Génie Logiciel

Jean-Charles Régin

Master Informatique

Plan

- Software engineering in general
- Object Programming and code questions
- Design patterns

- All in parallel

Génie logiciel

- Wikipédia : « Le **génie logiciel** (en anglais : *software engineering*) : ensemble des méthodes, des techniques et des outils concourant à la production d'un logiciel, au-delà de la seule activité de programmation ».
- Manière dont le code source d'un logiciel est spécifié puis produit.
- Concerne le Cycle de vie des logiciels :
  - analyse du besoin
  - élaboration des spécifications
  - conceptualisation du mécanisme interne au logiciel
  - techniques de programmation
  - développement
  - phase de test
  - maintenance.

Génie logiciel

- Pas de méthodes particulières présentées
- Basé sur mon expérience

Logiciel

- Arrêté ministériel du 22 décembre 1981: un logiciel est défini comme "un ensemble de programmes, de procédés, de règles, de documentation relatifs au fonctionnement d'un ensemble de traitement de l'information".

Obligation de résultat

- L'**obligation de résultat**, dans le cadre des obligations contractuelles (nées d'une convention conclue entre deux ou plusieurs parties) est celle qui impose à son débiteur d'atteindre un résultat déterminé : le résultat posé et défini dans le contrat.
- Elle s'oppose, dans le même cadre contractuel, à l'obligation de moyens, qui oblige le débiteur à mettre en œuvre tous les moyens à sa disposition pour obtenir le résultat attendu, sans pouvoir garantir le résultat.
- https://fr.wikipedia.org/wiki/Obligation_de_résultat

Ce que les clients veulent

- Les clients achètent du temps gagné.
  - Il ne faut donc pas faire des choses trop précises, trop fines
  - Il faut leur faire gagner du temps et gagner du temps ce n'est pas passer sa vie à lire des docs ou des rapports techniques.
- Le logiciel doit donc avoir
  - un comportement par défaut irréprochable,
  - l'utilisation doit etre simple
    - on simplifie les mécanismes même s'ils doivent etre moins puissants mais plus efficaces
    - si on veut faire plus puissant et bien l'utilisateur acceptera de passer du temps voir de coder des trucs dans son coin parce que est-on sur que la puissance qu'on lui donne est vraiment ce qu'il veut ? meme pas !!!

Logiciel

- **Les bugs surviennent quand le logiciel ne correspond pas au besoin.**
- Un bug = non-respect de la spécification du système, c'est-à-dire de la définition de ses fonctionnalités, de ce que le système est censé faire.
- Un programme bogué = programme dont la mise en œuvre ne vérifie pas la spécification

- **Ne jouez pas sur les mots**

Bugs

Logiciel

- Sans bug ?
- Validation de Windows 2000
  - 600 000 bêta testeurs
  - Au lancement de sa commercialisation il restait 63 000 problèmes potentiels dans le code, dont 28 000 sont réels.
- 0,57 erreurs (var non initialisées…) par millier de lignes de code en moyenne
- MySQL : 0,09

Bugs

- Le 13 novembre 2007 au soir, **10000 sites web** hébergés par Ovh, connaissent des difficultés.
  - Ovh a subit un crash majeur sur un des 16 serveurs de stockage, qui fonctionne avec en Raid-6 avec 28 disques, permettant une tolérance de panne de 2 disques, or 3 disques sont en panne et le système Raid a été rendu in-opérationnel.
  - Les sites ont été inaccessibles (erreur 404) environ 20h.
  - Les serveurs SQL n'ont pas été touchés.
  - Les backups récent ont sauté aussi, les webmestres se sont retrouvés avec des sites datant d'avril 2007. Toutes les modifications faites depuis 7 mois par les utilisateurs sont perdues
  - Ovh a annoncé un "geste commercial" de quelques euros

Bugs

- 29/12/2005 **Cotisation retraite** - Quelque 113 personnes ont eu la surprise de recevoir pour Noël un avis d'échéance de leurs cotisations retraite pour l'année 2006 de... deux milliards d'euros
- 01/11/2005 **Gros bug à la bourse de Tokyo** : toutes les cotations bloquées toute la journée.
- 17/11/2004 «**aucune couverture réseau**» durant 15 à 48h. Chez Bouygues Telecom deux serveurs d'acheminement des appels sont tombés en panne. L'un était censé être le secours de l'autre.

Bugs identifiés

- **Convocation de centenaires à l'école :** Convocation à l'école primaire de personnes âgées de 106 ans.
  - Codage sur deux caractères de l'age
- **Mission Vénus** : passage à 5 000 000 de Km de la planète, au lieu de 5 000 Km prévus.
  - remplacement d'une virgule par un point (au format US des nombres).
- **Mariner 1** : 27 juillet 1962, la première sonde spatiale du programme Mariner fut détruite peu de temps après son envol. Coût : 80 millions de dollars.
  - un trait d'union oublié dans un programme Fortran
    (« plus coûteux trait d'union de l'histoire », Arthur C. Clarke).
- **Passage de la ligne :** Au passage de l'équateur un F16 se retrouve sur le dos.
  - changement de signe de la latitude mal pris en compte.

Bugs

- Site web : https://github.com/umutphp/famous-bugs

Logiciel partout et très gros

- **montre : 2 K instructions**
- **téléphone mobile : 150 K instructions**
- **automobile : 1 M instructions**
- **central téléphonique : 1 M instructions**
- **noyau Linux : 3,7 M instructions**
- **système de combat du porte avions Charles de Gaulle : 8 M instructions**
- **portail Yahoo : 11 M instructions**
- **W95 : 10 M instructions**
- **WNT : 16,5 M instructions**
- **W2K : 30 à 50 M instructions**
- **Office VX pour Mac : 25 M**
- **Direction générale de la comptabilité publique (Bercy) : 160 M instructions Cobol**
- **Catia : 200 M instructions**

Logiciel partout et très gros

- 90% des nouvelles fonctionnalités des automobiles sont apportées par l'électronique et l'informatique embarquées.
  "Il y a plus d'informatique dans la Volvo S80 que dans le chasseur F15" déclarait en janvier 2000 le Président d'Audi.

Logiciel : une règle

- Un logiciel n'est jamais utilisé comme le programmeur le pense
- On ne sait donc pas bien ce qui va intéresser les utilisateurs
  - Mise sur la marché rapide
  - Modifications après compréhension

Logiciel : fonctions et besoin

- Un logiciel contient
  - Plein de fonctions pour les débutants
  - Des fonctions pour les gens moyens
  - Plein de fonctions pour les experts
- Les utilisateurs sont
  - Certains sont débutants
  - La grande majorité est moyen
  - Peu sont experts
- Adéquation ?

Logiciel

- Détection de bugs automatique ?
- Problème de la Halte
  - consiste, étant donné un programme informatique quelconque (au sens machine de Turing), à dire s'il finira par s'arrêter ou non.
  - Indécidable : il n'existe pas de programme informatique qui prendrait comme entrée le code d'un programme informatique quelconque et qui grâce à la seule analyse de ce code ressortirait VRAI si le programme s'arrête et FAUX sinon.

La Halte : conséquences

- Problème du ramasse-miettes : on cherche à libérer des zones mémoires juste après leur dernière utilisation.

- Ce problème est équivalent à celui de l'arrêt. Réduction : soit P un programme dont on veut tester l'arrêt ; considérons le programme :
  créer une zone mémoire X (jamais utilisée dans P)
  exécuter P
  écrire dans X.

- Clairement, la zone mémoire X sert après sa création si et seulement si P termine. Donc, si on savait déterminer automatiquement au vu de la lecture du programme si on peut libérer X juste après son allocation, on saurait si P termine. Cela est impossible, donc il n'existe aucun algorithme de ramasse-miettes optimalement précis.

Comment minimiser les bugs ?

- En appliquant certaines règles de base
- En pensant aux problèmes par avance
- En testant

Génie Logiciel

- Définition générale et vue d'ensemble

Génie logiciel

- Gérer un logiciel c'est considérer les points suivants :
  - Compétition (que font les autres)
  - Evolution des techniques (Langages, normes (C++11,…), architecture SSE, multicores, GPGPU, CUDA…)
  - Tests
  - Le code
  - Gestion/Partage du code
  - Exemples (pas les tests)
  - Documentation (qd commence t'on ?)
  - Les dependances et les licences
  - Qualité et outils :
    - Du développement
    - Du logiciel en lui-même : facilité d'utilisation / accessibilité

Génie logiciel

- Gérer un logiciel c'est considérer les points suivants :
  - Optimisation du code (qd ? Ou ?) : profiler
  - Protection/Secrets (Obfuscation)
  - Multithreading
  - Conception de code (Perenité, Design Pattern)
  - Maintenance/Correction de bugs / Minimiser les modifications intrusives
  - Portages :
    - compilateurs/ machines/systèmes/langages (wrapper)
    - indépendance du code vis-à-vis du matériel, de l'OS
  - Evolution : ToDo list/ refactoring
  - Installer

Génie Logiciel

- Les tests

Tests

- **But :**
  - trouver un nombre maximum de comportements problématiques du logiciel
  - s'assurer que ce système réagit de la façon prévue par ses concepteurs (spécifications)
  - s’assurer que ce système est conforme aux attentes du client l'ayant commandé (besoins), respectivement.

Tests

- **Tests de boîte noire**
  Le test porte sur le fonctionnement externe du système. La façon dont le système réalise les traitements n'entre pas dans le champ du test.
- **Tests de boîte blanche**
  Le test vérifie les détails d'implémentation, c'est à dire le comportement interne du logiciel..

Tests conformité

- **Tests de conformité**
  Le test vérifie la conformité du logiciel par rapport à ses spécifications et sa conception.
- **Tests de non-conformité** : on vérifie que les cas non prévus ne perturbent pas le systèmes
  - **Tests de robustesse** : on introduit de nouvelles contraintes
  - **Tests de vulnérabilité** : on cherche à attaquer le systèmes, à le mettre en défaut

Tests unitaires/couvertures

- **Tests unitaires**
  Chaque module du logiciel est testée séparément, par rapport à ses spécifications, aux erreurs de logique.
- **Tests cas limites**
  Que se passe t'il si on atteint une borne ? Si un fichier n'est pas présent ? On essaie les cas limites
- **Tests d'erreurs**
  Comment le logiciel gère t'il les erreurs ? Peut-on avoir une erreur dans un gestion d'erreur ?
- **Tests de couverture**
  - chaque fonction est appelée par au moins un test
  - chaque code est traversé au moins une fois
  - des utilitaires existent (pure coverage). Automatique avec certains IDE

Tests de préversions

- **Tests beta**
  Réalisés par des développeurs ou des utilisateurs sélectionnés, ils vérifient que le logiciel se comporte pour l'utilisateur final comme prévu par le cahier des charges.
- **Tests alpha**
  Le logiciel n'est pas encore entièrement fonctionnel, les testeurs alpha vérifient la pré-version.

Tests d’intégration

- **Tests d'intégration**
  Les modules validés par les test unitaires sont rassemblés. Le test d'intégration vérifie que l'intégration des modules n'a pas altéré leur comportement.
- **Tests d'intégration système**
  L'application doit fonctionner dans son environnement de production, avec les autres applications présentes sur la plateforme et avec le système d'exploitation.
- **Tests de recette**
  Les utilisateurs finaux vérifient sur site que le système répond de manière parfaitement correcte.

Tests

- **Tests fonctionnels**
  L'ensemble des fonctionnalités prévues est testé : fiabilité, performance, sécurité, affichages, etc…
- **Tests de non régression**
  Après chaque modification, correction ou adaptation du logiciel, il faut vérifier que le comportement des fonctionnalités n'a pas été perturbé, même lorsqu'elle ne sont pas concernées directement par la modification.

