Développement informatique

Cours 4 :

- Introduction aux « Best practices »
- Introduction à la gestion des versions

« Best practices »

- Années 90 : Recherches intensives sur les patterns pour les interfaces utilisateur par Gamma, Helm, Johnson et Vlissides

→ publication en 1994 d'un livre « Design Patterns : Elements of reusable object oriented software »

→ Définition des patterns « Gof » (gang of four)

- 2005 : Définition de patterns plus intuitifs par Craig Larman : les patterns « GRASP » dont on peut trouver de nombreuses classifications dans la littérature

« Best practices »

Qu'est ce qu'un Pattern (patron de conception)?

→ Ce n'est pas un algorithme, mais une structure générique qui permet de résoudre le problème référencé

→ Il est indépendant du langage de programmation

→ il est « standardisé », suffisamment pour que tous puissent s'y référer.

→ C'est l'expérience qui montre que la solution est valide.

« Best practices » : GRASP

GRASP : acronyme de General Responsability Assignement Software Pattern

→ Pattern fondamentaux qu'il faut appliquer pour une bonne conception, issus en général du bon sens des développeurs/concepteurs (to grasp signifie aussi saisir, comprendre en anglais)

Principe de base : recherche du meilleur objet responsable pour une action.

« Best practices » : GRASP

On défini 9 patterns :

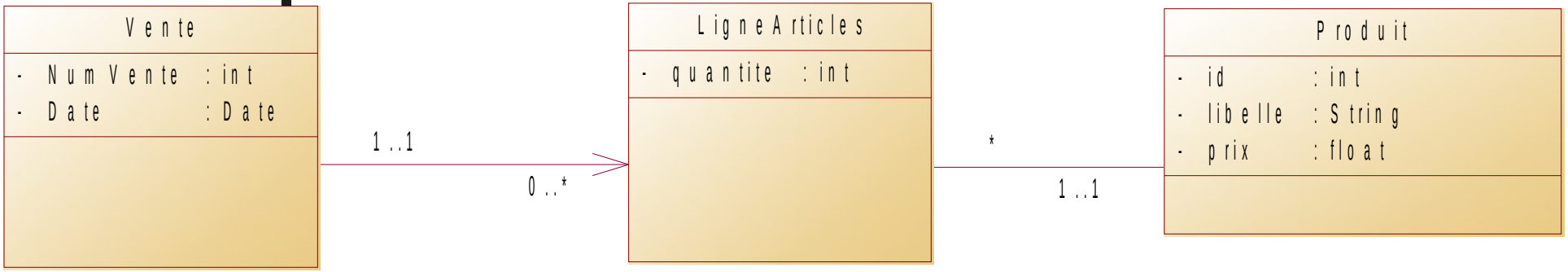
Expert , Faible couplage , Forte cohésion, Création, Contrôleur, Polymorphisme, Fabrication pure, Indirection, Protection des variations

« Best practices » : GRASP

1-Pattern Expert (Information Expert)

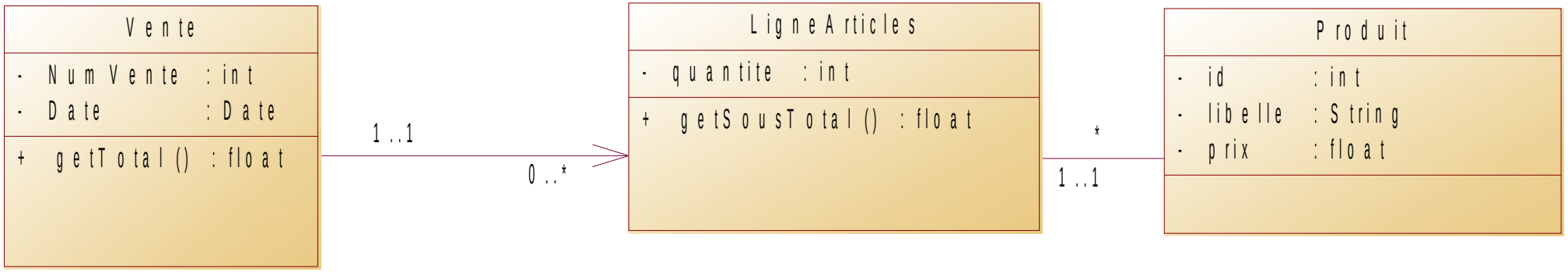
→ Affecter la responsabilité à la classe qui détient l'information.

L'objet chargé de la responsabilité d'une action est celui qui détient l'information concernée !



Si on veut obtenir le total général concernant une vente, comment s'y prendre ? Qui est responsable de ce calcul ?

« Best practices » : GRASP



C'est la classe Vente, qui offrira une méthode getTotal().. Mais comment cette méthode pourra-t-elle faire le calcul ?

« Best practices » : GRASP

2 Pattern Créateur (Creator)

→ La responsabilité de créer une classe incombe à la classe qui agrège, contient, enregistre, utilise étroitement ou dispose des données d'initialisation de la classe à créer.

Qui doit créer un objet A ? Qui en a la responsabilité ?

→ Décider qui doit être créateur en ce basant sur l'association entre objet et leurs interactions

« Best practices » : GRASP

Un objet de la classe B aura la responsabilité de créer des objets de la classe A si :

- **Il contient ou agrège des objets A**
- **Il possède les informations pour initialiser A**
- **Il utilise étroitement des objets A**

Dans l'exemple précédent, Vente est une bonne candidate pour instancier les LignesArticles...

