (p->gauche) {
```

```
            clear_from(p->gauche);
            delete p->gauche;
         }
         if (p->droite) {
            clear_from(p->droite);
            delete p->droite;
         }
      }
   public:
      void clear() noexcept {
         if (!racine) return;
         clear_from(racine);
         delete racine;
         racine = {};
      }
      ~arbre_binaire() {
         clear();
      }
   private:
      template <class F>
         F visiter(F fct, Noeud *p, int depth) {
            fct(p->valeur, depth);
            if (p->gauche) fct = visiter(fct, p->gauche, depth + 1);
            if (p->droite) fct = visiter(fct, p->droite, depth + 1);
            return fct;
         }
      template <class F>
         F visiter(F fct, const Noeud *p, int depth) const {
            fct(p->valeur, depth);
            if (p->gauche) fct = visiter(fct, p->gauche, depth + 1);
            if (p->droite) fct = visiter(fct, p->droite, depth + 1);
            return fct;
         }
   public:
      template <class F>
         F visiter(F fct) {
            if (!racine_) return fct;
            return visiter(fct, racine_, 0);
         }
      template <class F>
         F visiter(F fct) const {
            if (!racine_) return fct;
            return visiter(fct, racine_, 0);
         }
   };

#include <algorithm>
#include <iostream>
#include <string>
#include <random>

int main() {
   using namespace std;
   random_engine rd;
   mt19937 rng{ rd() };
   int vals[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
   shuffle(begin(vals), end(vals), rng);
   arbre_binaire<int> arbre;
   for(auto val : vals)
      arbre.ajouter(val);
   class aff {
      std::ostream &os;
   public:
      aff(std::ostream &os) : os{ os } {
      }
      void operator()(int i, int depth) {
         os << std::string(depth, ' ') << i << endl;
      }
   };
   class cumuler {
      int cumul {};
   public:
      cumuler() = default;
      void operator()(int val, int) {
         cumul += val;
      }
      int valeur() const {
         return cumul;
      }
   };
   arbre.visiter(aff{ cout });
   arbre.visiter([](int &val, int) { val *= 2; });
   arbre.visiter(aff{ cout });
   cout << "Somme: "
        << arbre.visiter(cumuler{}).valeur()
        << endl;
}
```

Quelques textes d'autres sources :

- Un Wiki sur le sujet : http://en.wikipedia.org/wiki/Visitor_pattern
- Une approche sophistiquée en C++, que son auteur Anand Shankar Krishnamoorthi nomme le visiteur coopératif : http://www.artima.com/cppsource/cooperative_visitor.html
- Comprendre Visiteur à partir du *Pattern Matching*, une proposition de Phil Freeman en 2013 : http://blog.functorial.com/posts/2013-10-02-Visitor-Pattern.html
- Visiter un graphe en profitant des mécanismes de C++ 14, par Adrien Hamelin en 2014 : https://aboutcpp.wordpress.com/2014/12/12/a-function-to-visit-nodes-of-a-graph-with-c14/
- Texte du Code Project, par Phillip Voyle en 2015, qui décrit un visiteur variadique pour évaluer des expressions arithmétiques : http://www.codeproject.com/Articles/896108/Expression-Evaluator-Example-Using-a-Variadic-Visi
- Un visiteur générique : ../Divers--cplusplus/Visiteurs-generiques.html
- Réaliser une forme de réflexivité avec un visiteur statique, texte de 2015 : https://medium.com/@mattgician/libraryless-reflection-in-c-288d7873e3a6
- Texte du Code Project en 2015 par Phillip Voyle qui décrit un évaluateur d'expressions arithmétiques construit à l'aide d'un visiteur : http://www.codeproject.com/Articles/896108/Expression-Evaluator-Example-Using-a-Variadic-Visi
- En 2016, Arne Mertz présente deux approches à l'implémentation du schéma de conception Visiteur :
  - http://arne-mertz.de/2016/04/visitor-pattern-oop/
  - http://arne-mertz.de/2016/04/visitor-pattern-part-2-enum-based-visitor/
- Textes de Vittorio Romeo en 2016, portant sur la visite d'un **`variant`** à partir de λ :
  - https://vittorioromeo.info/index/blog/variants_lambdas_part_1.html
  - https://vittorioromeo.info/index/blog/variants_lambdas_part_2.html
- Quelques réflexions sur le schéma de conception Visiteur, proposées en 2017 par Ewan (qui ne donne pas son nom de famille) : http://thejavasnowman.com/thoughts-on-the-visitor-design-pattern/
- Un cas vécu d'application du schéma de conception Visiteur en situation de TDD, relaté par Rafa Del Nero en 2017 : https://nobugsproject.com/2017/09/03/design-patterns-saga-14-real-project-situations-with-visitor/
- Revisiter (!) le schéma de conception Visiteur, par Jonathan Müller en 2017 : https://foonathan.net/blog/2017/12/21/visitors.html
- Visiter un arbre à coût réduit en Java « moderne », par Haoyi en 2018 : http://www.lihaoyi.com/post/ZeroOverheadTreeProcessingwiththeVisitorPattern.html

---

La section décrivant les **principaux schémas de conception en parallélisme** a été déplacée dans une page à part entière, à même la section sur le parallélisme et la programmation concurrente de ce site.

Vers le musée des horreurs.