Tests

- **Tests de performance**
  On mesure les temps de réponse, les temps de traitements. On recherche les opérations lentes et critiques
- **Tests de montée en charge**
  On cherche à quitter la simulation et à pousser le système dans ces retranchements. On augmente le nombre de données principales et on regarde le comportement (quadratique, cubique, lags …)
  Augmentation du nombre d'utilisateurs pour un serveur

Tests

- **Tests de stress**
  On combine tests de performance et montée en charge. On peut simuler des défaillances. Longs en général

Tests

- Quand les faire ?
  - Dès le début
  - Tout le temps, mais les développeurs n'aiment pas écrire des tests
- Qui les écrits ?
  - Les développeurs
  - Quelqu'un d'autre c'est bien aussi (autre point de vue = bugs détectés plus vite; « j'ai pas pensé à ça ! »)

Génie Logiciel

- Le code source

Le code source

- Il doit
  - Être lisible par les autres (clair). Vous n'êtes pas seul.
    - Noms des fonctions
    - Noms des variables
  - Respecter les normes de codage (_ devant les champs, accolades obligatoires …). Difficile mais important.
  - Être clairement structuré : être compréhensible assez facilement
  - Être concis
  - Ne pas faire de suppositions sur d'autre partie de votre code

Le code source

- Le non respect des normes des entreprises est le premier reproche fait aux stagiaires par les entreprises
- Difficile au début, mais nécessaire
- Parfois connaissance implicite (pas de document détaillant tout)
- Exemple :
  - pas de commentaires pour le code
  - pas de noms de fonctions trop long
  - pas de return au milieu des fonctions

Le code source

- Il existe quelques règles communes de codage
  - https://google.github.io/styleguide/javaguide.html
  - https://www.oracle.com/java/technologies/javase/codeconventions-programmingpractices.html
  - http://www.possibility.com/Cpp/CppCodingStandard.html#names
  - http://geosoft.no/development/cpppractice.html
  - http://www.codingstandard.com/HICPPCM/index.html
  - http://www.misra-cpp.org/

Verification code source

- Checkstyle
- Intellij : automatique à partir des settings:
  - File/Setting/Editor/CodeStyle/Java
- Les normes DO-178B pour l'avionique et MISRA C pour l'automobile imposent un ensemble d'objectifs à atteindre sur le logiciel selon la criticité qui lui est attribuée

Génie Logiciel

- Gestion / Partage du code

Gestion / Partage du code

- Plusieurs personnes doivent interagir en même temps sur le même code.
- Code commun + chacun développe un truc de son côté
  - De temps en temps modification du code commun
- Parfois on veut juste tester
- Parfois on veut revenir à une version précédente (par exemple comprendre une régression ou l'apparition d'un bug)

Gestion / Partage du code

- Un **logiciel de gestion de versions** permettant de stocker des informations pour une ou plusieurs ressources informatiques permettant de récupérer toutes les versions intermédiaires des ressources, ainsi que les différences entre les versions.

- Les plus connus : Git, Mercurial, SVN (subversion), CVS

Gestion / Partage du code

- Mutualisation du développement:
  - Un groupe de développeurs utilise le gestionnaire pour stocker toute évolution du code source.
  - Le gestionnaire
    - gère les mises à jour des sources pour chaque développeur,
    - conserve une trace de chaque changement (avec commentaire).
    - travaille par fusion de copies locale et distante, et non par écrasement de la version distante par la version locale.
  - Deux développeurs sur une même source
    - les changements du premier à soumettre son travail ne seront pas perdus lorsque le second, qui a donc travaillé sur une version non encore modifiée par le premier, renvoie ses modifications.

Gestion / Partage du code

- Mutualisation du développement:
  - Fonctionne par un système de commit/update
  - Une copie globale + chacun dispose d'une copie en locale
  - Chacun travaille avec sa copie locale
  - On veut « committer » les modifications = modifier le fichier global à partir du local
  - On commence par « updater » la copie locale :
    - Introduction des modifications du fichier global dans le fichier local
    - Prise en compte des modifs de la version globale et modification de la version locale pour etre compatible. Vérification du code (compilation, tests…). On update de nouveau jusqu'à équivalence entre les versions
  - Modification de la version globale par la version locale

Gestion / Partage du code

- Création de branches = modification sur un ensemble de fichiers
  - maintenance d'anciennes versions du logiciel (sur les branches) tout en continuant le développement des futures versions (sur le tronc) ;
  - développement parallèle de plusieurs fonctionnalités volumineuses sans bloquer le travail quotidien sur les autres fonctionnalités.

Génie Logiciel

- Les exemples

Les exemples

- Il est important de donner un certain nombres d'exemples
- La plupart des utilisateurs ne lisent pas la documentation, mais regardent les exemples
- Ils doivent être pédagogiques :
  - On doit pouvoir les comprendre sans documentation
  - On doit pouvoir s'en inspirer : codage par mimétisme est assez fréquent
- On met un peu de documentations dans les exemples.

Génie Logiciel

- La documentation

La documentation

- Elle est très importante
- Elle peut avoir valeur légale
- Avantages :
  - Elle amène à des clarifications
  - Elle donne une vision et une cohérence d'ensemble
  - Elle fait réfléchir
  - Elle donne un référence
  - Elle permet de définir clairement certains concepts
- Inconvénients :
  - Elle peut figer les structures internes
  - Elle peut limiter les évolutions
  - Elle est pénible à écrire
  - Elle met en évidence certains défauts (mauvaise structure …)

La documentation

- Il y a une grande différence entre
  - la documentation interne (du code, de son organisation, des son fonctionnement, des algorithmes, des idées …)
  - la documentation externe (ce que le client voit, les autres équipes …)

- Il faut garder cette différence. Documenter = rigidifier
- L'open source est très rarement documenté (fait exprès)

La documentation

- Même s'il y a des documentalistes ou si elle est sous-traitée les développeurs et responsables de projets sont impliqués (relecture – modifications obligatoires)
- Il existe des outils pour la générer automatiquement
  - Doxygen
  - Javadoc en java

La documentation

- 3 types de documentation courants
  - **Getting started :**
    - petit manuel expliquant comment utiliser le logiciel à partir de rien.
    - commence par un guide d'installation, puis présente rapidement le logiciel avec des exemples très simple
  - **Manuel de l'utilisateur :**
    - manuel principalement basé sur des exemples et des concepts.
    - explique comment faire quelque chose avec le logiciel
  - **Manuel de référence :**
    - manuel de type dictionnaire
    - explique de façon très précise le fonctionnement de chaque fonctionnalité

La documentation

- Documentation entre versions :
  - **Migration guide :**
    - explique à un ancien utilisateur comment passer à la nouvelle version (l'utiliser au mieux).
    - explique comment utiliser de nouvelles fonctions à la place de précédentes
  - **Release notes :**
    - contient l'ensemble des modifications depuis la dernière version
    - référencie les fonctionnalités obsolètes et les remplaçante
    - présente les nouveautés

La documentation

- Indépendance ou pas du code source ?
  - ça se discute
- Difficulté majeure : faire évoluer le produit et la documentation en même temps
- Une documentation obsolète est un vrai problème

- Ce qui est valable pour la documentation est aussi valable pour les commentaires

Génie Logiciel

- La qualité et les outils

Qualité logicielle

- Qualité logicielle
  - Qualité du développement et les outils
  - Qualité du logiciel

- Qualité
  - Aptitude à l'emploi + conformité aux spécifications

  - Automatisation des tests
  - Procédures de contrôle
  - Définition de critères et de mesures

Qualité du développement

- Outils de développement
  - Visual Studio
  - Eclipse, IntelliJ
- Outils de tests unitaires
  - cppUnit
  - Visual Studio : http://msdn.microsoft.com/fr-fr/library/bb385902.aspx
- Outils de suivi des problèmes (issue tracking system)
  - Jira (atlassian)
- Outils de couverture de code
  - Couverture de code : pure coverage

Qualité du développement

- Outils de tests mémoire
  - Fuite mémoire (memory leaks) : absence de désallocation mémoire : la mémoire grossit en permanence.
  - Accès à de la mémoire non allouée
  - Accès à de la mémoire libérée
  - Double désallocations
  - Outils : Valgrind (Linux) Purify, BoundsChecker, Insure++ (Windows)

Fuites mémoires : Memory leaks

- When a button is pressed:
  - Get some memory, which will be used to remember the floor number
  - Put the floor number into the memory
  - Are we already on the target floor?
    - If so, we have nothing to do: finished
    - Otherwise:
      - Wait until the lift is idle
      - Go to the required floor
      - Release the memory we used to remember the floor number
- Fuite mémoire si on appuie sur le bouton de l'étage courant.

Qualité du développement

- Outils de profiling : détection des parties critiques du code, en terme de performance
  - Quantify, gprof, prof
  - Visual Studio Profiler http://msdn.microsoft.com/fr-fr/magazine/cc337887.aspx (DEMO)
  - AMD CodeAnalyst, Intel Vtune
  - VisualVM, async profiler, JITWatch

- Optimisations guidées par le profil
  - On fait plusieurs « run »
  - Le compilateur s'adapte (ex: définit les tests par défauts différemment)

Profilage

- **Par Sampling**
  - On fait tourner le code et toutes les ms (ou moins) on regarde dans quelle fonction on est : on établit des statistiques d'utilisation
  - Avantage : le code n'est pas modifié, et cela donne une idée réelle
  - Inconvénient : pas toujours très précis
- **Par Instrumentation**
  - On modifie le code afin de pouvoir mesuer précisément le temps passé dans chaque fonction
  - Avantage : résultats assez fins
  - Inconvénient : modifie le code donc ne mesure pas exactement la réalité

Qualité du développement

- Outils d'obfuscation (obfuscateur) : modifie un code source afin de le rendre illisible. Surtout pour Java et .NET (décompilables facilement)
  - http://en.wikipedia.org/wiki/Obfuscated_code
  - RetroGuard, ProGuard et yGuard pour Java
  - Visual Studio Dotfuscator pour .NET http://msdn.microsoft.com/en-us/library/ms227240.aspx

Qualité du logiciel

- Facilité d'utilisation
- Accessibilité

- On doit pouvoir faire simplement des choses simples
- On peut accepter de la complexité pour des choses complexes

- La majeure partie des utilisateurs est de niveau moyen : pas débutant et pas experts

- Voir section architecture

Génie Logiciel

- Optimisation du code

Optimisation du code

- Ne pas optimiser n'importe quelle partie du code
- Chercher le code critique
- Ne pas chercher à optimiser bêtement
  - On fait une boucle complexe au lieu de 2 simple
  - Le compilateur sait faire plein d'optimisations
  - On lit moins de données dans un fichier (l'accès est couteux)

Optimisation du code

- Optimisation
  - Du modèle / de l'architecture (on considère le problème sous un autre angle)
  - De l'algorithme / de méthode
  - Du code source

- Exemple : résolution de problème difficiles. Gains potentiels
  - Changement de modèle : 1 000 000
  - Changement d'algorithme de recherche : 1 000
  - Optimisation du code source : 10

Optimisation du code

- Exemple : résolution de problème difficiles. Gains potentiels
  - Changement de modèle : 1 000 000
  - Changement d'algorithme de recherche : 1 000
  - Optimisation du code source : 10
- Projet ROCOCO (plusieurs années, 6 personnes impliquées)
  - Changement de modèle : 5%
  - Changement d'algorithme de recherche : 20%
  - Optimisation du code source : 75%

Optimisation du code

- Optimisation
  - Du modèle / de l'architecture (on considère le problème sous un autre angle)
    - Très difficile
    - Peut ne conduire à aucun résultat
  - De l'algorithme / de méthode
    - Assez difficile
    - Peut ne pas bien marcher, amélioration pas certaine
  - Du code source
    - Beaucoup plus facile
    - Résultats à peu près certains

Amélioration de l'algorithme

- Regardez la littérature
- Chercher les bons mots clés (doivent marcher dans wikipédia) puis DBLP et google scholar
- Trouver des bonnes pages web de chercheurs ou de gens particulièrement intéressés
- Attention aux résultats uniquement théoriques
  - Tri : au moins n log(n) en théorie
  - Ok, mais uniquement si comparaison
  - Tri de nombres : presque linéaire ! Voir Cormen

Amélioration du code

- Documentations
  - Intel : http://www.intel.com/Assets/PDF/manual/248966.pdf
  - AMD : http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF
  - C++ : http://www.agner.org/optimize/optimizing_cpp.pdf http://www.open-std.org/jtc1/sc22/WG21/docs/TR18015.pdf* http://en.wikibooks.org/wiki/Optimizing_C%2B%2B

Génie Logiciel

- Protection / Secrets (Obfuscation/obfucateur)

Protection du code / Secrets

- Il ne s'agit pas ici de brevet logiciel
- Comment protéger certaines parties du code
  - Obfuscation
    http://en.wikipedia.org/wiki/Obfuscated_code
  - Ne pas les donner! Donc pas de .h détaillés