« Best practices » : GRASP

3 Faible couplage

→ Affecter les responsabilités de sorte que le couplage (entre Classe) reste faible. (Réduire l'impact des modifications, limiter les interactions)

Il faut essayer d'affecter les responsabilités au plus près. Il faut avoir accès le plus directement possible à la bonne méthode, sans passer par trop d'instances. Il y a fort couplage si il y a beaucoup d'objets en jeu pour obtenir une information. Ce pattern est lié à Expert qui nous pousse à trouver l'objet qui dispose des informations, afin de lui affecter une responsabilité.

« Best practices » : GRASP

En général ce pattern est appliqué inconsciemment, c'est davantage un principe d'appréciation qu'une règle. Il faut essayer de trouver le juste équilibre et le rapport optimum avec les patterns Expert en Information, Forte Cohésion afin d'obtenir des classes :

- **Cohésives**
- **Réutilisables**
- **Compréhensibles**
- **Maintenables**
- **Indépendantes de la modification d'autres composants**

« Best practices » : GRASP

4 Forte cohésion

→ Affecter les responsabilités de sorte que la cohésion (dans la Classe) demeure forte

La cohésion est une notion assez vague dont l'objectif est de déterminer si l'objet n'en fait pas « trop »

« Best practices » : GRASP

Un objet avec 100 méthodes, des milliers de lignes de codes, couvrant divers domaines n'est plus cohérent → perte de cohésion, risque de fort couplage !

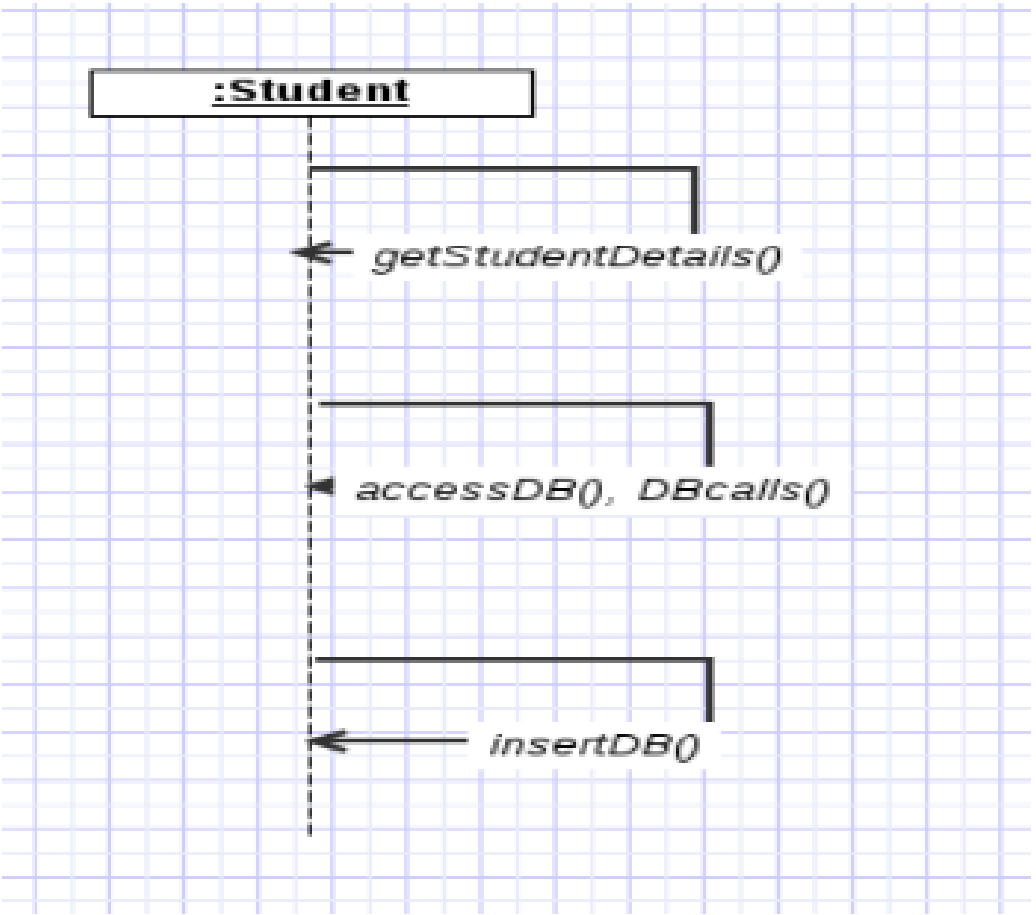
solution : Il vaut mieux déléguer, instituer une collaboration avec de nouveaux objets