- Idées de base :
  - on crée une classe encapsulant une autre classe.
  - on place la classe secrète dans une librairie
  - on minimise les .h visibles

Protection du code / Secrets

- Class MaSuperStructureDeDonnee {
  int* superdataastucieux
  …
  void superastucepourajouter(int i){…}

  public:
  ajouter(int i){superastucepourajouter(i);}
  …};
- .h visible
  Class VisibleSdd {
  MaSuperStructureDeDonnee* super; //ptr
  public:
  ajouter(int i);
  …}
- .cpp invisible
  VisibleSdd::ajouter(int i){super->ajouter(i);}

- Inconvénient : on perd de la performance
- Avantage : introduit de la souplesse

Génie Logiciel

- Multithreading

Multithreading

- Il est indispensable de penser au multithreading de nos jours
- Attention : multithreading différent de parallélisme au sens large
  - Pas la même chose que les systèmes répartis
    - Pas de problèmes de pertes de messages, de dates
    - Se méfier des sémaphores, mutex etc… : c'est lourd!
    - Tendance actuelle : spin lock et attente active
  - Mémoire partagée
- Systèmes multicores sont communs

Multithreading

- Apprendre à penser différemment
  - Il va falloir utiliser plusieurs cœurs pour une même application
  - Pas de variables partagées non protégées
  - Design Pattern : Moniteur (on regroupe les objets partagés)
  - Attention aux variables statiques (singleton) : voir thread local storage

Super Multithreading

- Ordinateur vectoriel = GPU
- Super multithreading = massivement parallèle
  - OpenCL
  - CUDA
  - GPGPU : general purpose GPU

Génie Logiciel

- Portages

Portages

- Le même logiciel peut être utilisé avec
  - Des compilateurs différents
  - Des machines différentes
  - Des OS différents
  - Des langages différents

- Ce n'est pas une obligation, mais cela peut se produire
- Le portage 32bits/64bits ont presque disparus (même le Raspberry Pi 4 est en 64 bits)

Partages

- Code utilisables avec plusieurs compilateurs/machines/OS
  - Explique en partie le succès de Java
  - On prévoit dès le début une utilisation multi-systèmes
  - .h (hardware.h) définissant les constantes et spécificités de chaque compilateur, chaque machine, chaque OS
  - On rend le code le plus indépendant possibles des compilateurs/machines/OS
    - Utilisation d'un sous-ensemble du langage
    - Gestion mémoire contrôlée
    - Gestion des threads contrôlée (Windows vs POSIX)
  - Franchement difficile avec le graphique
  - Utilisation de librairies permettant cela
  - Demande une très bonne connaissance des compilateurs/machines/OS

Portages

- Wrapper : utilisation d'un code écrit dans un langage à partir d'un autre langage
  - Fine couche de code qui permet de traduire une interface en une autre interface. Ex code C et classe C++ autour
  - Utiliser pour simplifier, pour clarifier
  - Utiliser pour rendre proprement compatible des formats d'apparences incompatibles

- Java Native Interface
  - framework qui permet à du code Java s'exécutant à l'intérieur de la JVM d'appeler et d'être appelé par des applications natives écrites dans d'autres langages

Génie Logiciel

- Installer

Installer

- Programme permettant l'installation propre de programme. Gère aussi la désinstallation
  - InstallShield
  - Windows installer
  - Nullsoft scriptable install system (NSIS) http://nsis.sourceforge.net/Main_Page

Installer

- Le test de l'installation d'un logiciel n'est pas simple parce que
  - Les machines ne sont pas toutes les mêmes
    - les logiciels installés peuvent être différents
    - L'historique d'installation/désinstallation ne sont pas les mêmes
  - Si une installation a précédemment échouée, elle a pu laisser des traces utiles à l'installation courante

- Idée : virtualisation
  - Avantage :
    - on repart d'un système propre et vierge à chaque fois
  - Inconvénients
    - La virtualisation peut contenir des bugs
    - Ne correspond à aucune installation réelle

- Idée : containeurs (docker)
  - Virtualisation non locale et portable (cloud computing)

Génie Logiciel

- Maintenance

Maintenance

- Maintenance
  - **corrective** : corriger les bugs
    - défauts de fonctionnement
    - non-conformités d'un logiciel,
  - **évolutive** : faire évoluer l'application
    - en l'enrichissant de fonctions ou de modules supplémentaires
    - en remplaçant une fonction existante par une autre
    - en proposant une approche différente.
  - **adaptative** :
    - adapter l'application pour qu'elle fonctionne sur des versions plus récentes des logiciels de base (sans changer la fonctionnalité du logiciel)
    - migrer l'application sur de nouveaux logiciels de base (système d'exploitation, système de gestion de base de données).

Maintenance

- Correction de bugs
  - Bugs de code simple
  - Bugs d'algorithme : l'algorithme est faux
  - Bugs conceptuels plus complexe (le code est juste mais ce n'est pas cela qu'il faut faire)
- Minimiser les modifications intrusives
  - A chaque nouveau code de nouveaux bugs sont introduits, donc il faut éviter d 'écrire du code pour rien
  - Régression très désagréable. Introduction dans nouvelles versions de problèmes qui n'existaient pas avant :
    - bugs
    - performance temps/mémoire

Génie Logiciel

- Evolution / Réécriture

Evolution

- Il est nécessaire de pouvoir faire évoluer un logiciel
- Un logiciel n'est jamais utilisé comme son concepteur l'avait pensé ou de la façon dont le concepteur l'avait imaginé.
- Un logiciel évolue
  - Parce que le concepteur a des nouvelles idées
  - Parce que les utilisateurs ont de nouvelles requêtes
  - Parce la compétition est là.

Evolution

- Il n'est pas facile de faire évoluer un logiciel, surtout s'il est mal structuré
  - On ne comprend pas bien l'agencement des modules
  - On ne mesure pas bien l'impact des modifications
  - Une mauvaise modularité impose un travail trop important
- La possibilité d'évoluer doit être prise en compte tout le temps
- Cela permet aussi de fixer des limites au développement courant
  - On fera cette partie dans la prochaine version
  - On améliorera cette autre partie dans les prochaines versions

Réécriture / Refactoring

- De temps en temps il faut réécrire/repenser certaines parties du code
- A partir de combien de « if » une fonction doit-elle être réécrite ?
- Les requêtes évoluant, il est normal que le code aussi.

Refactorisation / Réusinage

- La **refactorisation** (refractoring) est une opération de maintenance du code informatique. Elle consiste à retravailler le code source non pour ajouter une fonctionnalité supplémentaire au logiciel mais pour
  - améliorer sa lisibilité,
  - simplifier sa maintenance,
  - changer sa généricité (on parle aussi de remaniement).

Refactorisation

- Au fur et à mesure de la vie d'un logiciel on est amené
  - à implémenter de nouvelles fonctions
  - à corriger des bugs.

- Ces modifications ne s'imbriquent pas toujours avec élégance dans l'architecture du logiciel.

- Le code source devient de plus en plus complexe .De temps en temps il faut le simplifier
  - Supprimer les informations redondantes/dupliquées
  - Simplifier l'algorithmique des méthodes
  - Limiter la complexité des classes
  - Limiter le nombre de classes

Refactorisation

- **Refactorisation simple**
  - Modification de la présentation (mise en page / indentation / commentaires)
  - Modification de l'algorithmique
  - Relocalisation de procédures
- **Refactorisation complexe : Refonte de la conception**
  - modifier la hiérarchie de classes composant l'application.
  - modification de la structure du code
- **Activités de refactorisation**
  - Suppression du code mort
  - Ajout d'assertions (mise en place d'assert)
  - Renommage des classes et des méthodes pour être plus en cohérence avec leur rôle

Génie Logiciel

- Architecture Logicielle

Architecture Logicielle

- L'**architecture logicielle** décrit d'une manière symbolique et schématique les différents composants d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions.
- Complémentaire de l'analyse fonctionnelle.
- Analyse fonctionnelle
  - Décrit ce que doit réaliser un système, le « quoi »
- Architecture logicielle
  - Décrit comment le système doit être conçu pour répondre aux spécifications, le « comment »

Architecture Logicielle

- Objectifs
  - Réduction des coûts
  - Augmentation de la qualité

Architecture Logicielle

- **Norme ISO 9126 Qualité logicielle**
  - **Interopérabilité**
  - **Portabilité**
  - **Compatibilité :**
  - **Validité**
  - **Vérifiabilité**
  - **Intégrité**
  - **Fiabilité**
  - **Maintenabilité**
  - **Réutilisabilité**
  - **Extensibilité**
  - **Efficacité**
  - **Autonomie**
  - **Transparence**
  - **Composabilité**
  - **Simplicité d'utilisation**

Architecture Logicielle

- **Interopérabilité**
  - capacité du logiciel à communiquer et à utiliser les ressources d'autres logiciels
- **Portabilité**
  - possibilité de compiler le code source et/ou d'exécuter le logiciel sur des plates-formes (machines, systèmes d'exploitation, environnements) différents.
- **Compatibilité**
  - possibilité de fonctionner correctement dans un environnement ancien (compatibilité descendante) ou plus récent (compatibilité ascendante).
- **Validité**
  - conformité des fonctionnalités du logiciel avec celles décrites dans le cahier des charges.
- **Vérifiabilité**
  - simplicité de vérification de la validité.

Architecture Logicielle

- **Intégrité**
  - faculté à protéger ses fonctions et ses données d'accès non autorisés.
- **Fiabilité**
  - faculté à gérer correctement ses propres erreurs de fonctionnement en cours d'exécution.
- **Maintenabilité**
  - simplicité de correction et de modification du logiciel
- **Réutilisabilité**
  - capacité de concevoir le logiciel avec des composants déjà conçus
  - réutilisation simple de ses propres composants pour le développement d'autres logiciels.
- **Extensibilité**
  - possibilité d'étendre simplement les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité.

Architecture Logicielle

- **Efficacité**
  - capacité à exploiter au mieux les ressources offertes par la ou les machines où le logiciel sera implanté.
- **Autonomie**
  - capacité de contrôle de son exécution, de ses données et de ses communications.
- **Transparence**
  - capacité pour un logiciel de masquer à l'utilisateur (humain ou machine) les détails inutiles à l'utilisation de ses fonctionnalités.
- **Composabilité**
  - capacité pour un logiciel de combiner des informations provenant de sources différentes.
- **Simplicité d'utilisation**
  - facilité d'apprentissage et d'utilisation du logiciel par les usagers.

Architecture Logicielle

- Au moins 5 types
  - Approche ascendante : Développement pour et par la réutilisation
  - Approche descendante : Architecture en appels et retours
  - Architecture en couches
  - Architecture centrée sur les données
  - Architecture orientée objets

Architecture Logicielle

- **Approche ascendante : Développement pour et par la réutilisation**
- La réutilisation de composants logiciels est l'activité permettant de réaliser les économies les plus substantielles
- La réutilisation de composants nécessite de
  - créer une architecture logicielle permettant une intégration harmonieuse de ces composants.
  - créer et maintenir une bibliothèque logicielle

Architecture logicielle

- **Approche ascendante : Développement pour et par la réutilisation**
- Avec une bibliothèque on renverse les rôles
- Créer une application revient
  - à créer les composants de bibliothèque nécessaires
  - puis à construire l'application à l'aide de ces composants.
- Une telle bibliothèque, facilitant le développement d'application est un framework
- L'architecte doit
  - trouver les composants logiciels appropriés
  - créer les composants manquants, les documenter et les intégrer à la bibliothèque.

Architecture Logicielle

- **Approche descendante : Architecture en appels et retours**
  - Par raffinement successifs (N. Wirth) : diviser pour régner
  - Découper une fonctionnalité en sous-fonctionnalités qui sont également divisées en sous sous-fonctionnalités et ainsi de suite
  - Très utilisée

Architecture Logicielle

- **Architecture en couches**
  - La conception de logiciels nécessite de recourir à des bibliothèques.
  - Une bibliothèque très spécialisée utilise des bibliothèques moins spécialisées qui elles-mêmes utilisent des bibliothèques génériques
  - Les nouveaux composants utilisent les anciens et ainsi de suite, la bibliothèque tend donc à devenir une sorte d'empilement de composants.
  - Couche = composants possédant une grande cohésion (sémantiques semblables)
  - Architecture en couches = un empilement de couches.
  - Tous les composants des couches supérieures dépendants fonctionnellement des composants des couches inférieures.
  - Très utilisée (trop ?)