- Compréhension et maintenance du code facilité

- Réutilisation du code

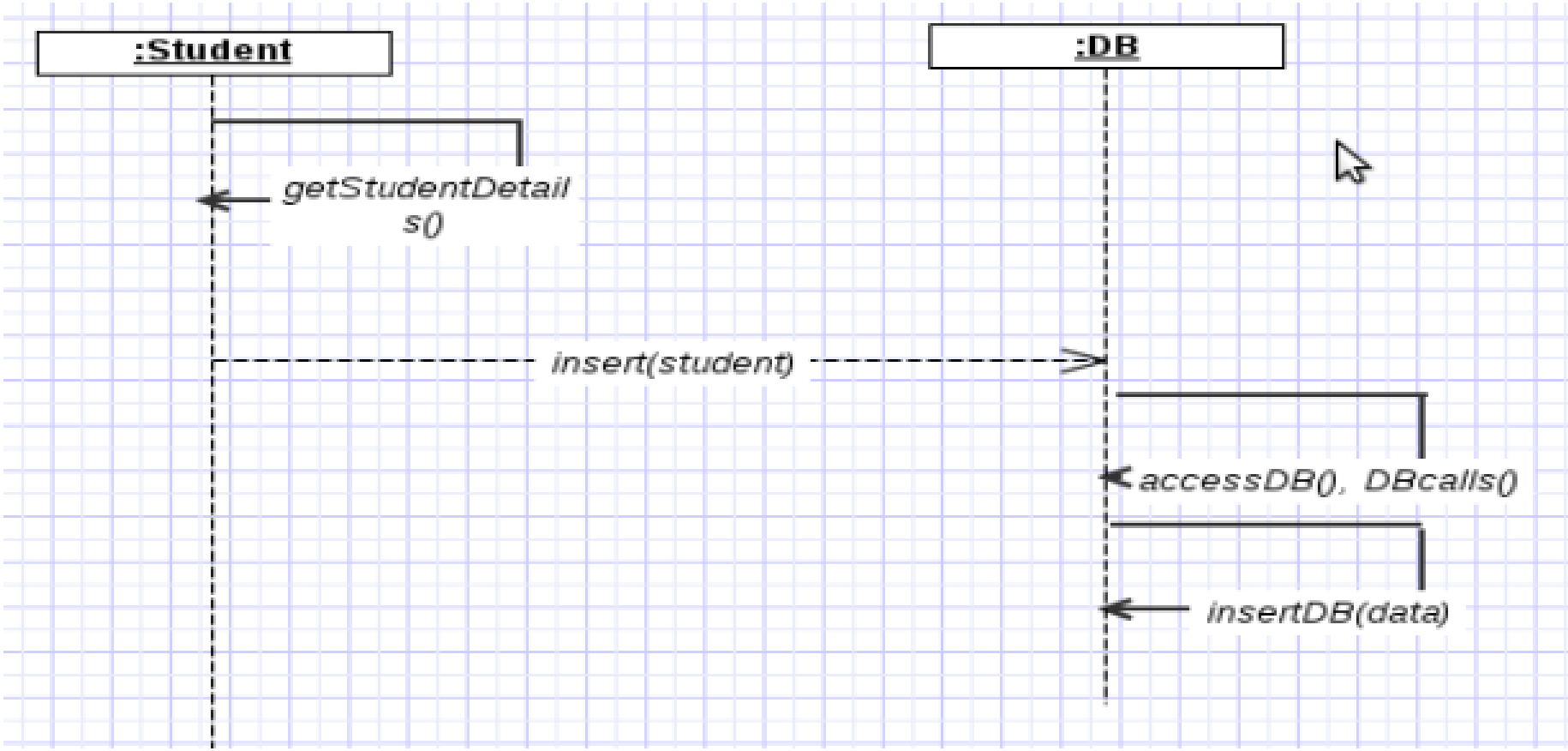
« Best practices » : GRASP

Exemple de faible cohésion



« Best practices » : GRASP

Exemple de forte cohésion



« Best practices » : GRASP

5 Contrôleur (Controller)

→ Affecter la responsabilité à une classe « façade » représentant l'ensemble d'un système ou un scénario de cas d'utilisation.

Il faut gérer l'ensemble des actions, coordonner les différents éléments. Il faut une classe Contrôleur, qui déléguera ensuite aux divers objets spécialisés, et récupérera les résultats de leurs actions.

« Best practices » : GRASP

Le pattern Contrôleur consiste en l'affectation de la responsabilité de la réception et/ou du traitement d'un message système à une classe.

Il reçoit les demandes, et le redirige vers la bonne classe, celle qui en a la responsabilité.

« Best practices » : GRASP

Parfois, si il y a beaucoup de cas d'utilisation, ou beaucoup d'actions à coordonner, pour éviter de perdre le Pattern 'cohésion forte', on peut utiliser une classe Façade, qui utilisera des Contrôleurs qui géreront ensuite des cas spécialisés.

Ainsi, sur une grosse application, si l'ensemble des actions possibles devenait trop important, on pourrait imaginer une façade générale qui utiliserait les contrôleurs 'GestionClients', 'GestionCommandes' et 'GestionProduits'.

« Best practices » : GRASP

On parlera de classe « ballonnée » (bloated)

- Si elle gére trop de responsabilités (solution → ajouter des contrôleurs)

- Si elle exécute des taches au lieu de déléguer aux autres classes. (→ déléguer le travail a d'autres classes)

« Best practices » : GRASP

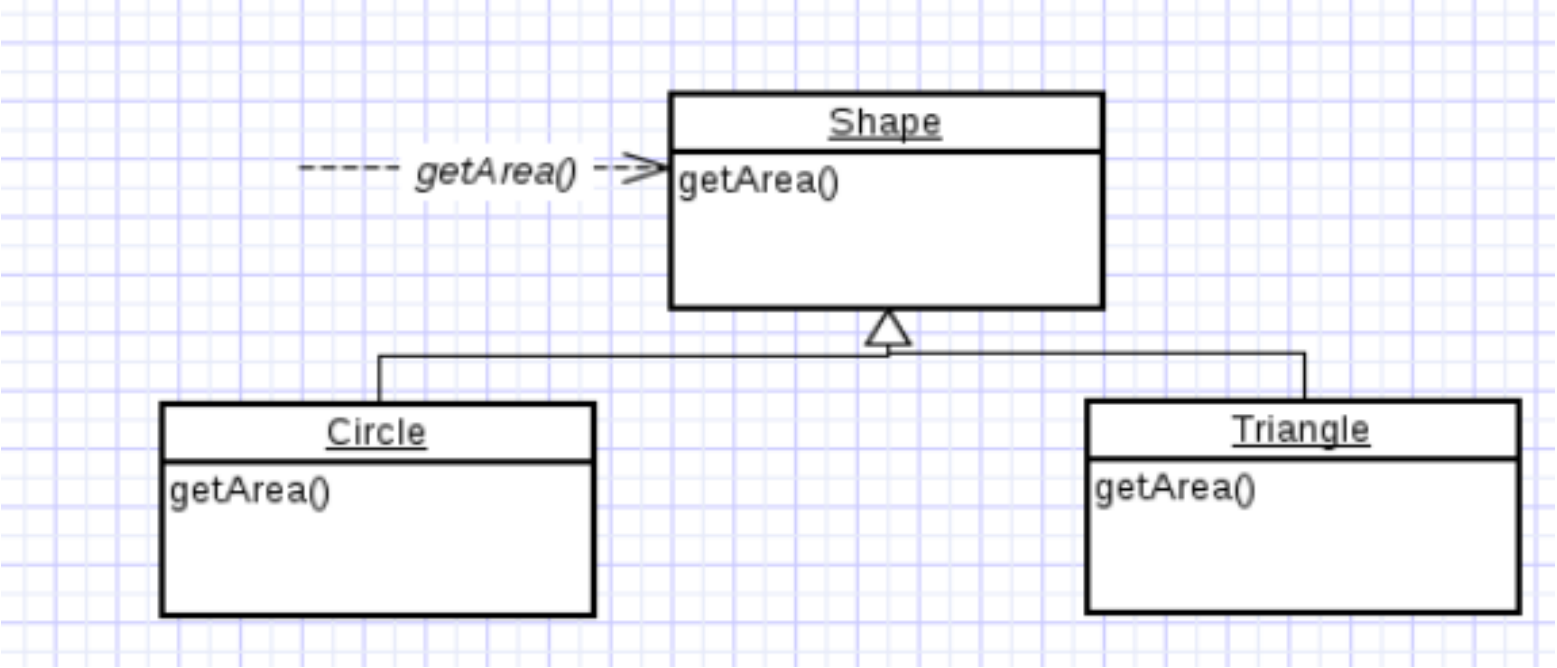
6 Polymorphisme

→ Utiliser le polymorphisme pour implémenter les variations de comportement en fonction de la classe.

Il permet également de réaliser des composants logiciels enfichables grâce aux patterns adaptateurs. Il est utile dans le cadre de prévisions de remplacement ou d'ajout de composants.

« Best practices » : GRASP

Exemple de
Polymorphisme



« Best practices » : GRASP

7 Fabrication Pure

→ Affecter un ensemble de responsabilités fortement cohésif à une classe artificielle ou de commodité qui ne représente pas un concept du modèle de domaine.

Ce pattern est utilisé lorsque les patterns ne parviennent pas à assurer les principes de forte cohésion et de faible couplage.Le concepteur doit inventer de nouveaux concepts abstraits sans relation directe avec les domaine. Ils sont plutôt génériques…

Par exemple, une classe chargée de la persistance (DBStorage, DAO...., Logger)

Ce n'est pas un objet métier, c'est un choix du concepteur pour résoudre SA problématique de stockage.

« Best practices » : GRASP

8 Indirection

→ Affecter des responsabilités à un objet qui sert d'intermédiaire entre d'autres composants ou services pour éviter de les coupler directement.

Lorsque certaines classes ont trop de responsabilités, la cohésion tend à baisser (pattern forte cohésion) et la classe devient plus difficile à comprendre et à maintenir. Une solution à ce type de problème est de déléguer une partie des traitements et des données à d'autres classes en ajoutant un niveau d'indirection.

« Best practices » : GRASP

9 Protection des variations

→ identifier les points de variations ou d'instabilité prévisibles. Affecter les responsabilités pour créer une interface stable autour d'eux.

L'application sera sujette à évolutions. Il faut la protéger des conséquences de ces modifications prévisibles. Le principe consiste à utiliser les mécanismes d'abstraction pour créer une interface stable autour des objet du système.

Il faut empêcher qu'une variation d'un sous-système n'impacte trop son environnement. On retrouve l'idée qui nous guide depuis le début : comment permettre l'évolution sans compromettre l'ensemble du logiciel ?