Architecture Logicielle

- **Architecture centrée sur les données**
  - Un composant central, baptisé serveur de données (SGBD, Datawarehouse, Blackboard), est responsable de la gestion des données (conservation, ajout, retrait, mise-à-jour, synchronisation, ...)
  - Les composants périphériques, baptisés clients, utilisent le composant central
  - Le serveur peut être
    - passif (SGBD, Datawarehouse) en ne faisant qu'obéir aux ordres
    - actif (Blackboard) en notifiant un client si un changement qui le concerne se produit.
  - Séparation claire des données (serveurs) des traitements et de la présentation (clients)
  - Avantage : très grande intégrabilité : des clients peuvent être ajoutés sans affecter les autres clients.
  - Défaut : tous les clients sont dépendants de l'architecture des données qui doit rester stable et qui est donc peu extensible.

Architecture Logicielle

- **Architecture orientée objets**
- Les composants du système (objets) intègrent des données et les opérations de traitement de ces données.
- La communication et la coordination entre les objets sont réalisées par un mécanisme de passage de messages.
- Encapsulation = architecture détaillée de chaque objet
  - Regroupement des données et des méthodes concernant les mêmes objets
  - Données accessibles par une couche d'interface et non par accès direct
  - Masquage des sous-fonctions inutiles pour utiliser l'objet
- Héritage
  - Regroupe les fonctionnalités communes à plusieurs classes dans un même objet
  - Permet d'éviter la redondance de code
  - Facilite l'extensibilité
- Polymorphisme
  - utilise des objets différents (possédant des comportements distincts) de manière identique
  - Permet de définir des méthodes aux comportements semblables sur des objets différents

Génie Logiciel

- Conception de code
- SOLID
- Philosophy of Software Design
- Design Patterns et anti-patterns

Conception de code

- Quelque soit le métier que vous comptez faire dans l'informatique, il est préférable de savoir coder
  - Estimation du temps de développement
  - Compréhension des difficultés internes / des clients
  - Compréhension des avantages et des inconvénients des produits
  - Il est difficile de faire des produits simples quand on ne sait pas coder : qu'est-ce qui est complexe ou pas ?
  - Vous avez besoin de recul
  - Les informaticiens ont de la considération pour les bons codeurs

Conception de code

- Ecrire un code propre qui puisse évoluer est difficile
- Le code « sale » permet de gagner du temps à court terme, mais coute très cher ensuite
  - Bugs
  - Design complexe
  - Difficilement modulable
  - Evolution non pensée
- Evolution peut entrainer une dégradation de la qualité du code et à la fin une non maitrise du code
  - Faillites connues

Le mauvais code

- C'est du code qui est très dépendant de son utilisation
- Coder souvent très rapidement pour répondre à un besoin spécifique
- Image : « Couler dans le bronze »
- L'enchevêtrement des fonctions est très grand
  - Dès que l'on touche à quelque chose, on est amené à toucher autre chose
  - Vision globale à peu près impossible

Le mauvais code

- Contient des fonctions avec « effet de bord »
- Une fonction est dite à **effet de bord** (side-effect) si elle modifie un état autre que sa valeur de retour.
  - modifie une variable statique ou globale
  - modifie un ou plusieurs de ses arguments
  - écrit des données vers un écran ou un fichier
  - lit des données provenant d'autres fonctions à effet de bord.
- Les effets de bord rendent souvent le comportement des programmes plus difficiles à comprendre.

Effet de bord

```
#include <iostream>
using namespace std;
int a;
void f() {
        a = 2;
}
int main () {
        a = 1;
        cout << a << endl;
        f();
        cout << a << endl;
}
```

Le mauvais code

- Le problème majeur : la productivité décroit
- On n'ose plus toucher au programme
- On écrit des interfaces de partout
  - On les comprend
  - On sait ce qu'elle font
  - Ca minimise les interactions avec le mauvais code

- Problème : ca a le même effet que le mauvais code
  - Tout est complexifier
  - On ne peut toucher à rien

Le mauvais code

- Syndrome de la vieille voiture :
  - On possède une vieille voiture en bon état voir très bon état. On la nettoie souvent, on essaie d'en prendre soin
  - Un jour, on heurte quelque chose. Le pare-choc est abimé, c'est bien ennuyeux. Ca coûte trop cher de le réparer, alors on laisse comme ça.
  - La voiture est maintenant rayée, mais comme le pare-choc est abimé on n'essaie plus de camoufler la rayure
  - La voiture prend un coup de vieux et le délabrement s'accélère
- Le logiciel évolue de la même façon

Le mauvais code

- Pourquoi écrit-on du mauvais code ?
  - Pressé par le temps
  - On en a marre de ce code
  - On a la flemme de le repenser
- Règle pour le code
  - Plus tard signifie jamais

Le mauvais code

- Supprimer du mauvais code n'est pas simple
  - Les produits existent, ce n'est pas si simple de les réécrire. Il faut se mettre d'accord sur la réécriture
    - Que fait-on ?
    - Qui le fait ?
  - La direction préfère que vous soyez plus productif (au sens nouveau produit)
  - Est-ce que cela vaut la peine ?

- On réécrit.

Le mauvais code

- Réécriture totale :
  - Pas seulement un changement de version (maintenance peut être problématique)
  - En général un changement de nom du produit
    - Windows 11, 10, 7 vs Vista vs XP : pas la même chose que les Service Packs
    - Pas toujours simple : Office, Firefox etc…

Le mauvais code

- Comment l'éviter ?
  - Dire la vérité : à la direction, aux gens du marketing, aux chefs de projets
  - C'est risqué (licenciement) mais c'est responsable
  - Les codeurs doivent assumer un certain pouvoir
    - Si un patient dit à un chirurgien qu'il ne devrait pas se laver les mains avant d'opérer le chirurgien doit refuser
- Avec l'expérience, on sait qu'on n'a pas le choix : on doit éviter le mauvais code

Le code propre

- Qu'est-ce que c'est ?
  - B. Stroustrup (inventeur du C++)
    - Un bon code fait une chose et la fait bien
    - Simple (pour détecter facilement les bogues)
    - Agréable, élégant (pour la lisibilité)
    - Dépendances minimes (pour la maintenance)
    - Gestion des erreurs totales
    - Perfs proches de l'idéal (pour éviter l'optimiseur fou qui va le dégrader)

Le code propre

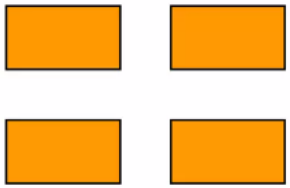
- G. Booch, (Object Oriented Analysis and Design with Applications)
  - Ne cache jamais les intentions du concepteur
  - Constitué d'abstractions nettes et de lignes de contrôle franches.
  - Simple et direct.
- Dave Thomas, fondateur d'OTI, parrain de la stratégie d'Eclipse
  - Un code propre peut être lu et amélioré par un développeur autre que l'auteur d'origine.
  - Il dispose de tests unitaires et de tests de recette
  - Il fournit une API claire et minimale

Principes

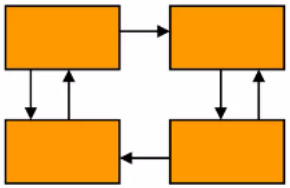
- DRY (don't repeat yourself)
  - Ne pas implémenter les mêmes choses plusieurs fois
- KISS (keep it simple stupid)
  - Eviter la complexité à tout pris
- YAGNI (You ain't gonna need it)
  - Ne pas implémenter des choses si vous n'êtes pas certain qu'elles seront utiles

Couplage (coupling)

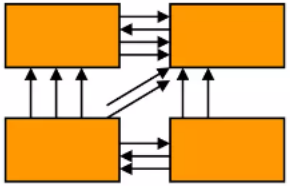
- Degré de dépendance entre les composants
- En gros les include (C/C++) et les imports (Java)



No dependencies



Loosely coupled-some dependencies



Highly couples-many dependencies

Couplage

- Facon d'être dépendant
  - Référence entre composants
    - A invoque B
    - A dépend de B pour faire son traintement
  - Quantité d'information passé d'un composant à un autre : un paramètre, un tableau, un bloc
  - Contrôle d'un composant sur un autre
    - Control flag passé à B
  - Degré de complexité de l'interface entre composants
    - C et D echange des informations avant que D puisse terminer son exécution

Types de couplage

- Couplage de contenu (le pire) : un composant altère un autre composant (modifie ses données)
- Couplage commun : deux composants communiquent via une variable globale
- Couplage externe : deux modules sont liés par un élement extérieur au logiciel (machines…)
- Couplage de contrôle : un flag de contrôle permet à deux modules de communiquers

Types de couplage

- Stamp coupling : la communication se fait par une structure de données passée en paramètre et la structure de données contient plus d'information que le recepteur n'en a besoin
- Couplage de donnée (le mieux): la communication se fait par le passage de paramètres. On ne passe pas plus de paramètres que nécessaire
- Pas de couplage : les modules sont indépendants

Génie Logiciel

- Conception de code
- SOLID
- Philosophy of Software Design
- Design Patterns et anti-patterns

SOLID

- Cinq principes de conception destinés à produire des architectures logicielles plus compréhensibles, flexibles et maintenables.
- Proposé part Robert C. Martin.

SOLID

- Responsabilité unique (Single responsibility principle)
  - une classe, une fonction ou une méthode doit avoir une et une seule responsabilité
- Ouvert/fermé (Open/closed principle)
  - une entité applicative (classe, fonction, module ...) doit être fermée à la modification directe mais ouverte à l'extension
- Substitution de Liskov (Liskov substitution principle)
  - une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme
- Ségrégation des interfaces (Interface segregation principle)
  - préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale
- Inversion des dépendances (Dependency inversion principle)
  - il faut dépendre des abstractions, pas des implémentations

Responsabilité unique

- Module qui compile et imprime un rapport. Ce module peut changer pour deux raisons.
  - le contenu du rapport peut changer.
  - le format du rapport peut changer.
- Ces deux choses changent pour des causes différentes : l'une substantielle, et l'autre cosmétique. Le principe de responsabilité unique dit que ces deux aspects du problème ont deux responsabilités distinctes (i.e deux tâches différentes), et devraient donc être dans des classes ou des modules séparés.
- Cela rend la classe plus robuste

Principe ouvert/fermé

- On considère une classe d'une librairie (elle est donc « terminée »)
- Ouvert signifie que cette classe a la capacité d'être étendue.
- Fermée signifie qu'elle ne peut être modifiée que par extension, sans modification de son code source.

- En général on procèdera par héritage pour l'étendre. On ne changera pas la classe en elle-même.

Substitution de Liskov

- Si S est un sous-type de T , alors tout objet de type T peut être remplacé par un objet de type S sans altérer les propriétés désirables du programme concerné.

LSP: Liskov substitution principle

- L'exemple classique d'une violation du LSP est la suivante :
- Soit une classe Rectangle représentant les propriétés d'un rectangle : hauteur, largeur. On lui associe donc des accesseurs pour accéder et modifier la hauteur et la largeur librement. En postcondition, on définit la règle : la hauteur et la largeur sont librement modifiables.
- Soit une classe Carré que l'on fait dériver de la classe Rectangle. En effet, en mathématiques, un carré est un rectangle. Donc, on définit naturellement la classe Carré comme sous-type de la classe Rectangle. On définit comme postcondition la règle : les « quatre côtés du carré doivent être égaux ».
- On s'attend à pouvoir utiliser une instance de type Carré n'importe où un type Rectangle est attendu.
- Problème : Un carré ayant par définition quatre côtés égaux, il convient de restreindre la modification de la hauteur et de la largeur pour qu'elles soient toujours égales. Néanmoins, si un carré est utilisé là où, comportementalement, on s'attend à interagir avec un rectangle, des comportements incohérents peuvent subvenir : les côtés d'un carré ne peuvent être changés indépendamment, contrairement à ceux d'un rectangle.
- Solutions :
  - Une mauvaise solution consisterait à modifier les setter du carré pour préserver l'invariance de ce dernier. Mais ceci violerait la postcondition des setter du rectangle qui spécifie que l'on puisse modifier hauteur et largeur indépendamment.
  - Une solution pour éviter ces incohérences est de retirer la nature Mutable des classes Carré et Rectangle. Autrement dit, elles ne sont accessibles qu'en lecture. Il n'y a aucune violation du LSP, néanmoins on devra implémenter des méthodes "hauteur" et "largeur" à un carré, ce qui, sémantiquement, est un non sens.
  - Une meilleure solution consiste à ne pas considérer un type Carré comme substitut d'un type Rectangle, et les définir comme deux types complètement indépendants. Ceci ne contredit pas le fait qu'un carré soit un rectangle. La classe Carré est un représentant du concept « carré ». La classe Rectangle est un représentant du concept « rectangle ». Or, les représentants ne partagent pas les mêmes propriétés que ce qu'ils représentent

Ségrégation des interfaces

- Aucun client ne devrait dépendre de méthodes qu'il n'utilise pas.
  - Il faut donc diviser les interfaces volumineuses en plus petites plus spécifiques, de sorte que les clients n'ont accès qu'aux méthodes intéressantes pour eux.
  - Ces interfaces rétrécies sont également appelées interfaces de rôle. Tout ceci est destiné à maintenir un système à couplage faible, donc plus facile à refactoriser.

Ségrégation des interfaces

- Au lieu d'avoir une grosse classe Job d'une imprimante multifonction qui fait tout , on crée plusieurs interfaces suivant le type de travail que l'on veut faire : scanner, imprimer, agrafer, etc…
- Autre exemple bien connu: le distributeur de billets avec retraits et dépôts

Inversion des dépendances

- Les deux assertions de ce principe sont :
  - Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
  - Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.
- En programmation classique, les modules de haut niveaux dépendent des modules de bas niveau. Et on construit des modules de plus en plus abstraits donc de plus en plus haut niveau en « empilant » les modules.
  - Pour éviter cette dépendance très forte on propose de changer la relation H dépend de B, en:
    - H dépend de IB et B implémente IB où IB est une interface de B

Génie Logiciel

- Conception de code
- SOLID
- Philosophy of Software Design
- Design Patterns et anti-patterns

Clean code

- Robert Martin (Uncle Bob) a proposé la concept de clean code: « Clean Code: A Handbook of Agile Software Craftsmanship »
  - Contient de bonnes idées
    - a function should not perform both high-level and low-level tasks, because this is confusing and muddles the function's responsibility
  - Mais contient aussi de mauvais conseils (en plus trop dirigistes)
    - Une fonction ne doit pas faire plus de 20 lignes de code
    - Pas de paramètres booleans
    - 3 paramètres c'est trop (ideal = 0 parametre)

Philosophy of Software Design

- Excellent book of John Ousterhout, Pr at Stanford University
- Available: https://milkov.tech/assets/psd.pdf

Complexité

- But éviter la complexité ☺ Vœux pieu…
- Complexité = tout ce qui rend le système compliqué à comprendre et à modifier
- Ne pas être péremptoire : pas lier au nombre de lignes d'une fonction, aux classes…
  - **Change amplification** : on veut modifier quelque chose et ca demande de modifier de nombreuses classes et de nombreux codes
  - **Cognitive load** : Qu'est-ce que le développeur doit savoir pour faire une modification ? Il faut apprendre pour savoir donc plus il ya de choses à savoir plus ca va prendre du temps (cas du GC en Java vs C).
    - Plus de lignes de code peut réduire le cognitive load
  - **Unknown unknowns :** Qu'est-ce que je dois modifier dans le code; Où est-ce ? Par exemple avec la fonction search je dois pouvoir trouver des informations

Exemple

- Page web avec background
  - Background définit pour chaque page; pb avec des milliers de pages
    - Fichier de style !
  - Probleme pour certaines pages qui modifient le background en fonction du fichier de style
    - Je dois le savoir !
    - Comment retrouver ces pages ? Avec une page de style je peux chercher les variables et essayer de trouver qui les modifient

Cause de la complexité

- Dépendance
- Obscurité

Programmation tactique vs stratégique

- **Tactical programming** : ce qui compte c'est que ca marche !
  - Si on s'arrete là alors on a aura difficilement un bon système
  - Tactical tornado : le developpeur qui pousse cette idée à l'extrème : je code tout, tout de suite et super vite : « moi au moins j'ai une solution qui tourne ». Souvent bien vu par les managers (surtout si ceux-ci sont un peu fainéant). Ceux qui héritent de leur code les détestent : il faut revoir tout leur code, et tout reconsidérer, corriger les bugs etc… Donc c'est long ! Et ces développeurs sont souvent moins bien considérés car lents

Programmation tactique vs stratégique

- Strategic programming : Avoir un code qui marche n'est pas suffisant. Le plus important est le fonctionnement à long terme; Un bon design qui fonctionne doit être le but.
  - Ca demande plus de temps et ca demande de réfléchir
  - Il est intéressant d'améliorer le système à chaque fois qu'on peut le faire

Les modules doivent être profond

- La programmation modulaire propose de décomposer le système en une collection de modules plus ou moins indépendants.
- Modules = sous systèmes, classes…
- Un module est composé de deux parties
  - Son interface
  - Son implémentation

- Interface: ce que le module peut faire pour l'utilisateur, mais pas comment
- Implementation: tout ce qui est nécessaire pour réaliser l'interface

- Attention : tout ce qui est public peut être utilisé par un utilisateur (ou c'est comme cela que l'utilisateur le comprendra)

Exemple : un indexeur

- Un indexeur attribue un index différent à chaque élément et à partir de l'element on peut retrouver son index
- Interface
  - Int Indexer.push(elt): ajoute l'éléments s'il n'est pas déjà dans l'indexeur et retourne son index
  - Int Indexer.index(elt): retourne l'index d'un element et -1 s'il n'est pas dans l'indexer
  - Pas d'autres fonctions

Exemple : un indexeur

- Un indexeur attribue un index différent à chaque élément et à partir de l'element on peut retrouver l'index
- Implementation : certainement par une « table de hash ». Par quelle table de hash ?

Interface à la surface

- L'interface doit être light et les modules profonds
- Le nombre de lignes ou la complexité des fonctions des modules n'a pas d'importance
- Gestion des fichiers en Unix/C: excellent
  - ssize_t read(int fd, void* buffer, size_t count);
  - ssize_t write(int fd, const void* buffer, size_t count);
  - off_t lseek(int fd, off_t offset, int referencePosition);
  - int close(int fd);

Classitis

- Défaut de Java: trop de classes. Trop d'intermédiaires. Trop de petites classes
  - FileInputStream fileStream =new FileInputStream(fileName);
  - BufferedInputStream bufferedStream =new BufferedInputStream(fileStream);
  - ObjectInputStream objectStream = new ObjectInputStream(bufferedStream);
- On ne réutilisera jamais le fileStream ni le bufferStream, alors à quoi ca sert ?
- **Les cas simples doivent etre simples**

Cacher l'information

- Comment créer des modules profond
  - On cache l'information
    - La structure de données est cachée
    - Le protocole TCP/IP est caché
    - Le parser JSON est caché
    - Le schedule des threads est caché
- Interface simplifiée
- Evolution simplifiée

Cacher l'information

- Avoid exposing internal (implementation) variables
  - On a un tableau en interne on ne fait pas de getter de ce tableau.

Révéler l'information

- Une information impacte plusieurs modules
- Format de fichier partagés entre modules
- Comment puis je réorganiser l'information de façon à ce que cela ne soit pas trop répendu ? Peu de classes partage le format de fichiers

Temporal decomposition

- Cela fait partie de l'information révélée :
- On fait des suppositions sur l'ordre d'appels de certaines opérations
- Fichier : lecture - modification – ecriture. Modele ?
- Cas 1: 3 classes -> module qui lit, un qui modifie, uun qui ecrit.
  - Format du fichier est exposé et partagé par le module de lecture et d'ecriture. Ce n'est pas bon
- Cas 2: Une classe pour les mécanismes de bases : lecture et ecriture ensemble. C'est le bon format
- Quand on ecrit un module : on se focalise sur ce dont on a besoin pour effectuer une tache et pas sur l'ordre des taches

- Typiquement on a des classes plus grandes. Mais moins de couplage entre les classes

Surexposition

- Map<String> parameters(){…} expose la structure interne. On ne doit pas avoir besoin de savoir ce qui est fait en interne

General purpose module?

- Quand doit-on développer un module à vocation générale plutôt que juste spécifique pour ce qu'on veut faire ?
  - Faire des classes somewhat general purpose
    - Ca repond au besoin instantanée
    - L'interface permet d'être générale
- Example: editeur de textes:
  - Pleins de petites fonctions (backspace, delete car…)
  - Moins de fonctions plus simples à comprendre
    - Insert(Position p, String text)
    - Delete (Position start, Position end)

General purpose module?

- Est-ce que l'API est simple à comprendre ?
- Comment vais-je l'utiliser ?
- Est-ce qu'elle couvre tous mes besoins ?

Different layers, different abstractions

- Les logiciels ont souvent different niveaux (layers) empiler: très bas niveau, niveau moyen, haut niveau
  - TCP/IP – transfert sur la reseau – serveur html
- ATTENTION aux **pass-through methods** : méthode qui ne font qu'appeler aune autre méthode avec les mêmes paramètres
  - Probleme de responsabilité entre classes

Le pattern decorator

- A decorator object takes an existing object and extends its functionality
  - provides an API similar or identical to the underlying object, and its methods invoke the methods of the underlying object.
  - BufferedInputStream class is a decorator: given an InputStream object, it provides the same API but introduces buffering.
  - Other example: Window class implements a simple form of window that is not scrollable, and a ScrollableWindow class decorates the Window class by adding horizontal and vertical scrollbars.
- The motivation for decorators is to separate special-purpose extensions of a class from a more generic core.

Pass-through variable

- Au lieu de passer en permanence un parametre qui traverse les niveaux il est souvent plus intéressant d'utiliser un objet contexte

Better together or better appart?

- Il faut essayer de regrouper des classes
- Se poser des questions
  - Est-ce que ces deux classes partagent de l'information ?
  - Est-ce qu'elles sont utilisées ensemble ?
  - Est-ce qu'elle ont un socle commun
  - Est-ce qu'il est facile de les différencier ou bien faut-il regarder les autres classes pour comprendre leur différences ?
- Si le même code commence à être répéter alors il faut se poser des questions
- ATTENTION: si on implemente des spécificites dans la mecanosme général alors le design est mauvais (**special-general mixture**)

Exceptions add complexity

- Réduire le nombre d'endroits dans le code où les exceptions sont traitées
- Les exceptions sont plus difficiles à traiter que le code normal
- Il ne devrait pas avoir plus de code pour les exceptions que pour le code normal.
- bonne gestion des exceptions: ne pas avoir d'exception:
  - Au lieu de renvoyer une exception si un fichier n'est pas présent, le code ne fait rien du tout

Example: suppression de fichier

- Windows:
  - On ne peut pas supprimer un fichier qui est utilisé par un autre processus -> frustrant et énervant. On a une erreur mais on ne sait pas comment la corriger
- Linux:
  - Le fichier est marqué supprimé. On ne peut plus l'ouvrir. Ceux qui travaillent dessus peuvent finir leur travail et modifier le fichier mais quand plus personne ne l'utilisera alors il sera supprimé physiquement

Exemple: java substring

- SI beginIndex ou endIndex ne sont pas bien mis -> IndexOutOfBoundsException
- C'est gênant. On devrait dans ce cas retourner l'interval avec les valeur la plus petite, plus grande ou égal à beginIndex et la plus grande, plus petite ou égal à endIndex. Souvent on veut cela et on doit réimplémenter cela

Mask error, exception aggregation and crash

- Autres techniques de reduction des problèmes dus aux exceptions
  - On recommence l'opération: on n'arrive pas à écrire un fichier alors on recommence avant de lever une exception
  - On regroupe le traitement d'exceptions en un endroit
  - On crashe l'application: OutOfMemory: ca ne sert à rien de checker en permanence. En C il arrive qu'il n'y ait plus assez de mémoire pour afficher le problème !