« Best practices » : un peu d’humour

Aside from "undefined behavior" gremlins (which is a very real concern) a big part of best practices in C is the same as best practices in any language.
* Hire people that know what the &%#* they are doing
* Use a source code version control system
* Have a clear, extremely detailed specification of what is to be implemented.
* Divide up the functionality into sensible modules. Decide on useful API's at every level.
* Because you divided things up and have nice API's you can easily write unit tests for anything and everything. Migrate the unit tests into becoming unit-level regression test over time.
* Use high quality pre-existing libraries and tools rather than roll your own.
* Make sure everyone understands the differences between ISO C vs. POSIX vs. WIN32.
* Use C when appropriate, but don't be afraid of shifting some tasks to/from other languages.
* Refactor as an ongoing task.
* Give programmers a well-thought out work environment that re-inforces that the company values good, clean design.

« Best practices » : un peu d’humour

Some additional steps I take for C programs:
* use valgrind once in a while and fix everything it points out.
* even if the code isn't multithreaded, make sure it will work as independent threads w/o interference
* test on 32 and 64 bit platforms
* test on both big and little endian platforms
* use multiple compilers. my company uses 5 completely different compilers with full warnings enabled in every nightly build
* check error return values even after trivial function calls like fclose() or shutdown()
* use unix/linux limit or ulimit command to simulate many aspects of an old, underpowered, or overloaded machine
* C is still low level so you can't just gloss over buffer overflows and sql injection issues or assume that library functions will take care of little issues. don't get lazy!

Further reading (in no particular order):
* http://c-faq.com/
* Elements of Programming Style (Kernighan & Plauger)
* C Unleashed (Heathfield, Kirby et al)
* http://www.joelonsoftware.com/articles/fog0000000043.html

Intégration Continue,Gestion des versions

3 Grandes catégories d'outils :

- Logiciel de suivi de problèmes (Bug tracking system)
- Gestion de versions (versioning)
- Intégration continu

→ Certains outils/serveurs proposent 2, voir 3 de ces fonctionnalités

Intégration Continue,Gestion des versions

1 – Bug tracker

Un logiciel de suivi de problèmes ou système de suivi de problèmes (de l'anglais 'issue tracking system') est un logiciel qui permet d'aider les utilisateurs et les développeurs à améliorer la qualité d'un logiciel.

Les utilisateurs soumettent leurs demandes d'assistance dans le logiciel. Les développeurs sont alors toujours au fait des problèmes rencontrés. (source Wikipedia)

Intégration Continue,Gestion des versions

Principaux outils open source :

-Bugzilla

-Jira

-Mantis

-Trac

-RedMine

-Intégré a gitlab

Intégration Continue,Gestion des versions

Fonctionnalités :

-Bug tracker : notification d'incident (bug report)

problèmes techniques

règles et réglementation

fonctionnels

- Gestion de tache (délai , répartition)

- Planification (Méthode agile, Roadmap)

- Demande d'amélioration (RFE : Request For Enhancement) ou Demande de nouvelles fonctionnalité (RFF)

Intégration Continue,Gestion des versions

Principe général : Création d'un ticket

→ Numéro du ticket

→ Description du bug et contexte

→ Reproduction du bug / copie d'écran

→ Sévérité (défini par l'auteur du ticket, modifiable par l'administrateur du projet)

→ Auteur du ticket

→ Développeur affecté à la correction

→ Évolution du traitement (urgent, résolu, invalide…)

2 Gestion de versions

-Consiste à maintenir l'ensemble des versions d'un ensemble de documents

→ Activité fastidieuse et complexe

-Intégrée dans certaines applications (Wiki, OpenOffice, Microsoft Word, . . .)

-Gestion des versions successives d'un document

-Conserve toutes les versions successives dans un référentiel ou dépôt (repository)

-Ne stocke en général que les différences

Intégration Continue,Gestion des versions

-Permet de revenir aisément à une version antérieure

-Travail collaboratif

-Chaque utilisateur travaille sur une copie locale (pas de verrouillage)

-Le système signale les conflits

-Architecture centralisée ou distribuée

Intégration Continue,Gestion des versions

3 grande catégories :

Local : Revision Control System (RCS)

Centralisé :

-Concurrent Version System (CVS)

-Sub-Version (SVN) ,

-Team Foundation System (TFS, Microsoft) (partiellement faux...)