Commentaires et Nommage

- Les bons commentaires c'est bien
- Il ne faut pas documenter l'implémentation dans l'interface

- Choisir des bons noms de méthodes est difficile mais très important

- Essayez d'être consistent dans votre code

Génie Logiciel

- Conception de code
- SOLID
- Philosophy of Software Design
- Design Patterns et anti-patterns

Design patterns

- **Motifs de conceptions** (parfois patron de conception) décrivent des solutions standard pour répondre à des problèmes d'architecture et de conception des logiciels. Wikipédia
- **But :** Résoudre les problèmes récurrents suivant le paradigme objet.
- **Définition**
  - décrivent des procédés de conception généraux (pas comment résoudre un problème particulier ce qui est le rôle des algorithmes)
  - formalisation de bonnes pratiques, ce qui signifie qu'on privilégie les solutions éprouvées.
  - méthode de conception = manière standardisée de résoudre un problème qui s'est déjà posé par le passé.
  - un outil de capitalisation de l'expérience appliqué à la conception logicielle.
- Grande influence sur l'architecture logicielle

Design patterns

- Avantages :
  - Un vocabulaire commun
  - Capitalisation de l'expérience
  - Un niveau d 'abstraction plus élevé qui permet d'élaborer des constructions logicielles de meilleure qualité
  - Réduire la complexité
  - Réduire le temps de développement
  - Eviter la présence d'anti-patrons
  - Guide/catalogue de solutions
  - Indépendant du langage

Design patterns

- « Design Patterns – Elements of Reusable Object-Oriented Software », Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, 1995.
- Les auteurs = Gang of Four
- Viennent des travaux de Christopher Alexander dans les années 1970

Design patterns

- La description d'un pattern suit un formalisme fixe
  - Nom
  - Description du problème
  - Description de la solution
  - Conséquences
  - …

Design patterns

- 3 types :
  - Creational Patterns
    - gestion de la création d'objets de façon indirecte
  - Structural Patterns
    - organisation des classes en séparant l'interface de l'implémentation
  - Behavioural Patterns
    - organisation des objets pour qu'ils collaborent
- 23 patterns dans le livre
- Il en existe d'autres. On en verra quelques uns

Design Patterns

- Avant-propos
  - Je vais essayer de ne pas théoriser (faire de la théorie) autour des Design Patterns mais d'être le plus concret possible
  - Je vais essayer de donner des exemples génériques mais précis

Design Patterns

- Créational
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton

Abstract Factory

- Permet d'utiliser des classes sans se soucier
  - de leur implémentation réelle
  - des objets réellement crées
- Et en utilisant uniquement des interfaces abstraites

- Exemple :
  - je veux utiliser un bouton graphique qui a une méthode paint avec Windows et OSX.
  - Les détails d'implémentation et les création d'objets m'importe peu.
  - Je veux juste faire : button ←createButton() puis button→paint()

Abstract Factory

```
struct Button {
        virtual void paint() = 0;
};
struct WinButton : public Button {
        void paint (){ std::cout << " I'm a window button \n"; }
};
struct OSXButton : public Button {
        void paint (){ std::cout << " I'm a OSX button \n"; }
};
```

```
struct GUIFactory {
        virtual Button* createButton () = 0;
};
struct WinGUIFactory : public GUIFactory {
        Button* createButton (){ return new WinButton(); }
};
struct OSXGUIFactory : public GUIFactory {
        Button* createButton (){ return new OSXButton(); }
};
```

Abstract Factory

- **LE CODE MAJEUR**

```
struct Application {
        Application(GUIFactory* factory){
                Button * button = factory->createButton();
                button->paint(); }
};
```

- /* application : */

```
int main() {
        GUIFactory* factory1 = new WinGUIFactory();
        GUIFactory* factory2 = new OSXGUIFactory();
        Application* winApp = new Application (factory1);
        Application* osxApp = new Application (factory2);

        delete factory1, factory2;
        return 0;
}
```

Builder

- Le but de ce pattern est d'abstraire les étapes de construction d'un objet
  - Pour décomposer la création d'objets complexes
  - Pour être générique : différentes implémentations de ces étapes conduisent à la construction de représentation d'objets différentes.

- Exemple :
  - Je veux faire des pizzas
  - C'est les ingrédients mis qui définissent une pizza, donc une classe

Builder

- /* Produit */
  class Pizza { // **CONTIENT JUSTE LES CHAMPS**
  private String pate = "";
  private String sauce = "";
  private String garniture = "";
  public void setPate(String pate) { this.pate = pate; }
  public void setSauce(String sauce) { this.sauce = sauce; }
  public void setGarniture(String garniture) {
  this.garniture = garniture; }
  }
- /* Construction Abstraite */
  abstract class CtorPizza {//**CONTIENT LES METHODES INITIALISANT LES CHAMPS**
  protected Pizza pizza;
  public Pizza getPizza() { return pizza; }
  public void creerNouvellePizza() { pizza = new Pizza(); }
  public abstract void mettrePate();
  public abstract void ajouterSauce();
  public abstract void ajouterGarniture();
  }

Builder

- /* Construction Concrete */ //**EXEMPLE INSTANTIATION**

```
class CtorPizzaHawaii extends CtorPizza {
        public void mettrePate() { pizza.setPate("croisée"); }
        public void ajouterSauce() { pizza.setSauce("douce"); }
        public void ajouterGarniture() {
        pizza.setGarniture("jambon+ananas"); }
}
```

- /* Construction Concrete */ //**EXEMPLE INSTANTIATION**

```
class CtorPizzaPiquante extends CtorPizza {
        public void mettrePate() { pizza.setPate("feuilletée"); }
        public void ajouterSauce() { pizza.setSauce("piquante"); }
        public void ajouterGarniture() {
                pizza.setGarniture("pepperoni+salami"); }
}
```

Builder

```
/* Directeur */
// CLASSE QUI GERE LA CREATION DES PIZZAS
class Serveur {
        private CtorPizza ctorPizza;
        public void setCtorPizza(CtorPizza mp) { ctorPizza = mp; }
        public Pizza getPizza() { return ctorPizza.getPizza(); }

        public void construirePizza() {
                ctorPizza.creerNouvellePizza();
                ctorPizza.mettrePate();
                ctorPizza.ajouterSauce();
                ctorPizza.ajouterGarniture();
        }
}
```

Builder

```
/* Un client commandant une pizza. */
class Exemple {
        public static void main(String[] args) {
                Serveur serveur = new Serveur();
                CtorPizza ctorPizzaHawaii = new
                        CtorPizzaHawaii();
                CtorPizza ctorPizzaPiquante = new
                        CtorPizzaPiquante();

                serveur.setCtorPizza(ctorPizzaHawaii);
                serveur.construirePizza();
                Pizza pizza = serveur.getPizza();
        }
}
```

Factory Method

- On crée dynamiquement des objets en fonction de paramètres passés à la fabrique
  - On détermine la classe dont on va créer une instance en fonction du paramètre passé en paramètre
  - Un Animal à partir de son nom
    - « Chat » donne instance de la classe Chat
    - « Chien » donne instance de la class Chien

Factory Method

```
public class FabriqueAnimal {
        Animal getAnimal(String typeAnimal){
                if (typeAnimal.equals("chat")) {
                        return new Chat();
                } else if (typeAnimal.equals("chien")) {
                        return new Chien();
                }
        }
}
```

Prototype

- Création par clonage
  - Gestion de temps de création très long (reprise et modifs des anciens éléments)
  - Copie d'objets sans utiliser les constructeurs par recopie

- On définit une fonction virtuelle clone qui réalise une copie
- Au lieu d'appeler new et de créer un objet from scratch on clone un objet précédent
- Permet de faire des codes indépendants de la notion de constructeur par recopie (que l'on peut justement vouloir interdire par ailleurs à cause du temps de copie)

Singleton

- On veut ne créer qu'une et une seule instance d'une classe
  - Besoin de coordonner des opérations
  - Efficacité
- Principe
  - Le premier appel crée une instance
  - Les suivants retourne un pointeur vers cette instance

Singleton

- template<typename T> class Singleton {
  public:
  static T& Instance() {
  // suppose que T a un constructeur par défaut
  static T theSingleInstance;
  return theSingleInstance;
  }
  };

- class OnlyOne : public Singleton<OnlyOne> {
  // constructeurs/destructeur de OnlyOne accessibles au Singleton
  friend class Singleton<OnlyOne>;
  //...définir ici le reste de l'interface
  };

Design Patterns

- Structural Patterns
  - Adapter or Wrapper
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight (poids-plume)
  - Proxy

Adaptor

- Permet de convertir une interface en une autre interface
  - Résout les problèmes d'incompatibilité d'interfaces
  - Permet l'intégration de classes non modifiables
  - Permet de fédérer du code : cas des portages

- Exemple
  - Une API sous Unix (ex: POSIX)
  - Une API sous Windows (Multithread)
  - Je veux une API commune afin de ne pas être géner dans le reste de mon code avec cette dépendance du système :
    - pour Windows je fait un wrapper autour de l'API Windows
    - pour Unix je fait un wrapper autour de l'API Unix
    - Les deux wrappers ont la meme API : la dépendance a disparu

Bridge

- But découpler l'interface d'une classe et son implémentation
  - Permet de faire varier les deux indépendamment
  - Permet de cacher la structure de l'implémentation

- Exemple
  - La plupart des structures de données
  - Une pile : peut-être implémentée par un tableau ou par une liste. En externe on ne doit pas le savoir. La structure interne peut-être cachée, on doit pouvoir la changer sans impacter l'API
- Parfois appelé Handle

Bridge

- class MyClass {
  MyClassI* _impl;
  int premier(){return _impl→premier();}
  int dernier(){return _impl→dernier();}
  }

- Class MyClassI {
  // ici les données membres
  int premier(){ // ici le code de la fonction
  }
  int dernier(){ // ici le code de la fonction
  }
  }

Composite

- But essayer de fédérer les traitements de plusieurs objets qui sont presque semblables
  - Augmenter le code commun
  - Réduire le code répétitif

- On a l'impression d'utiliser de multiples objets de la même façon avec des lignes de codes identiques ou presque

Composite

- Exemple
  - On a des adultes
  - On a des enfants
  - Chacun a des traitements particuliers
  - On va créer une classe Personne et travailler avec autant que possible. Seuls certains traitements seront spécialisés

- L'idée ici est de penser COMPOSANT d'un GROUPE. On va travailler avec le GROUPE

Decorator

- But : attacher dynamiquement de nouvelles caractéristiques à un objet à **runtime,** à condition que certains travaux aient été fait à compile-time
- Alternative au sous-classement qui ne peut être fait qu'à compile-time
- L'extension des fonctionnalités est appelée décoration et est réalisée par une class Décorator qui enrobe la classe originale.

Decorator

- On procède comme suit. Soit Component une classe que l'on veut pouvoir enrichir à runtime
  - On sous-classe Component par une classe Decorator
  - On ajoute dans Decorator un pointeur vers la classe Component
  - Le constructeur de Decorator accepte un objet Component ce qui permet d'affecter le pointeur
  - Dans la classe Decorator toutes les méthodes héritées de Component sont redirigées vers le pointeurs
    - La fonction func() de component devient func(){ptr→func();} dans le Decorator
  - Dans la classe Decorator on surcharge toutes les fonctions de Component dont on veut modifier le comportement
- On définit alors plusieurs Decorator. A runtime on pourra déterminer lesquels utiliser

Decorator

- En C# // Déclarations

```
abstract class Voiture {
        public abstract double Prix { get; }
}
class AstonMartin : Voiture {
        public override double Prix { get { return 999.99; } }
}
```

- // Décorateur

```
class Option : Voiture {
        protected Voiture _originale;
        protected double _tarifOption;
        public Option(Voiture originale, double tarif) {
                _originale = originale;
                _tarifOption = tarif;
        }
        public override double Prix { get { return _originale.Prix + _tarifOption; } }
}
```

Decorator

```
class Climatisation : Option {
        public Climatisation (Voiture originale) : base(originale, 1.0) { }
}
class Parachute : Option {
        public Parachute (Voiture originale) : base(originale, 10.0) { }
}
class Amphibie : Option {
        public Amphibie (Voiture originale) : base(originale, 100.0) { } }
```

```
// Implémentation
class Program {
        static void Main() {
                Voiture astonMartin= new AstonMartin();
                astonMartin = new Climatisation(astonMartin);
                astonMartin = new Parachute(astonMartin);
                astonMartin = new Amphibie(astonMartin);

                Console.WriteLine(astonMartin.Prix); // affiche 1110.99
        }
}
```

Facade

- Une facade est un objet qui fournit une interface simplifiée à une grande partie de code comme une librairie
- Une façade
  - Rend l'utilisation d'une librairie plus simple à utiliser et à comprendre en introduisant des méthodes pour les tâches les plus courantes
  - Rend l'utilisation d'une libraire indépendante du reste du code

Facade

- Utilisation de parties complexes comme des fonctions hardware sur le CPU ou la mémoire
- /* Complex parts */

```
class CPU {
        public void freeze() { ... }
        public void jump(long position) { ... }
        public void execute() { ... }
}
class Memory {
        public void load(long position, byte[] data) { ... }
}
class HardDrive {
        public byte[] read(long lba, int size) { ... }
}
```

Facade

```
/* Facade */
class Computer {
        public void startComputer() {
                cpu.freeze();
                memory.load(BOOT_ADDRESS,
                        hardDrive.read(BOOT_SECTOR, SECTOR_SIZE));
                cpu.jump(BOOT_ADDRESS);
                cpu.execute();
        }
}
```

```
/* Client */
class You {
        public static void main(String[] args) {
                Computer facade = new Computer();
                facade.startComputer();
        }
}
```

Flyweight (poids plume)

- But: faire de l'objet même pour les très petites choses, comme des lettres, des icones qui reviennent tout le temps ... Sans utiliser trop de mémoire.
- Un flyweight est un objet qui minimise la mémoire utilisé en partageant autant que possible les données avec des objets similaires
- Souvent certaines parties de l'état d'un objet peuvent être partagées et il est commun de les mettre dans des structures de données externes et de les passer, quand ils sont utilisés, à des objets "poids plumes" temporaires

Flyweight (poids plume)

- Exemple classique
- Représentation graphique de caractères
- Il est utile d'avoir pour chaque caractère d'un document un objet qui contient diverses informations sur le caractères
  - Sa police
  - Sa taille
  - Sa mise en évidence (en gras, italique)
- Il est clairement impossible d'avoir de façon indépendante cette information pour chaque caractère
- Solution
  - Pour chaque caractère on a un pointeur vers un objet flyweight qui est partagé par tous les caractères ayant les mêmes propriétés

Proxy

- But : contrôler l'accès à un objet
- C'est une classe fonctionnant comme une interface vers autre chose :
  - une connexion réseau,
  - un large objet en mémoire,
  - un fichier
  - une ressource qu'il est impossible de dupliquer
- Cela permet de faire croire qu'il y a des objets distincts alors que cela n'est pas le cas.
  - On utilisera normalement un objet alors qu'il est en fait manipulé à distance via un réseau
- L'utilisateur n'a pas à gérer ce que fait le proxy, il peut utiliser le proxy comme s'il était l'objet et c'est le proxy qui gère l'indirection

Proxy

- Exemple typique de proxy
  - le pointeur avec compteur de référence

- Par exemple on peut coupler des proxys avec le flyweight pattern
  - On crée une instance d'un gros objet
  - On crée de multiple instance du proxy vers cet objet
  - Toutes les opérations demandées au proxy sont transférées vers l'objet original
  - Le projet gère alors les références vers l'objet et peut le faire disparaitre lorsqu'il devient inutile

Design Pattern

- Behavioral
  - Chain of responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method (patron de méthode)
  - Visitor

Chain of responsibility

- Pattern qui consiste en une source d'objet commande et en une chaine d'objets de traitements
  - Chaque objet de traitement contient un ensemble de règles logiques
    - qui décrivent les types d'objet de commandes qu'il peut traiter
    - qui décrivent comment passer ceux qu'ils ne peut pas traiter au suivant de la chaîne
- Un mécanisme existe aussi pour ajouter des éléments en fin de la chaîne

Command

- But : gérer l'invocation en séparant le code initiateur de l'action du code de l'action elle-même
  - Gère la délégation
  - Permet l'exécution différée sans avoir besoin de tout connaitre lors de la compilation

- Défini une classe qui est utilisée pour représenter et encapsuler toutes les informations requise pour appeler une méthode plus tard
  - Le nom de la méthode
  - L'objet qui possède la méthode
  - Les valeurs des paramètres de la méthode

Command

- 3 termes sont souvent associés avec ce pattern
  - Client : instantie la commande et fourni les informations pour appeler la méthode plus tard
  - Invoker : décide quand la méthode doit être appelée
  - Receiver : instance de la classe qui contient la méthode, objet sur lequel la méthode est invoquée

Command

- /*the Command interface*/
```
public interface Command {
        void execute();
}
```
- /* ma fonction */
```
public Func implements Command {
        MyObj _obj;
        int _para1;
        OtherObj _para2;
        // ctor …
        void execute(){
                _obj→afonction(_para1,_para2);
        }
}
```
- /* la classe MyObj */

  …
- /* l'appelant */
```
Command c=new Func(obj,p1,p2);
c → execute();
```

Interpreter

- L'interpreter spécifie comment évaluer des séquences dans un langage
  - Définit la grammaire du langage
  - L'utilise pour interpréter des états du langage
  - The interpreter pattern specifies how to evaluate sentences in a language. The basic idea is to have a class for each symbol (terminal or nonterminal) in a specialized computer language. The syntax tree of a sentence in the language is an instance of the composite pattern and is used to evaluate (interpret) the sentence

Interpreter

```
interface Expression {
        public int interpret(X variables);
}
class Number implements Expression {
        private int number;
        public Number(int number) { this.number = number; }
        public int interpret(X variables) { return number; }
}
class BinaryOp {
        Expression leftOp;
        Expression rightOp;
        public BinaryOp(Expression left, Expression right) { leftOp = left; rightOp = right; }
}
class Plus extends BinaryOp implements Expression {
        public int interpret(X variables) { return leftOp.interpret(variables) + rightOp.interpret(variables); }
}
class Minus extends BinaryOp implements Expression {
        interpret(X variables) { return leftOp.interpret(variables) - rightOp.interpret(variables); }
}
class Variable implements Expression {
        private String name;
        public Variable(String name) { this.name = name; }
        public int interpret(X variables) { return variables.get(name); }
}
```

Interpreter

```
class Evaluator {
    private Expression syntaxTree;
    public Evaluator(String expression) {
        Stack<Expression> eStack = new Stack<Expression>();
        for (String token : expr.split(" ")) {
                if (token.equals("+")) {
                    Expression subExpr = new Plus(eStack.pop(),eStack.pop());
                    eStack.push( subExpr);
                }
                if (token.equals("-")) {
                    Expression subExpr = new Minus(eStack.pop(), eStack.pop());
                    eStack.push( subExpr );
                }
                default : eStack.push( new Variable(token) );
        }
        syntaxTree = eStack.pop();
    }
    public int evaluate(X context) {
        return syntaxTree.interpret(context);
    }
}
```

Iterator

- Un iterator est un objet qui permet de parcourir les éléments contenus dans un autre objet, le plus souvent un conteneur (liste, arbre, pile, ensemble …)

- Ressemble à un pointeur disposant de deux primitives
  - Accéder à l'élément pointé en cours
  - Se déplacer pour pointer vers le suivant

Iterator

- Permet de définir des algorithmes génériques sur des structures de données sans avoir besoin de faire des versions pour chaque type de données

- Exemple :
  - Plus court chemin dans un graphe
  - Même algorithme et même code si le graphe est représenté par
    - Liste des voisins
    - Matrice d'adjacence

Iterator

- La STL (Standard Template Library) fait grand usage des iterator

- Attention, à la modification (suppression /ajout) du container pendant le balayage
  - Certains iterator impose un maintien constant du container pendant le balayage

Iterator

```
public class BitSetIterator implements Iterator<Boolean> {
        private final BitSet bitset;
        private int index;
        public BitSetIterator(BitSet bitset) { this.bitset = bitset; }
        public boolean hasNext() {
                return index < bitset.length();
        }
        public Boolean next() {
                boolean b = bitset.get(index++);
                return new Boolean(b);
        }
        public void remove() { … }
}
```

Mediator

- Le mediator fournit une interface unifiée pour un ensemble d'interfaces
  - Réduire les dépendances entre les classes
  - Simplifier la communication entre les classes

- Problème de communication entre classes
  - Beaucoup de classes : traitement de données répartis entre les classes
  - Problème de compréhension et de lisibilité

- Mediator : on crée une classe qui sert à faire la communication avec toutes les classes
  - Lorsqu'une classe veut interagir avec une autre, elle passe par le médiator qui transmettra l'information

- Utilisé lors de la maintenance ou le refactoring de code
- Permet de définir des protocoles propres
- Réduit le coupling (couplage : échange d'information entre les classes)

Memento

- Le memento fournit la manière de renvoyer un objet à un état précédent (undo via rollback)
- Utilisé par 2 objets
  - L'**originator** (le créateur) : objet ayant un état interne
  - Le **caretaker** (le gardien) : celui qui modifie le créateur mais qui veut pouvoir défaire les modifications
- Principe
  - Le caretaker demande à l'originator un objet memento
  - Le caretaker fait la modification de l'originator
- Pour restaurer l'état précédent on utilise le memento
- Notons que le caretaker ne doit pas toucher au memento

Memento

- On va donc faire une classe MyObjMemento qui est une classe interne
- MyObj aura deux fonctions
  - saveToMemento qui retourne l'objet MyObjMemento
  - restoreFromMemento qui prend en paramètre un MyObjMemento et qui restore l'objet courant

- Peut être difficile à mettre en œuvre, surtout si plusieurs modifications
- Difficile de retourner directement à un état sans passer par les sauvegarde intermédiaire
  - Aspect ayant souvent peu d'intérêt donc il vaut lieux penser en terme de pile

Observer

- L'observer est un pattern dans lequel un objet (le sujet)
  - maintient la liste de ses dépendants (les observers)
  - notifie automatiquement les observers après chaque modification du sujet en appelant une de leur méthode (ou en passant par des commandes)

- Principalement utilisé pour mettre en oeuvre la programmation événementielle
- Appelé aussi Publisher/Subscriber

Observer

- Mise en œuvre
  - Système d'abonnement : le subscriber s'abonne au publisher et définit la fonction que le publisher devra appeler quand un événement se produit
  - Le publisher gère la liste des abonnés
  - Quand le publisher est modifié, alors il notifie tous les subscriber abonné en appelant la fonction associée

- Quadruplet (Publisher, Event, Subscriber, Command)
  - Fonction notify : quand le publisher subit l'événement Event alors il appelle la commande Command pour le subscriber

State

- State est utilisé pour représenté l'état d'un objet afin de modifier son comportement sans pour autant modifier l'instance

- Deux classes
  - State : définit l'abstraction du comportement d'un objet
  - Context : interface entre cet objet et le reste de l'application

- Gère des automates à états simple

State

- Idée principale
  - La classe State définit des actions qui sont dépendantes d'un contexte (l'état particulier dans lequel on se trouve)
    - Les méthodes prennent un contexte en paramètre
    - Les méthodes peuvent modifier le contexte
  - L'action demandée à State est donc effectuée par le contexte : indirection
    - State::Func(Context Ctx){Ctx→Func();}
    - Eventuellement Ctx→Func(); et modification de Ctx

State

```
interface State {
        public void writeName(StateContext sContext, String name);
}
class StateA implements State {
        public void writeName(StateContext sContext, String name) {
                System.out.println(name.toLowerCase());
                sContext.setState(new StateB());
        }
}
class StateB implements State {
        private int count=0;
        public void writeName(StateContext sContext, String name){
                System.out.println(name.toUpperCase());
                if(++count>1) { sContext.setState(new StateA()); }
        }
}
```

State

```
public class StateContext {
        private State myState;
        public StateContext() { setState(new StateA()); }
        // normally only called by classes implementing the State interface public
void setState(State stateName) {
                this.myState = stateName;
        }
        public void writeName(String name) {
                this.myState.writeName(this, name);
        }
}
```

```
public class TestClientState {
        public static void main(String[] args) {
                StateContext sc = new StateContext();
                sc.writeName("Monday");
                sc.writeName("Tuesday");
                sc.writeName("Wednesday");
                sc.writeName("Thursday");
                sc.writeName("Saturday");
                sc.writeName("Sunday");
        }
}
```

Strategy

- Strategy permet de changer l'algorithme sélectionné à runtime
- Cela permet à chaque objet de décider quel algorithme va être utilisé