- IBM Rational Clear Case

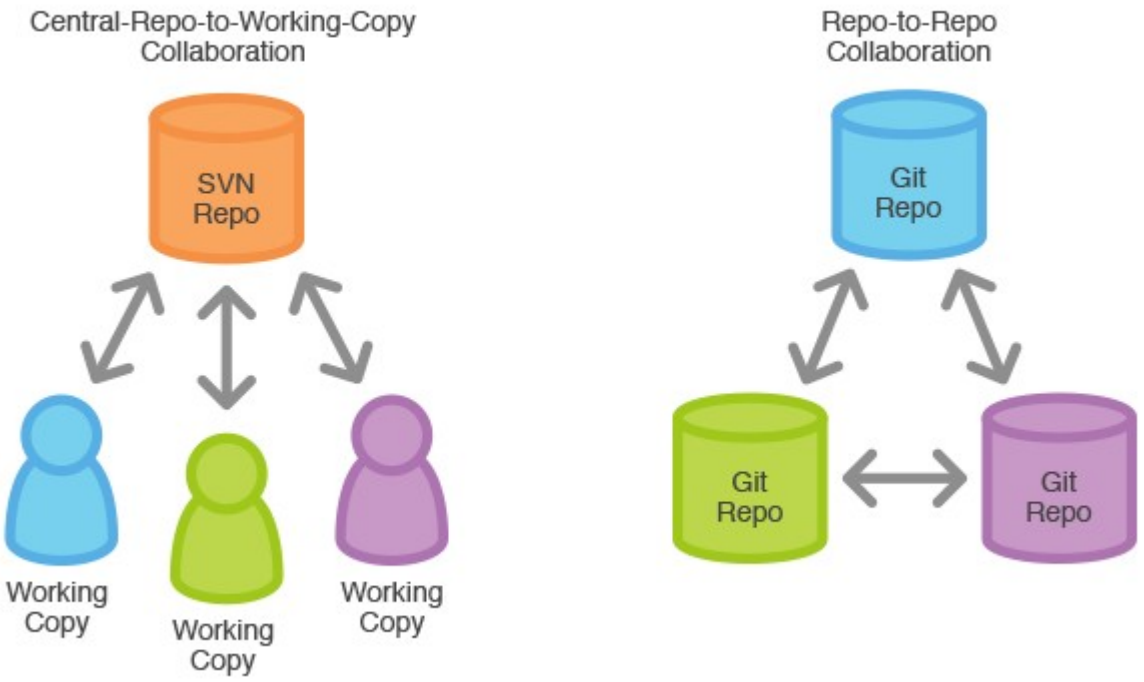
Distribué :

Git (Linus Torvalds pour les Kernels Linux)

Mercurial

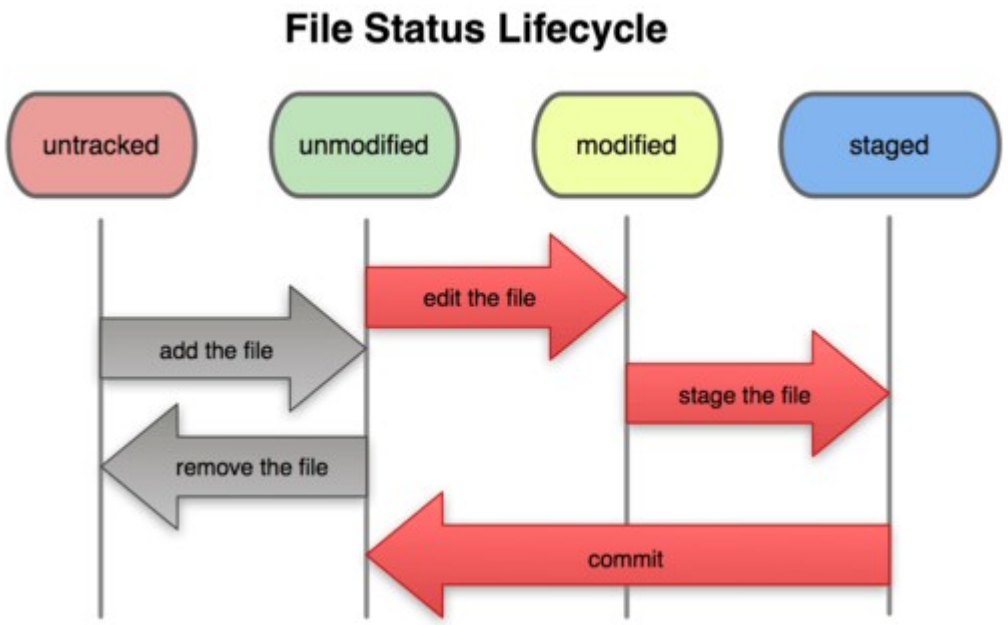
BitKeeper

Principes de Git

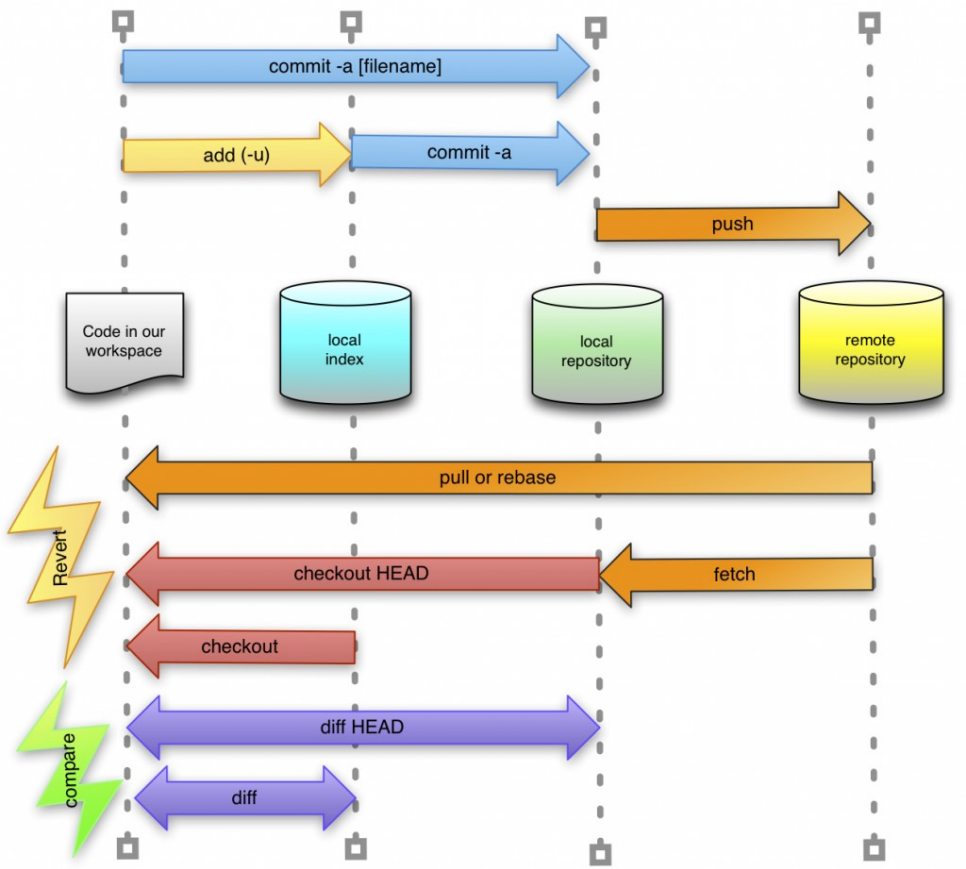


Systèmes centralisés vs systèmes distribués

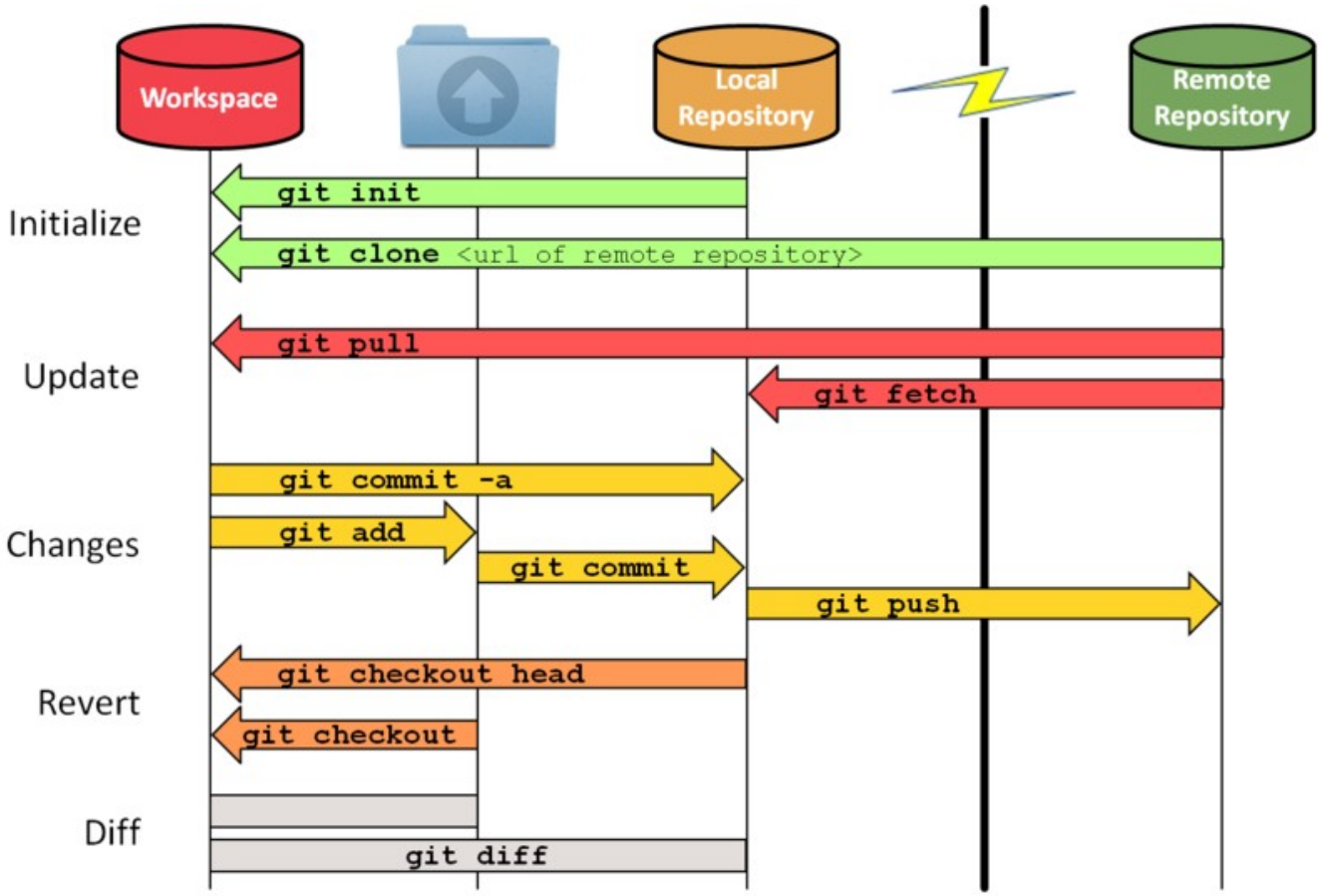
Cycle de vie d'un fichier



Principe de fonctionnement de Git

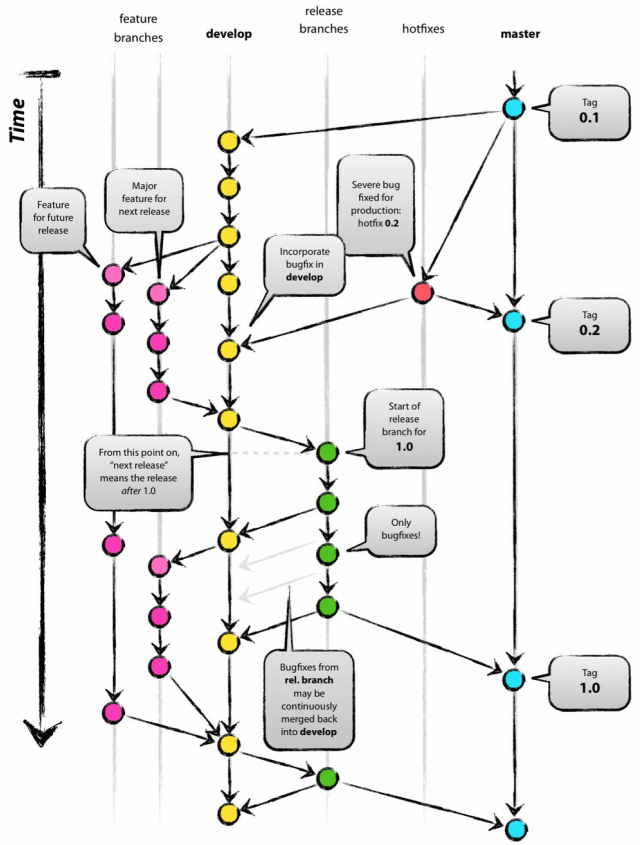


Intégration Continue,Gestion des versions



Intégration Continue,Gestion des versions

Workflow : gitflow



Intégration Continue,Gestion des versions

Outils Disponibles :

- Ligne de commande
- Intégration a l'OS (ex : tortoiseGit)
- Application indépendante (smartGit, gitkraken)
-outils intégré a l'IDE

-Serveurs : Gitlib, SCM-Server, Gitlab...

Intégration Continue,Gestion des versions

3 Intégration continue

Pratique de développement où :

- Les membres d'une équipe intègrent fréquemment leur travail :
- A pour but de détecter les problèmes d'intégration au plus tôt
- Est issue d'eXtreme Programming
- S'appuie généralement sur un serveur d'intégration continue
- Automatiser les compilations
- Rendre les compilations auto-testantes
- Tester dans un/plusieurs environnement(s) de production cloné
- Rendre disponible facilement le dernier exécutable
- Tout le monde doit voir ce qui se passe
- Automatiser le déploiement

Etapes de mise au point du code : Analyse statique

-Permet d'obtenir des informations sur un programme sans l'exécuter, recherche de motifs généraux (; après un for, . . .)

-Est un bon complément aux tests unitaires

-Outils d'analyse statique (moteur, ensemble de règles configurables)

-Utilisé dans le contexte de la revue (audit) de code

-Quelques outils : Findbugs, Pmd, sourcemeter, Lint, SonarQube...

Intégration Continue,Gestion des versions

Etapes de mise au point du code : Assertions

-Expression booléenne exprimant une propriété sémantique (invariant, pré-condition)

-Permet de vérifier que ce que l'on croit « vrai » l'est véritablement à l'exécution

-Technique simple pour assister la réalisation de programmes corrects

-Normalement pas de surcoût dans la version de production (destinées aux versions de débogage éliminées lors de la compilation pour les versions de production