- Exemple :
  - Minimum d'un container
    - Pas trié : on balaie
    - Trié : on retourne le premier
  - Je veux un container générique

Strategy

- Solution
  - Utilisation de test ou de switch : si mon tableau est trié alors algo1, sinon algo2
    - Problèmes : répétition de test/switchs pour toutes les fonctions (maximum, 2ndmin etc…)

- Principe :
  - On crée une classe Strategy racine interne qui a autant de classes dérivées que d'algorithmes possibles
  - La classe externe contient un pointeur sur la classe Strategy racine interne
  - On va ensuite déterminer à runtime quelle instance sera effectivement affecté au pointeur

Strategy

```
class StrategyInterface {
        public: virtual void execute() const = 0;
};
class ConcreteStrategyA: public StrategyInterface {
        public: virtual void execute() const {
                cout << "ConcreteStrategyA execute method" << endl;
        }
};
class ConcreteStrategyB: public StrategyInterface {
        public: virtual void execute() const {
                cout << "ConcreteStrategyB execute method" << endl;
        }
};
class ConcreteStrategyC: public StrategyInterface {
        public: virtual void execute() const {
                cout << "ConcreteStrategyC execute method" << endl;
        }
};
```

Strategy

```
class Context {
        private: StrategyInterface * _strategy;
        public: explicit Context(StrategyInterface* strategy): _strategy(strategy) { }
        void set_strategy(StrategyInterface *strategy) { _strategy = strategy; }
        void execute() const { _strategy→execute(); }
};
```

```
int main(int argc, char *argv[]) {
        ConcreteStrategyA cStrategyA;
        ConcreteStrategyB cStrategyB;
        ConcreteStrategyC cStrategyC;
        Context context(&cStrategyA);
        context.execute();
        context.set_strategy(&cStrategyB);
        context.execute();
        context.set_strategy(&cStrategyC);
        context.execute();
        return 0;
}
```

Template Method

- Définit une méthode non abstraite (Template method) dans une classe abstraite et fait appel à des méthodes abstraites pour fonctionner
- C'est la classe parent qui appelle les opérations n'existant que dans les sous-classes
  - Un peu inhabituel, normalement : les sous-classes concrètes appellent les méthodes de la super-classe.
- N'est pas lié aux templates du C++ ou Java

Template Method

- Parfois appelé méthode socle parce qu'elle ancre solidement un comportement qui s'applique à toute la hiérarchie de classe
- Fortement relié au pattern NVI (Non-Virtual Interface).
- Le pattern NVI reconnait le bénéfice d'une méthode non abstraite appelant des méthodes abstraites, ce qui permet de modifier facilement le code interne (on modifie les pré et les post traiements avant les appels au code abstrait).

Template Method

```
abstract class Game {
        protected int playersCount;
        abstract void initializeGame();
        abstract void makePlay(int player);
        abstract boolean endOfGame();
        abstract void printWinner();

        /* A template method : */
        final void playOneGame(int playersCount) {
                this.playersCount = playersCount;
                initializeGame();
                int j = 0;
                while (!endOfGame()) {
                        makePlay(j); j = (j + 1) % playersCount;
                } printWinner();
        }
}
```

Template Method

```
//Now we can extend this class in order to implement actual games: class Monopoly extends
Game {
        /* Implementation of necessary concrete methods */
        void initializeGame() { // Initialize money }
        void makePlay(int player) { // Process one turn of player }
        boolean endOfGame() { // Return true of game is over according to Monopoly rules }
        void printWinner() { // Display who won }
        /* Specific declarations for the Monopoly game. */
        // ...
}
class Chess extends Game {
        /* Implementation of necessary concrete methods */
        void initializeGame() { // Put the pieces on the board }
        void makePlay(int player) { // Process a turn for the player }
        boolean endOfGame() {
                // Return true if in Checkmate or Stalemate has been reached }
        void printWinner() { // Display the winning player }
        /* Specific declarations for the chess game. */
        // ...
}
```

Visitor

- Le pattern visitor (visiteur) est un manière de séparer un algorithme de la structure de l'objet sur lequel il opère
- Permet l'ajout de nouvelles opérations sur un objet sans modifier la structure de l'objet (la classe de l'objet)
- Limite les modifications intrusives
- Conforme au principe ouvert/fermé
  - **open/closed principle** "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*"

Visitor

- Principe :
  - On ajoute à chaque classe que l'on veut pouvoir étendre une méthode virtuelle (méthode callback) accept qui prend en paramètre un Visitor : MyObj::accept(Visitor* v)
  - On définit un visitor : c'est-à-dire une sous-classe de Visitor
  - On ajoute dans Visitor une fonction virtuelle visit qui prend en paramètre l'objet que l'on veut étendre : visit(MyObj* obj)
  - On surcharge cette fonction dans le visiteur que l'on a créé
  - On définit le code de la fonction accept de MyObj pour que la fonction visit soit appelée

```
MyObj::accept(Visitor* v){
    v→visit(this)
}
```

Visitor

- class Visitor {
  virtual void visit(MyObjA* obj);
  virtual void visit(MyObjB* obj);
  }
- Class MyVisitor1 : public Visitor {
  void visit(MyObjA* obj){
  // traitement pour obj de type MyObjA
  }
  void visit(MyObjB* obj){
  // traitement pour obj de type MyObjB
  }
- On a
  - MyObjA::accept(Visitor* v){v→visit(this);}
  - MyObjB::accept(Visitor* v){v→visit(this);}
- On a un ensemble d'objet O, on veut faire les traitements de MyVisitor1 pour tous les objets :
  MyVisitor1 v;
  for(int i=0;i<n;i++){
  O[i]→accept(v);
  }

Visitor

- On veut définir un nouveau comportement
- On crée une classe NvComportement qui hérite de Visitor

```
Class NvComportement : public Visitor {
        void visit(MyObjA* obj){
                // traitement pour obj de type MyObjA
        }
        void visit(MyObjB* obj){
                // traitement pour obj de type MyObjB
        }
```

- On a un ensemble d'objet O, on veut faire les traitements de NvComportement pour tous les objets :

```
        NvComportement v;
        for(int i=0;i<n;i++){
                O[i]→accept(v);
        }
```

Design Patterns

- Autres patterns :
- Creational
  - Lazy Initialization
  - Object Pool
- Concurrency patterns
  - Monitor

Lazy Initialization

- Pattern permettant de retarder l'initialisation d'un objet
- Utile si l'initialisation est couteuse
- Idée : éviter de faire du travail systématique pour rien
- Mise en œuvre
  - Simplement en utilisant un flag qui détermine si l'objet est à jour ou pas

Lazy Initialization

- Se généralise pour les mises à jour
- On utilise un stamp/timer (horloge) pour déterminer si l'objet est à jour ou pas.
  - Compteur correspondant au moment où certains événements se produisent
  - Compteur global et local (lié à l'objet). On compare si égalité. Si pas égalité
    - on met à jour
    - on affecte le compteur local avec le global
- Exemple :
  - De temps en temps on a besoin qu'une liste soit triée
    - On maintient la liste triée. Peut complexifier le code
    - On met à jour uniquement lorsqu'on en a besoin

Object Pool

- Un object pool est un ensemble d'objets non initialisés qui sont prêts à être utilisés

- Evite une allocation/Désallocation permanente d'objets
  - Au lieu de d'allouer les objets on les demande à l'object pool
  - Au lieu de désallouer des objets on les rends à l'object pool

- C'est un type particulier de pattern factory
- Le pooling d'objets peut offrir une amélioration sensible des performances.
- Attention dans un environnement multithreadé de ne pas bloquer les objets en attente

Monitor

- Approche pour synchroniser plusieurs tâches utilisant de façon concurrente plusieurs ressources

- Le principe est de regrouper dans une même classe des méthodes sur des objets qui doivent être en exclusion mutuelle (un seul thread à la fois)
  - La synchronisation un peu partout dans le code est très difficile à gérer
  - On définit les données qui doivent être partagées par des objets et on les place dans une classe particulière : le Monitor
  - Le Monitor gère les accès concurrentiels à l'aide d'outils de synchronisation : lock, unlock, sémaphore …
  - Toutes les fonctions du monitor sont thread-safe
  - En général, le Monitor est en accès exclusif

Anti-patterns

- Les anti-patterns sont des erreurs courantes de conception des logiciels.
- Leur nom vient de l'absence ou de la mauvaise utilisation de design patterns
- Les anti-patterns se caractérisent souvent par
  - une lenteur excessive du logiciel,
  - des coûts de réalisation ou de maintenance élevés,
  - des comportements anormaux
  - la présence de bugs

Anti-patterns

- Pas vraiment formalisé
- Valables dans beaucoup de domaines de la gestion de projet
  - Organisation
  - Management
  - Conception
  - Programmation

Anti-patterns

- Certains sont marrants
  - **Management Champignon :** garder ses employés non informés et mal-informés (les garder dans le noir et les nourrir avec du crottin)

Anti-patterns

- **Abstraction inverse :**
  - interface qui n'offrent pas les fonctions nécessaires à l'utilisateur alors qu'elle pourrait. L'utilisateur doit utiliser des fonctions complexes pour faire des choses simples
- **Usine à gaz** (Gaz factory):
  - Système d'une complexité non nécessaire qui fuit de partout
- **Interface énorme** (Interface bloat) :
  - Interface tellement énorme qu'on arrive plus à l'implémenter
- **Bouton magic** (Magic pushbutton) :
  - Coder l'implémentation directement à l'intérieur du code de l'interface sans aucune abstraction
- **Situation de compétition** (Race hazard) :
  - Ne pas tenir compte de différents ordre possibles de déclenchement des événements.

Anti-patterns

- **Objet divin** (God object):
  - Concentrer trop de fonctions et de données dans une seule classe (contraire de diviser pour régner)
- **Fantôme** (Poltergeists):
  - Objets dont le seul rôle est de passer de l'information à un autre objet.
- **Couplage séquentiel** (Sequential coupling):
  - Classe qui requière que ses méthodes soient appelées dans un ordre particulier
- **Problème du yoyo** (Yo-yo problem):
  - Graphe d'héritage tellement long et compliqué que le programmeur doit passer en permanence d'une classe à l'autre pour comprendre la structure du programme
- **Action à distance** (Action at distance):
  - Emploi immodéré de variables globales ou de dépendances entre objets

Anti-patterns

- **Croyance aveugle** (Blind faith):
  - Prétendre avoir corriger un bug sans faire un test pour le vérifier
- **Ancre de bateau** (Boat anchor):
  - Garder un composant inutilisé en pensant que cela pourra resservir plus tard
- **Programmation religieuse** (Cargo cult programming):
  - Utiliser des patterns ou des méthodes sans les comprendre
- **Codage matériel** (Hard code):
  - Programmer en faisant des suppositions non explicités sur la machine ou l'OS utilisé
- **Coulée de lave** (Lava flow):
  - Mettre en production une partie du code immature forçant la lave à se solidifier et en empêchant ainsi sa modification

Anti-patterns

- **Nombre magique** (Magic numbers):
  - Introduire des nombre inexpliqué dans des algorithmes (en pensant que jamais une donnée n'aura cette valeur)
  - Valable aussi avec des chaînes
- **Programmation spaghetti** (Spaghetti code):
  - Faire un code ressemblant à un plat de spaghetti : il est impossible de modifier une petite partie du code sans altérer le fonctionnement de tous les autres composants
- **Programmation par couper-coller** (Copy and paste programming):
  - Programmer en procédant principalement par couper-coller et modification au lieu de développer du code générique
- **Marteau d'or** (Golden hammer):
  - Supposer que la solution favorite résout tous les problèmes
- **Facteur d'impossibilité** (Improbability factor):
  - Programmer en supposant qu'il est improbable qu'une erreur arrive

Anti-patterns

- **Optimisation prématurée** (Premature optimization):
  - Programmer en optimisant en permanence des bouts de code au lieu de s'intérer à la structuration, la maintenance et le design correcte (et parfois même l'efficacité réelle)
- **Programmer par Accident** (Programming by accident):
  - Programmer en corrigeant les bugs en se contentant de modifier le code au hasard jusqu'à ce que cela marche et non en réfléchissant aux problèmes
- **Réinventer la roue** (Reinventing the wheel):
  - Préférer mal réinventer des solutions existantes plutôt que d'utiliser l'existant