Etapes de mise au point du code : les tests

-Visent à mettre en évidence des défauts de l'élément testé

-Objectifs : réduire le nombre de bogues, améliorer la qualité du logiciel

-Ne permet pas de prouver l'absence de bogue

-Méthode de travail :

- -tester après l'implémentation (approche classique)
- -écrire le test d'abord (approche Test Driven Development)

Intégration Continue,Gestion des versions

Quelques types de test :

-Fonctionnel : s'assure que les résultats attendus sont bien obtenus

-Non fonctionnel : analyse les propriétés non fonctionnelles d'un système (performances, sécurité, . . .)

-Unitaire : vérifie la conformité des éléments de base d'un programme

-Intégration : vérifie la cohérence des différents modules entre eux

-Recette : confirme la conformité du système avec les besoins

Quelques outils : JUnit, TestNG, Apache JMeter. CUnit

Couverture de code

-Mesure la qualité des tests effectués

-Le degré de couverture est représenté par des indices statistiques

→Statement Coverage mesure le nombre de lignes exécutées

→Condition Coverage vérifie l'évaluation des conditions

-Les portions de codes non testées sont mises en évidence

Intégration Continue,Gestion des versions

Profiling

-Collecter des informations sur le comportement d'une application pendant son exécution

→ CPU, mémoire, threads, . . .

-A un impact sur le comportement de l'application

Quelques outils : Valgrind, Intel VTune.

Intégration Continue,Gestion des versions

Coût et Avantages de l’intégration continue :

- Détection précoce des bugs du un un petit changement du code et suivi durant le projet
- Évite des désastres de dernière minute quand chaque développeur essaie de vérifier sa version légèrement incompatible
- Possibilité de revenir en arrière (nouveau bug, échec d'un test unitaire)
- Disponibilité permanente d'un « build »
- Les développeurs sont incités a produire un code plus modulaire et moins complexe
- Force l'utilisation de test automatique
- Retour d'information rapide de l'impact d'une modification du code
- Génération d'indicateur (code coverage, code complexity)